SensorBee Documentation

Release 0.4

Preferred Networks, Inc.

May 02, 2017

Contents

Contents:

Part I

Preface

This is the official documentation of SensorBee. It describes all the functionality that the current version of SensorBee officially supports.

This document is structured as follows:

- Preface, this part, provides general information of SensorBee.
- *Part I* is an introduction for new users through some tutorials.
- *Part II* documents the syntax and specification of the BQL language.
- *Part III* describes information for advancedusers about extensibility capabilities of the server.
- *Reference* contains reference information about BQL statements, built-in components, and client programs.

CHAPTER 1

What is SensorBee?

SensorBee is an open source, lightweight, stateful streaming data processing engine for the Internet of Things (IoT). SensorBee is designed to be used for streaming ETL (Extract/Transform/Load) at the edge of the network including Fog Computing. In ETL operations, SensorBee mainly focuses on data transformation and data enrichment, especially using machine learning. SensorBee is very small (stand-alone executable file size < 30MB) and runs on small computers such as Raspberry Pi.

The processing flow in SensorBee is written in BQL, a dialect of CQL (Continuous Query Language), which is similar to SQL but extended for streaming data processing. Its internal data structure (tuple) is compatible to JSON documents rather than rows in RDBMSs. Therefore, in addition to regular SQL expressions, BQL implements JSON notation and type conversions that work well with JSON. BQL is also schemaless at the moment to support rapid prototyping and integration.

Note: Supporting a schema in SensorBee is being planned to increase its robustness, debuggability, and speed. However, the version that will support the feature has not been decided yet.

SensorBee manages user-defined states (UDSs) and BQL utilizes those states to perform stateful processing on streaming data. An example of stateful processing is machine learning. Via a Python extension, SensorBee supports deep learning using Chainer, a flexible deep learning framework developed by Preferred Networks, Inc. and Preferred Infrastructure, Inc. The combination of SensorBee and Chainer enables users to support not only online analysis but also online training of deep learning models at the edge of the network with the help of GPUs. Preprocessing of data and feature extraction from preprocessed results can be written in BQL. The results can be computed in an online manner and directly connected to deep learning models implemented with Chainer.

By combining JSON-like data structure of BQL and machine learning, SensorBee becomes good at handling unstructured data such as text written in natural languages and even video streams, which are not well supported by most data processing engines. Therefore, SensorBee can operate, for example, between a video camera and Cloud-based (semi-structured) data analytics services so that those services don't have to analyze raw video images and can only utilize the information extracted from them by SensorBee.

SensorBee can be extended to work with existing databases or data processing solutions by developing data source or sink plugins. For example, it officially provides plugins for fluentd, an open source data collector, and has various input and output plugins for major databases and Cloud services.

SensorBee has **not** been designed for:

- very large scale data processing
- massively parallel streaming data processing
- accurate numerical computation without any error

CHAPTER 2

Conventions

The following conventions are used in the synopsis of a command:

- Brackets (`[` and `]`) indicate optional parts.
  - Some statements such as `SELECT` have `[` and `]` as a part of the statement. In that case, those brackets are enclosed with single quotes (`'`).
- Braces (`{` and `}`) and vertical lines (`|`) indicate that one of candidates in braces must be chosen (e.g. one of a, b, or c has to be selected `{a | b | c}`).
- Dots (`...`) mean that the preceding element can be repeated.
- Commands that are to be run in a normal system shell are prefixed with a dollar sign (`$`).

Types and keywords in BQL are written with `fixed-size fonts`.

CHAPTER 3

Further Information

Besides this documentation, there're other resources about SensorBee:

Website

http://sensorbee.io/ has general information about SensorBee.

Github

The sensorbee organization contains SensorBee's core source code repository and its official plugins.

Mailing Lists

There are two Google Groups for discussion and questions about SensorBee: https://groups.google.com/forum/#!forum/sensorbee (English) and https://groups.google.com/forum/#!forum/sensorbee-ja (Japanese).

Part II

Tutorial

The following chapters gives an introduction to SensorBee.

CHAPTER 4

Getting Started

To get started with SensorBee, this chapter introduces word counting as the first tutorial. It covers the following topics:

- How to install and set up SensorBee
- How to build a custom `sensorbee` command
- How to use the `sensorbee` command
- How to query the SensorBee server with `sensorbee shell` and BQL

Prerequisites

SensorBee requires Go 1.4 or later to be installed and its development environment (`$GOPATH` etc.) to be set up correctly. Also, Git needs to be installed.

This tutorial assumes that readers know about basic Linux commands and basics of SQL.

SensorBee itself doesn't have to be installed at this point.

Word Count Example

As the first tutorial, this section shows a word count example. All programs and configuration files required for this tutorial are provided in the `wordcount` package of the Github repository https://github.com/sensorbee/tutorial.

Installing Word Count Example Package

The first thing that needs to be done is to `go get` the word count example package in the repository:

```
$ go get github.com/sensorbee/tutorial/wordcount
```

This command clones the repository to `$GOPATH/src/github.com/sensorbee/tutorial/wordcount` and also downloads all dependencies. In the `config` subdirectory of that path, there are configuration files for building and running SensorBee. After `go get` successfully downloaded the package, copy those configuration files to another temporary directory (**replace /path/to/ with an appropriate path**):

```
$ mkdir -p /path/to/wordcount
$ cp $GOPATH/src/github.com/sensorbee/tutorial/wordcount/config/* \
    /path/to/wordcount/
$ ls /path/to/wordcount
build.yaml
sensorbee.yaml
wordcount.bql
```

Everything necessary to try this tutorial is ready now except SensorBee. The next step is to build a custom `sensorbee` command that includes the plugins needed for this tutorial.

Building a `sensorbee` Executable

To build a `sensorbee` executable, the `build_sensorbee` program needs to be installed. To do so, issue the following command:

```
$ go get gopkg.in/sensorbee/sensorbee.v0/...
```

The `build_sensorbee` program is used to build a custom `sensorbee` executable with plugins provided by developers.

Then, move to the directory that has configuration files previously copied from the tutorial package and execute `build_sensorbee`:

```
$ cd /path/to/wordcount
/path/to/wordcount$ build_sensorbee
/path/to/wordcount$ ls
build.yaml
sensorbee
sensorbee.yaml
sensorbee_main.go
wordcount.bql
```

There are two new files in the directory: `sensorbee` and `sensorbee_main.go`. Both of them are automatically generated by the `build_sensorbee` command. `sensorbee` is the command to run the SensorBee server or shell. Under the hood, this command is built from `sensorbee_main.go` using `go build`.

`build_sensorbee` builds a `sensorbee` command according to the configuration in `build.yaml`:

```
/path/to/wordcount$ cat build.yaml
plugins:
  - github.com/sensorbee/tutorial/wordcount/plugin
```

Inserting a new go path to the `plugin` section adds a new plugin to the `sensorbee` command, but this tutorial only uses the wordcount plugin above. Other tutorials will cover this configuration file in more depth.

Run the Server

After building the `sensorbee` command having plugins for this tutorial, run it as a server:

```
/path/to/wordcount$ ./sensorbee run
INFO[0000] Setting up the server context                 config={"logging":
{"log_dropped_tuples":false,"min_log_level":"info","summarize_dropped_tuples":
false,"target":"stderr"},"network":{"listen_on":":15601"},"storage":{"uds":
{"params":{},"type":"in_memory"}},"topologies":{}}
INFO[0000] Starting the server on :15601
```

`sensorbee run` runs the SensorBee server. It writes some log messages to stdout but they can be ignored at the moment. It provides a HTTP JSON API and listens on `:15601` by default. However, the API isn't directly used in this tutorial. Instead of controlling the server via the API, this tutorial shows how to use the `sensorbee shell` command and the **BQL** language, which is similar to SQL but has some extensions for streaming data.

To test if the server has successfully started, run the following command in another terminal:

```
$ curl http://localhost:15601/api/v1/runtime_status
{"gomaxprocs":1,"goroot":"/home/pfn/go","goversion":"go1.4.2",
"hostname":"sensorbee-tutorial","num_cgo_call":0,"num_cpu":4,
"num_goroutine":13,"pid":33267,"user":"pfn",
"working_directory":"/path/to/wordcount/"}
```

The server is correctly working if a response like this returned.

Setting Up a Topology

Once the server has started, open another window or use screen/tmux to have another terminal to interact with the server. The server does nothing just after it started up. There are a few steps required to enjoy interacting with stream data.

Firstly, to allow the server to process some data, it needs to have a **topology**. A topology is a similar concept to a "database" in RDBMSs. It has processing components such as data sources, continuous views, and so on. Use the `sensorbee topology create` command to create a new topology `wordcount` for the tutorial:

```
/path/to/wordcount$ ./sensorbee topology create wordcount
/path/to/wordcount$ echo $?
0
```

`$?` (the return code of the `./sensorbee` command) will be `0` if the command was successful. Otherwise, it will be non-zero. Be careful to write `./sensorbee` (and not omit the `./`) in order to use the executable from your current directory, which has the correct plugins baked in.

Note: Almost everything in SensorBee is volatile at the moment and is reset every time the server restarts. A topology is dropped when the server shuts down, too. Therefore, `sensorbee topology create wordcount` needs to be run on each startup of the server until it is specified in a config file for `sensorbee run` later.

In the next step, start `sensorbee shell`:

```
/path/to/wordcount$ ./sensorbee shell -t wordcount
wordcount>
```

`-t wordcount` means that the shell connects to the `wordcount` topology just created. Now it's time to try some BQL statements. To start, try the `EVAL` statement, which evaluates arbitrary expressions supported by BQL:

```
wordcount> EVAL 1 + 1;
2
wordcount> EVAL power(2.0, 2.5);
```

```
5.65685424949238
wordcount> EVAL "Hello" || ", world!";
"Hello, world!"
```

BQL also supports one line comments:

```
wordcount> -- This is a comment
wordcount>
```

Finally, create a source which generates stream data or reads input data from other stream data sources:

```
wordcount> CREATE SOURCE sentences TYPE wc_sentences;
wordcount>
```

This `CREATE SOURCE` statement creates a source named `sentences`. Its type is `wc_sentences` and it is provided by a plugin in the `wordcount` package. This source emits, on a regular basis, a random sentence having several words with the name of a person who wrote a sentence. To receive data (i.e. tuples) emitted from the source, use the `SELECT` statement:

```
wordcount> SELECT RSTREAM * FROM sentences [RANGE 1 TUPLES];
{"name":"isabella","text":"dolor consequat ut in ad in"}
{"name":"sophia","text":"excepteur deserunt officia cillum lorem excepteur"}
{"name":"sophia","text":"exercitation ut sed aute ullamco aliquip"}
{"name":"jacob","text":"duis occaecat culpa dolor veniam elit"}
{"name":"isabella","text":"dolore laborum in consectetur amet ut nostrud ullamco"}
...
```

Type `C-c` (also known as `Ctrl+C` to some people) to stop the statement. Details of the statement are not described for now, but this is basically same as the `SELECT` statement in SQL except two things: `RSTREAM` and `RANGE`. Those concepts will briefly be explained in the next section.

Querying: Basics

This subsection introduces basics of querying with BQL, i.e., the `SELECT` statement. Since it is very similar to SQL's `SELECT` and some basic familiarity with SQL is assumed, two concepts that don't exist in SQL are described first. Then, some features that are also present in SQL will be covered.

Stream-Related Operators

BQL's `SELECT` statement has two components related to stream data processing: **stream-to-relation operators** and **relation-to-stream operators**.

Note: Skip the description of stream-to-relations and relation-to-stream operators if these aren't clear enough at the moment.

A stream-to-relation operator is a operator that literally converts a stream of tuples to relations (i.e., records in a table of the database). What it actually does is to define a window having a finite set of tuples on a stream. The operator is written as `[RANGE n TUPLES]` or `[RANGE n SECONDS]`. `[RANGE n TUPLES]` creates a window containing the last n tuples in the stream. `[RANGE n SECONDS]`, on the other hand, creates a window holding tuples observed in past n seconds (more precisely, the duration between the oldest and newest tuple is at most n seconds).

```
SELECT RSTREAM * FROM sentences [RANGE 1 TUPLES];
```

The previous example uses a stream-to-relation operator `[RANGE 1 TUPLES]`, i.e., each window only has a single tuple in it. This window can be thought of as the input relation for a SQL-like `SELECT` statement.

Another concept that doesn't exist in SQL is a relation-to-stream operator. It converts a relation, which is a result of the `SELECT` statement, to a stream of tuples. There are three types of operators:

- `RSTREAM`
- `ISTREAM`
- `DSTREAM`

`RSTREAM` emits all tuples in the relation resulting every time a new tuple arrives and the result is updated. `ISTREAM` only emits tuples that are in the current window and weren't in the previous window, that is, it emits tuples having newly been inserted into the current relation. `DSTREAM` only emits tuples in the previous relation, that is, it emits tuples deleted in the current relation.

In the previous example, `RSTREAM` is used as a relation-to-stream operator. Since the resulting relation is same as the input relation (i.e. window), it only has one tuple in it.

Note: The difference between using `RSTREAM` and `ISTREAM` should be described a little here. Assume that a source `s` emits following 4 tuples with timestamps t_1 to t_4:

```
t1: {"a": 1}
t2: {"a": 2}
t3: {"a": 2}
t4: {"a": 3}
```

When selecting these tuples by

```
SELECT RSTREAM * FROM s [RANGE 1 TUPLES];
```

the resulting output for each timestamp would be:

```
t1: {"a": 1}
t2: {"a": 2}
t3: {"a": 2}
t4: {"a": 3}
```

These tuples are identical to what the source `s` has emitted. On the other hand, when `ISTREAM` is used instead of `RSTREAM` in the previous `SELECT` statement, the statement emits only three tuples:

```
t1: {"a": 1}
t2: {"a": 2}
t4: {"a": 3}
```

The reason why it happens is that the resulting relation wasn't updated at t_3 since both relations at t_2 and t_3 have the same tuple `{"a": 2}` as a result.

In other words, when using `ISTREAM` with `[RANGE 1 TUPLES]`, a resulting tuple is emitted only when it's different from the previous resulting tuple. In contrast, `RSTREAM` emits the resulting tuple every time regardless of its value.

Therefore, when the stream-to-relation operator is `[RANGE 1 TUPLES]`, basically prefer `RSTREAM` to `ISTREAM` unless there's a strong reason to use `ISTREAM`. It leads to less confusing results.

To learn more about these operators, see *Queries* after finishing this tutorial.

Selection

The `SELECT` statement can partially pick up some fields of input tuples:

```
wordcount> SELECT RSTREAM name FROM sentences [RANGE 1 TUPLES];
{"name":"isabella"}
{"name":"isabella"}
{"name":"jacob"}
{"name":"isabella"}
{"name":"jacob"}
...
```

In this example, only the `name` field is picked up from input tuples that have "name" and "text" fields.

BQL is schema-less at the moment and the format of output tuples emitted by a source must be documented by that source's author. The `SELECT` statement is only able to report an error at runtime when processing a tuple, not at the time when it is sent to the server. This is a drawback of being schema-less.

Filtering

The `SELECT` statement supports filtering with the `WHERE` clause as SQL does:

```
wordcount> SELECT RSTREAM * FROM sentences [RANGE 1 TUPLES] WHERE name = "sophia";
{"name":"sophia","text":"anim eu occaecat do est enim do ea mollit"}
{"name":"sophia","text":"cupidatat et mollit consectetur minim et ut deserunt"}
{"name":"sophia","text":"elit est laborum proident deserunt eu sed consectetur"}
{"name":"sophia","text":"mollit ullamco ut sunt sit in"}
{"name":"sophia","text":"enim proident cillum tempor esse occaecat exercitation"}
...
```

This filters out sentences from the user `sophia`. Any expression which results in a `bool` value can be written in the `WHERE` clause.

Grouping and Aggregates

The `GROUP BY` clause is also available in BQL:

```
wordcount> SELECT ISTREAM name, count(*) FROM sentences [RANGE 60 SECONDS]
    GROUP BY name;
{"count":1,"name":"isabella"}
{"count":1,"name":"emma"}
{"count":2,"name":"isabella"}
{"count":1,"name":"jacob"}
{"count":3,"name":"isabella"}
...
{"count":23,"name":"jacob"}
{"count":32,"name":"isabella"}
{"count":33,"name":"isabella"}
{"count":24,"name":"jacob"}
{"count":14,"name":"sophia"}
...
```

This statement creates groups of users in a 60 second-long window. It returns pairs of a user and the number of sentences that have been written by that user in the past 60 seconds. In addition to `count`, BQL also provides built-in aggregate functions such as `min`, `max`, and so on.

Also note that the statement above uses `ISTREAM` rather than `RSTREAM`. The statement only reports a new count for an updated user while `RSTREAM` reports counts for all users every time it receives a tuple. Seeing the example of outputs from the statements with `RSTREAM` and `ISTREAM` makes it easier to understand their behaviors. When the statement receives `isabella`, `emma`, `isabella`, `jacob`, and `isabella` in this order, `RSTREAM` reports results as shown below (with some comments):

```
wordcount> SELECT RSTREAM name, count(*) FROM sentences [RANGE 60 SECONDS]
    GROUP BY name;
-- receive "isabella"
{"count":1,"name":"isabella"}
-- receive "emma"
{"count":1,"name":"isabella"}
{"count":1,"name":"emma"}
-- receive "isabella"
{"count":2,"name":"isabella"}
{"count":1,"name":"emma"}
-- receive "jacob"
{"count":2,"name":"isabella"}
{"count":1,"name":"emma"}
{"count":1,"name":"jacob"}
-- receive "isabella"
{"count":3,"name":"isabella"}
{"count":1,"name":"emma"}
{"count":1,"name":"jacob"}
```

On the other hand, `ISTREAM` only emits tuples updated in the current resulting relation:

```
wordcount> SELECT ISTREAM name, count(*) FROM sentences [RANGE 60 SECONDS]
    GROUP BY name;
-- receive "isabella"
{"count":1,"name":"isabella"}
-- receive "emma", the count of "isabella" isn't updated
{"count":1,"name":"emma"}
-- receive "isabella"
{"count":2,"name":"isabella"}
-- receive "jacob"
{"count":1,"name":"jacob"}
-- receive "isabella"
{"count":3,"name":"isabella"}
```

This is one typical situation where `ISTREAM` works well.

Tokenizing Sentences

To perform word counting, sentences that are contained in `sources` need to be split up into words. Imagine there was a user-defined function (UDF) `tokenize(sentence)` returning an array of strings:

```
SELECT RSTREAM name, tokenize(text) AS words FROM sentences ...
```

A resulting tuple of this statement would look like:

```
{
    "name": "emma",
    "words": ["exercitation", "ut", "sed", "aute", "ullamco", "aliquip"]
}
```

However, to count words with the `GROUP BY` clause and the `count` function, the tuple above further needs to be split into multiple tuples so that each tuple has one word instead of an array:

```
{"name": "emma", "word": "exercitation"}
{"name": "emma", "word": "ut"}
{"name": "emma", "word": "sed"}
{"name": "emma", "word": "aute"}
{"name": "emma", "word": "ullamco"}
{"name": "emma", "word": "aliquip"}
```

With such a stream, the statement below could easily compute the count of each word:

```
SELECT ISTREAM word, count(*) FROM some_stream [RANGE 60 SECONDS]
    GROUP BY word;
```

To create a stream like this from tuples emitted from `sentences`, BQL has the concept of a **user-defined stream-generating function (UDSF)**. A UDSF is able to emit multiple tuples from one input tuple, something that cannot be done with the `SELECT` statement itself. The `wordcount` package from this tutorial provides a UDSF `wc_tokenizer(stream, field)`, where `stream` is the name of the input stream and `field` is the name of the field containing a sentence to be tokenized. Both arguments need to be string values.

```
wordcount> SELECT RSTREAM * FROM wc_tokenizer("sentences", "text") [RANGE 1 TUPLES];
{"name":"ethan","text":"duis"}
{"name":"ethan","text":"lorem"}
{"name":"ethan","text":"adipiscing"}
{"name":"ethan","text":"velit"}
{"name":"ethan","text":"dolor"}
...
```

In this example, `wc_tokenizer` receives tuples from the `sentences` stream and tokenizes sentences contained in the `text` field of input tuples. Then, it emits each tokenized word as a separate tuple.

Note: As shown above, a UDSF is one of the most powerful tools to extend BQL's capability. It can virtually do anything that can be done for stream data. To learn how to develop it, see *User-Defined Stream-Generating Functions*.

Creating a Stream

Although it is now possible to count tokenized words, it is easier to have something like a "view" in SQL to avoid writing `wc_tokenizer("sentences", "text")` every time issuing a new query. BQL has a **stream** (a.k.a a **continuous view**), which just works like a view in RDBMSs. A stream can be created using the `CREATE STREAM` statement:

```
wordcount> CREATE STREAM words AS
    SELECT RSTREAM name, text AS word
    FROM wc_tokenizer("sentences", "text") [RANGE 1 TUPLES];
wordcount>
```

This statement creates a new stream called `words`. The stream renames `text` field to `word`. The stream can be referred by the `FROM` clause of the `SELECT` statement as follows:

```
wordcount> SELECT RSTREAM * FROM words [RANGE 1 TUPLES];
{"name":"isabella","word":"pariatur"}
{"name":"isabella","word":"adipiscing"}
{"name":"isabella","word":"id"}
```

```
{"name":"isabella","word":"et"}
{"name":"isabella","word":"aute"}
...
```

A stream can be specified in the `FROM` clause of multiple `SELECT` statements and all those statements will receive the same tuples from the stream.

Counting Words

After creating the `words` stream, words can be counted as follows:

```
wordcount> SELECT ISTREAM word, count(*) FROM words [RANGE 60 SECONDS]
    GROUP BY word;
{"count":1,"word":"aute"}
{"count":1,"word":"eu"}
{"count":1,"word":"quis"}
{"count":1,"word":"adipiscing"}
{"count":1,"word":"ut"}
...
{"count":47,"word":"mollit"}
{"count":35,"word":"tempor"}
{"count":100,"word":"in"}
{"count":38,"word":"sint"}
{"count":79,"word":"dolor"}
...
```

This statement counts the number of occurrences of each word that appeared in the past 60 seconds. By creating another stream based on the `SELECT` statement above, further statistical information can be obtained:

```
wordcount> CREATE STREAM word_counts AS
    SELECT ISTREAM word, count(*) FROM words [RANGE 60 SECONDS]
    GROUP BY word;
wordcount> SELECT RSTREAM max(count), min(count)
    FROM word_counts [RANGE 60 SECONDS];
{"max":52,"min":52}
{"max":120,"min":52}
{"max":120,"min":50}
{"max":165,"min":50}
{"max":165,"min":45}
...
{"max":204,"min":31}
{"max":204,"min":30}
{"max":204,"min":29}
{"max":204,"min":28}
{"max":204,"min":27}
...
```

The `CREATE STREAM` statement above creates a new stream `word_counts`. The next `SELECT` statement computes the maximum and minimum counts over words observed in past 60 seconds.

Using a BQL File

All statements above will be cleared once the SensorBee server is restarted. By using a BQL file, a topology can be set up on each startup of the server. A BQL file can contain multiple BQL statements. For the statements used in this tutorial, the file would look as follows:

```
CREATE SOURCE sentences TYPE wc_sentences;

CREATE STREAM words AS
    SELECT RSTREAM name, text AS word
        FROM wc_tokenizer("sentences", "text") [RANGE 1 TUPLES];

CREATE STREAM word_counts AS
    SELECT ISTREAM word, count(*)
        FROM words [RANGE 60 SECONDS]
        GROUP BY word;
```

Note: A BQL file cannot have the `SELECT` statement because it runs continuously until it is manually stopped.

To run the BQL file on the server, a configuration file for `sensorbee run` needs to be provided in YAML format. The name of the configuration file is often `sensorbee.yaml`. For this tutorial, the file has the following content:

```
topologies:
  wordcount:
    bql_file: wordcount.bql
```

`topologies` is one of the top-level parameters related to topologies in the server. It has names of topologies to be created on startup. In the file above, there's only one topology `wordcount`. Each topology has a `bql_file` parameter to specify which BQL file to execute. The `wordcount.bql` file was copied to the current directory before and the configuration file above specifies it.

With this configuration file, the SensorBee server can be started as follows:

```
/path/to/wordcount$ ./sensorbee run -c sensorbee.yaml
./sensorbee run -c sensorbee.yaml
INFO[0000] Setting up the server context                 config={"logging":
{"log_dropped_tuples":false,"min_log_level":"info","summarize_dropped_tuples":
false,"target":"stderr"},"network":{"listen_on":":15601"},"storage":{"uds":
{"params":{},"type":"in_memory"}},"topologies":{"wordcount":{"bql_file":
"wordcount.bql"}}}
INFO[0000] Setting up the topology                       topology=wordcount
INFO[0000] Starting the server on :15601
```

As written in log messages, the topology `wordcount` is created before the server actually starts.

Summary

This tutorial provided a brief overview of SensorBee through word counting. First of all, it showed how to build a custom `sensorbee` command to work with the tutorial. Second, running the server and setting up a topology with BQL was explained. Then, querying streams and how to create a new stream using `SELECT` was introduced. Finally, word counting was performed over a newly created stream and BQL statements that create a source and streams were persisted in a BQL file so that the server can re-execute those statements on startup.

In subsequent sections, there are more tutorials and samples to learn how to integrate SensorBee with other tools and libraries.

CHAPTER 5

Using Machine Learning

This chapter describes how to use machine learning on SensorBee.

In this tutorial, SensorBee retrieves tweets written in English from Twitter's public stream using Twitter's Sample API. SensorBee adds two labels to each tweet: age and gender. Tweets are labeled ("classified") using machine learning.

The following sections shows how to install dependencies, set them up, and apply machine learning to tweets using SensorBee.

Note: Due to the way the Twitter client receives tweets from Twitter, the behavior of this tutorial demonstration does not seem very smooth. For example, the client gets around 50 tweets in 100ms, then stops for 900ms, and repeats the same behavior every second. So, it is easy to get the misperception that SensorBee and its machine learning library are doing mini-batch processing, but they actually do not.

Prerequisities

This tutorial requires following software to be installed:

- Ruby 2.1.4 or later
  - https://www.ruby-lang.org/en/documentation/installation/
- Elasticsearch 2.2.0 or later
  - https://www.elastic.co/products/elasticsearch

To check that the data arrives properly in Elasticsearch and to show see this data could be visualized, also install:

- Kibana 4.4.0 or later
  - https://www.elastic.co/products/kibana

However, explanations on how to configure and use Kibana for data visualization are not part of this tutorial. It is assumed that Elasticsearch and Kibana are running on the same host as SensorBee. However, SensorBee can be configured to use services running on a different host.

In addition, Go 1.4 or later and Git are required, as described in *Getting Started*.

Quick Set Up Guide

If there are no Elasticsearch and Kibana instances that can be used for this tutorial, they need to be installed. Skip this subsection if they are already installed. In case an error occurs, look up the documentation at http://www.elastic.co/.

Installing and Running Elasticsearch

Download the package from https://www.elastic.co/downloads/elasticsearch and extract the compressed file. Then, run `bin/elasticsearch` in the directory with the extracted files:

```
/path/to/elasticserach-2.2.0$ bin/elasticsearch
... log messages ...
```

Note that a Java runtime is required to run the command above.

To see if Elasticsearch is running, access the server with `curl` command:

```
$ curl http://localhost:9200/
{
  "name" : "Peregrine",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "2.2.0",
    "build_hash" : "8ff36d139e16f8720f2947ef62c8167a888992fe",
    "build_timestamp" : "2016-01-27T13:32:39Z",
    "build_snapshot" : false,
    "lucene_version" : "5.4.1"
  },
  "tagline" : "You Know, for Search"
}
```

Installing and Running Kibana

Download the package from https://www.elastic.co/downloads/kibana and extract the compressed file. Then, run `bin/kibana` in the directory with the extracted files:

```
/path/to/kibana-4.4.0$ bin/kibana
... log messages ...
```

Access http://localhost:5601/ with a Web browser. Kibana is running correctly if it shows a page saying "Configure an index pattern". Since Elasticsearch does not have any data yet, no more operation is necessary at the moment. In the *Running SensorBee* section further configuration steps are described.

Installation and Setup

At this point, the environment described in the previous section is assumed to be installed correctly and working. Now, some more components needs to be set up before continuing this tutorial.

Installing the Tutorial Package

To setting up the system, `go get` the tutorial package first:

```
$ go get github.com/sensorbee/tutorial/ml
```

The package contains configuration files in the `config` subdirectory that are necessary for the tutorial. Create a temporary directory and copy those files to the directory (**replace /path/to/ with an appropriate path**):

```
$ mkdir -p /path/to/sbml
$ cp -r $GOPATH/src/github.com/sensorbee/tutorial/ml/config/* /path/to/sbml/
$ cd /path/to/sbml
/path/to/sbml$ ls
Gemfile
build.yaml
fluent.conf
sensorbee.yaml
train.bql
twitter.bql
uds
```

Installing and Running fluentd

This tutorial, and SensorBee, relies on fluentd. fluentd is an open source data collector that provides many input and output plugins to connect with a wide variety of databases including Elasticsearch. Skip this subsection if fluentd is already installed.

To install fluentd for this tutorial, bundler needs to be installed with the `gem` command. To see if it's already installed, run `gem list`. Something like `bundler (1.11.2)` shows up if it's already installed:

```
/path/to/sbml$ gem list | grep bundler
bundler (1.11.2)
/path/to/sbml$
```

Otherwise, install bundler with `gem install bundler`. It may require admin privileges (i.e. `sudo`):

```
/path/to/sbml$ gem install bundler
Fetching: bundler-1.11.2.gem (100%)
Successfully installed bundler-1.11.2
Parsing documentation for bundler-1.11.2
Installing ri documentation for bundler-1.11.2
Done installing documentation for bundler after 3 seconds
1 gem installed
/path/to/sbml$
```

After installing bundler, run the following command to install fluentd and its plugins under the `/path/to/sbml` directory (in order to build the gems, you may have to install Ruby header files before):

```
/path/to/sbml$ bundle install --path vendor/bundle
Fetching gem metadata from https://rubygems.org/.....
Fetching version metadata from https://rubygems.org/..
Resolving dependencies...
Installing cool.io 1.4.3 with native extensions
Installing multi_json 1.11.2
Installing multipart-post 2.0.0
Installing excon 0.45.4
```

```
Installing http_parser.rb 0.6.0 with native extensions
Installing json 1.8.3 with native extensions
Installing msgpack 0.5.12 with native extensions
Installing sigdump 0.2.4
Installing string-scrub 0.0.5 with native extensions
Installing thread_safe 0.3.5
Installing yajl-ruby 1.2.1 with native extensions
Using bundler 1.11.2
Installing elasticsearch-api 1.0.15
Installing faraday 0.9.2
Installing tzinfo 1.2.2
Installing elasticsearch-transport 1.0.15
Installing tzinfo-data 1.2016.1
Installing elasticsearch 1.0.15
Installing fluentd 0.12.20
Installing fluent-plugin-elasticsearch 1.3.0
Bundle complete! 2 Gemfile dependencies, 20 gems now installed.
Bundled gems are installed into ./vendor/bundle.
/path/to/sbml$
```

With `--path vendor/bundle` option, all Ruby gems required for this tutorial is locally installed in the `/path/to/sbml/vendor/bundle` directory. To confirm whether fluentd is correctly installed, run the command below:

```
/path/to/sbml$ bundle exec fluentd --version
fluentd 0.12.20
/path/to/sbml$
```

If it prints the version, the installation is complete and fluentd is ready to be used.

Once fluentd is installed, run it with the provided configuration file:

```
/path/to/sbml$ bundle exec fluentd -c fluent.conf
2016-02-05 16:02:10 -0800 [info]: reading config file path="fluent.conf"
2016-02-05 16:02:10 -0800 [info]: starting fluentd-0.12.20
2016-02-05 16:02:10 -0800 [info]: gem 'fluentd' version '0.12.20'
2016-02-05 16:02:10 -0800 [info]: gem 'fluent-plugin-elasticsearch' version '1.3.0'
2016-02-05 16:02:10 -0800 [info]: adding match pattern="sensorbee.tweets" type="...
2016-02-05 16:02:10 -0800 [info]: adding source type="forward"
2016-02-05 16:02:10 -0800 [info]: using configuration file: <ROOT>
  <source>
    @type forward
    @id forward_input
  </source>
  <match sensorbee.tweets>
    @type elasticsearch
    host localhost
    port 9200
    include_tag_key true
    tag_key @log_name
    logstash_format true
    flush_interval 1s
  </match>
</ROOT>
2016-02-05 16:02:10 -0800 [info]: listening fluent socket on 0.0.0.0:24224
```

Some log messages are truncated with `...` at the end of each line.

The configuration file `fluent.conf` is provided as a part of this tutorial. It defines a data source using `in_forward` and a destination that is connected to Elasticsearch. If the Elasticserver is running on a different

host or using a port number different from 9200, edit `fluent.conf`:

```
<source>
  @type forward
  @id forward_input
</source>
<match sensorbee.tweets>
  @type elasticsearch
  host {custom host name}
  port {custom port number}
  include_tag_key true
  tag_key @log_name
  logstash_format true
  flush_interval 1s
</match>
```

Also, feel free to change other parameters to adjust the configuration to the actual environment. Parameters for the Elasticsearch plugin are described at https://github.com/uken/fluent-plugin-elasticsearch.

Create Twitter API Key

This tutorial requires Twitter's API keys. To create keys, visit Application Management. Once a new application is created, click the application and its "Keys and Access Tokens" tab. The page should show 4 keys:

- Consumer Key (API Key)
- Consumer Secret (API Secret)
- Access Token
- Access Token Secret

Then, create the `api_key.yaml` in the `/path/to/sbml` directory and copy keys to the file as follows:

```
/path/to/sbml$ cat api_key.yaml
consumer_key: <Consumer Key (API Key)>
consumer_secret: <Consumer Secret (API Secret)>
access_token: <Access Token>
access_token_secret: <Access Token Secret>
```

Replace each key's value with the actual values shown in Twitter's application management page.

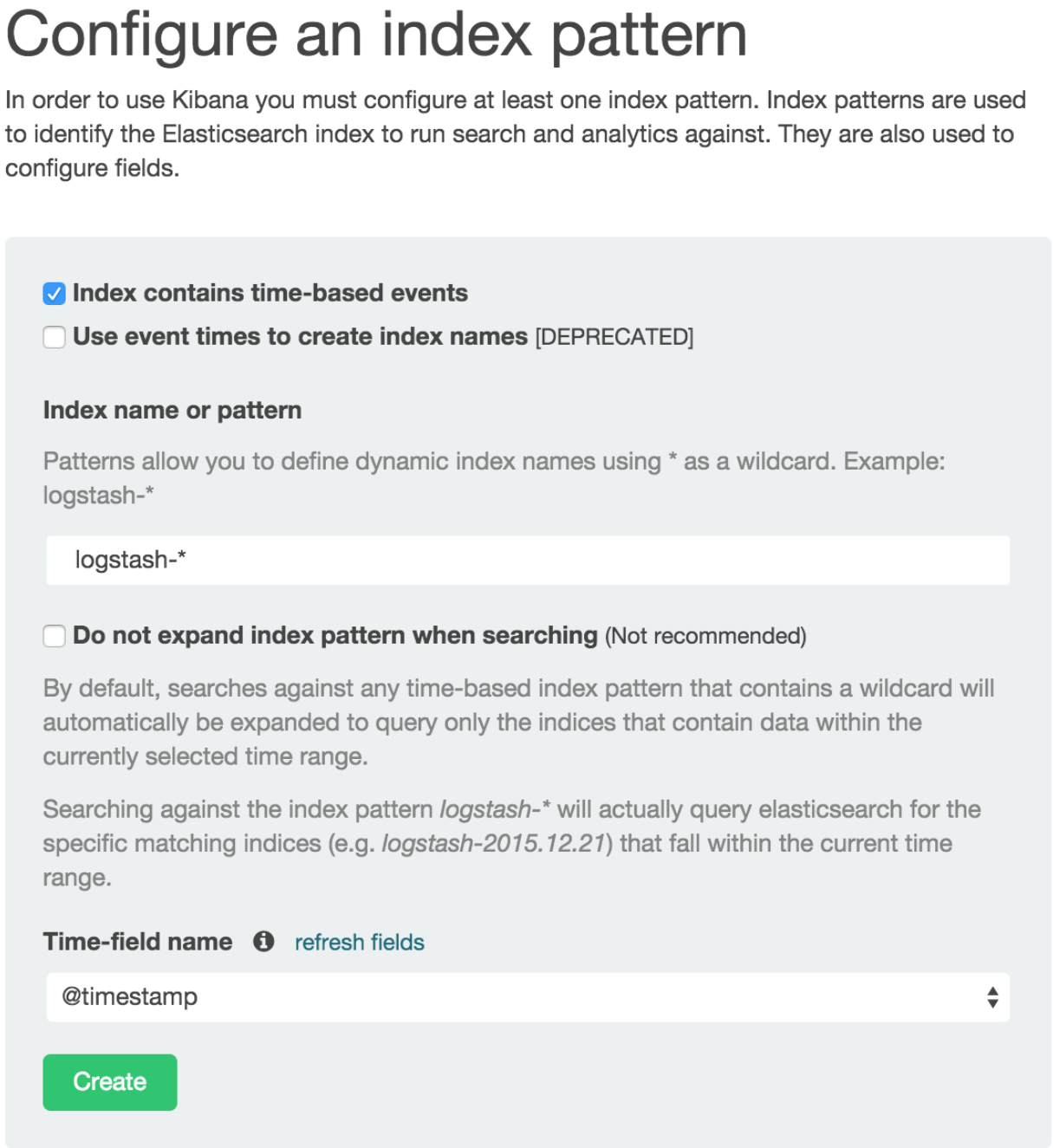
Running SensorBee

All requirements for this tutorial have been installed and set up. The next step is to install `build_sensorbee`, then build and run the `sensorbee` executable:

```
/path/to/sbml$ go get gopkg.in/sensorbee/sensorbee.v0/...
/path/to/sbml$ build_sensorbee
sensorbee_main.go
/path/to/sbml$ ./sensorbee run -c sensorbee.yaml
INFO[0000] Setting up the server context                 config={"logging":
{"log_dropped_tuples":false,"min_log_level":"info","summarize_dropped_tuples":
false,"target":"stderr"},"network":{"listen_on":":15601"},"storage":{"uds":
{"params":{"dir":"uds"},"type":"fs"}},"topologies":{"twitter":{"bql_file":
"twitter.bql"}}}
```

```
INFO[0000] Setting up the topology                       topology=twitter
INFO[0000] Starting the server on :15601
```

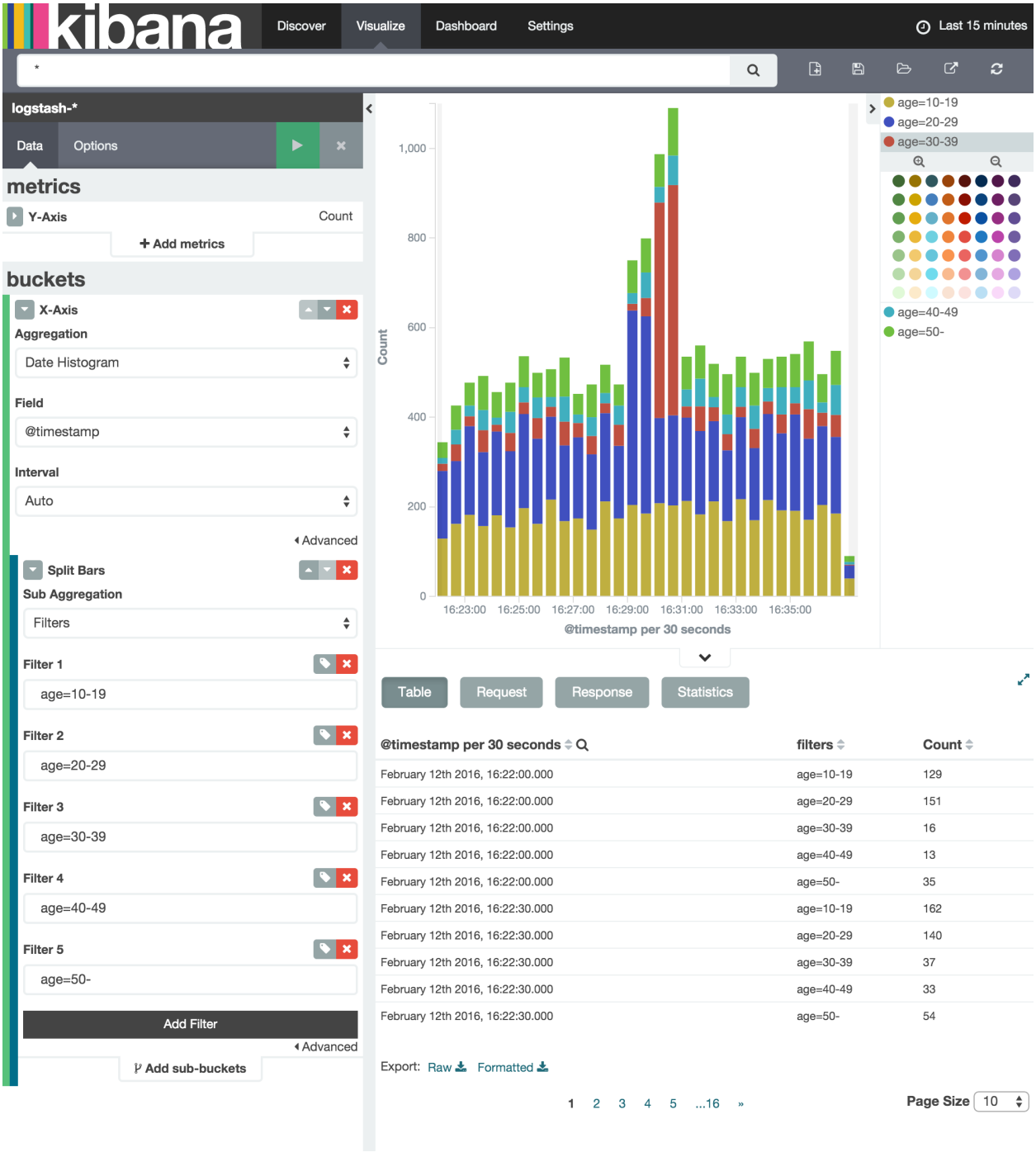
Because SensorBee loads pre-trained machine learning models on its startup, it may take a while to set up a topology. After the server shows the message `Starting the server on :15601`, access Kibana at http://localhost:5601/. If the setup operations performed so far have been sucessful, it returns the page as shown below with a green "Create" button:



(If the button is not visible, see the section on *Troubleshooting* below.) Click the "Create" button to work with data coming from SensorBee. After the action is completed, you should see a list of fields that were found in the data

stored so far. If you click "Discover" in the top menu, a selection of the tweets and a diagram with the tweet frequency should be visible.

Kibana can now be used to visualize and search through the data in Elasticsearch. Although this tutorial doesn't describe the usage of Kibana, many tutorials and examples can be found on the Web. The picture below shows an example chart showing some classification metrics:



Troubleshooting

If Kibana doesn't show the "Create" button, something may not be working properly. First, enter `sensorbee shell` to see SensorBee is working:

```
/path/to/sbml$ ./sensorbee shell -t twitter
twitter>
```

Then, issue the following `SELECT` statement:

```
twitter> SELECT RSTREAM * FROM public_tweets [RANGE 1 TUPLES];
... tweets show up here ...
```

If the statement returns an error or it doesn't show any tweet:

1. the host may not be connected to Twitter. Check the internet connection with commands such as `ping`.
2. The API key written in `api_key.yaml` may be wrong.

When the statement above shows tweets, query another stream:

```
twitter> SELECT RSTREAM * FROM labeled_tweets [RANGE 1 TUPLES];
... tweets show up here ...
```

If the statement doesn't show any tweets, the format of tweets may have been changed since the time of this writing. If so, modify BQL statements in `twitter.bql` to support the new format. *BQL Statements and Plugins* describes what each statement does.

When the statement above prints tweets, fluentd or Elasticsearch may have not been staretd yet. Check they're running correctly.

For other errors, report them to https://github.com/sensorbee/tutorial.

BQL Statements and Plugins

This section describes how SensorBee produced the output that was seen in the previous section: How it loads tweets from Twitter, preprocesses tweets for machine learning, and finally classifies tweets to extract demographic information of each tweets. `twitter.bql` in the `config` directory contains all BQL statements used in this tutorial.

The following subsections explains what each statement does. To interact with some streams created by `twitter.bql`, open another terminal (while the `sensorbee` instance from the previous section is still running) and launch `sensorbee shell`:

```
/path/to/sbml$ ./sensorbee shell -t twitter
twitter>
```

In the following sections of this tutorial, statements prefixed with `twitter>` can be executed in the SensorBee shell; statements without this prefix are statements from the `twitter.bql` file.

Creating a Twitter Source

This tutorial does not work without retrieving the public timeline of Twitter using the Sample API. The Sample API is provided for free to retrieve a portion of tweets sampled from the public timeline.

The github.com/sensorbee/twitter package provides a plugin for public time line retrieval. The source provided by that plugin has the type `twitter_public_stream`. The plugin can be registered to the SensorBee

server by adding `github.com/sensorbee/twitter/plugin` to the `build.yaml` configuration file for `build_sensorbee`. Now consider the first statement in the `twitter.bql` file:

```
CREATE SOURCE public_tweets TYPE twitter_public_stream
    WITH key_file = "api_key.yaml";
```

This statement creates a new source with the name `public_tweets`. To retrieve raw tweets from that source, run the following `SELECT` statement in the SensorBee shell:

```
twitter> SELECT RSTREAM * FROM public_tweets [RANGE 1 TUPLES];
```

Note: For simplicity, a relative path is specified as the `key_file` parameter. However, it is usually recommended to pass an absolute path when running the SensorBee server as a daemon.

Preprocessing Tweets and Extracting Features for Machine Learning

Before applying machine learning to tweets, they need to be converted into another form of information so that machine learning algorithms can utilize them. The conversion consists of two tasks: preprocessing and feature extraction. Preprocessing generally involves data cleansing, filtering, normalization, and so on. Feature extraction transforms preprocessed data into several pieces of information (i.e. features) that machine learning algorithms can "understand".

Which preprocessing or feature extraction methods are required for machine learning varies depending on the format or data type of input data or machine learning algorithms to be used. Therefore, this tutorial only shows one example of applying a classification algorithm to English tweets.

Selecting Meaningful Fields of English Tweets

Because this tutorial aims at English tweets, tweets written in other languages needs to be removed. This can be done with the `WHERE` clause, as you can check in the SensorBee shell:

```
twitter> SELECT RSTREAM * FROM public_tweets [RANGE 1 TUPLES]
    WHERE lang = "en";
```

Tweets have the `lang` field and it can be used for the filtering.

In addition to it, not all fields in a raw tweet will be required for machine learning. Thus, removing unnecessary fields keeps data simple and clean:

```
CREATE STREAM en_tweets AS
    SELECT RSTREAM
        "sensorbee.tweets" AS tag, id_str AS id, lang, text,
        user.screen_name AS screen_name, user.description AS description
    FROM public_tweets [RANGE 1 TUPLES]
    WHERE lang = "en";
```

This statement creates a new stream `en_tweets`. It only selects English tweets by `WHERE lang = "en"`. `"sensorbee.tweets" AS tag` is used by `fluentd` sink later. The items in that stream will look like:

```
{
    "tag": "sensorbee.tweets",
    "id": "the string representation of tweet's id",
    "lang": "en",
    "text": "the contents of the tweet",
    "screen_name": "user's @screen_name",
```

```
    "description": "user's profile description"
}
```

Note: `AS` in `user.screen_name AS screen_name` is required at the moment. Without it, the field would have the name like `col_n`. This is because `user.screen_name` could be evaluated as a JSON Path and might result in multiple return values so that it cannot properly be named. This specification might be going to be changed in the future version.

Removing Noise

Noise that is meaningless and could be harmful to machine learning algorithms needs to be removed. The field of natural language processing (NLP) has developed many methods for this purpose and they can be found in a wide variety of articles. However, this tutorial only applies some of the most basic operations on each tweets.

```
CREATE STREAM preprocessed_tweets AS
    SELECT RSTREAM
        filter_stop_words(
            nlp_split(
                nlp_to_lower(filter_punctuation_marks(text)),
            " ")) AS text_vector,
        filter_stop_words(
            nlp_split(
                nlp_to_lower(filter_punctuation_marks(description)),
            " ")) AS description_vector,
        *
    FROM en_tweets [RANGE 1 TUPLES];
```

The statement above creates a new stream `preprocessed_tweets` from `en_tweets`. It adds two fields to the tuple emitted from `en_tweets`: `text_vector` and `description_vector`. As for preprocessing, the statement applies following methods to `text` and `description` fields:

- Remove punctuation marks
- Change uppercase letters to lowercase
- Remove stopwords

First of all, punctuation marks are removed by the user-defined function (UDF) `filter_puncuation_marks`. It is provided in a plugin for this tutorial in the `github.com/sensorbee/tutorial/ml` package. The UDF removes some punctuation marks such as ",", ".", or "()" from a string.

Note: Emoticons such as ":)" may play a very important role in classification tasks like sentiment estimation. However, `filter_punctuation_marks` simply removes most of them for simplicity. Develop a better UDF to solve this issue as an exercise.

Second, all uppercase letters are converted into lowercase letters by the `nlp_to_lower` UDF. The UDF is registered in `github.com/sensorbee/nlp/plugin`. Because a letter is mere byte code and the values of "a" and "A" are different, machine learning algorithms consider "word" and "Word" have different meanings. To avoid that confusion, all letters should be "normalized".

Note: Of course, some words should be distinguished by explicitly starting with an uppercase. For example, "Mike" could be a name of a person, but changing it to "mike" could make the word vague.

Finally, all stopwords are removed. Stopwords are words that appear too often and don't provide any insight for classification. Stopword filtering in this tutorial is done in two steps: tokenization and filtering. To perform a dictionary-based stopword filtering, the content of a tweet needs to be tokenized. Tokenization is a process that converts a sentence into a sequence of words. In English, "I like sushi" will be tokenized as `["I", "like", "sushi"]`. Although tokenization isn't as simple as just splitting words by white spaces, the `preprocessed_tweets` stream simply does it for simplicity using the UDF `nlp_split`, which is defined in the `github.com/sensorbee/nlp` package. `nlp_split` takes two arguments: a sentence and a splitter. In the statement, contents are split by a white space. `nlp_split` returns an array of strings. Then, the UDF `filter_stop_words` takes the return value of `nlp_split` and removes stopwords contained in the array. `filter_stop_word` is provided as a part of this tutorial in the `github.com/sensorbee/tutorial/ml` package. It's a mere example UDF and doesn't provide perfect stopword filtering.

As a result, both `text_vector` and `description_vector` have an array of words like `["i", "want", "eat", "sushi"]` created from the sentence `I want to eat sushi.`.

Preprocessing shown so far is very similar to the preprocessing required for full-text search engines. There should be many valuable resources among that field including Elasticsearch.

Note: For other preprocessing approaches such as stemming, refer to natural language processing textbooks.

Creating Features

In NLP, a bag-of-words representation is usually used as a feature for machine learning algorithms. A bag-of-words consists of pairs of a word and its weight. Weight could be any numerical value and usually something related to term frequency (TF) is used. A sequence of the pairs is called a feature vector.

A feature vector can be expressed as an array of weights. Each word in all tweets observed by a machine learning algorithm corresponds to a particular position of the array. For example, the weight of the word "want" may be 4th element of the array.

A feature vector for NLP data could be very long because tweets contains many words. However, each vector would be sparse due to the maximum length of tweets. Even if machine learning algorithms observe more than 100,000 words and use them as features, each tweet only contains around 30 or 40 words. Therefore, each feature vector is very sparse, that is, only a small number its elements have non-zero weight. In such cases, a feature vector can effectively expressed as a map:

```
{
    "word": weight,
    "word": weight,
    ...
}
```

This tutorial uses online classification algorithms that are imported from Jubatus, a distributed online machine learning server. These algorithms accept the following form of data as a feature vector:

```
{
    "word1": 1,
    "key1": {
        "word2": 2,
        "word3": 1.5,
    },
    "word4": [1.1, 1.2, 1.3]
}
```

The SensorBee terminology for that kind of data structure is "map". A map can be nested and its value can be an array containing weights. The map above is converted to something like:

```
{
    "word1": 1,
    "key1/word2": 2,
    "key1/word3": 1.5,
    "word4[0]": 1.1,
    "word4[1]": 1.2,
    "word4[2]": 1.3
}
```

The actual feature vectors for the tutorial are created in the `fv_tweets` stream:

```
CREATE STREAM fv_tweets AS
  SELECT RSTREAM
    {
        "text": nlp_weight_tf(text_vector),
        "description": nlp_weight_tf(description_vector)
    } AS feature_vector,
    tag, id, screen_name, lang, text, description
FROM preprocessed_tweets [RANGE 1 TUPLES];
```

As described earler, `text_vector` and `description_vector` are arrays of words. The `nlp_weight_tf` function defined in the `github.com/sensorbee/nlp` package computes a feature vector from an array. The weight is term frequency (i.e. the number of occurrences of a word). The result is a map expressing a sparse vector above. To see how the `feature_vector` looks like, just issue a `SELECT` statement for the `fv_tweets` stream.

All required preprocessing and feature extraction have been completed and it's now ready to apply machine learning to tweets.

Applying Machine Learning

The `fv_tweets` stream now has all the information required by a machine learning algorithm to classify tweets. To apply the algorithm for each tweets, pre-trained machine learning models have to be loaded:

```
LOAD STATE age_model TYPE jubaclassifier_arow
    OR CREATE IF NOT SAVED
    WITH label_field = "age", regularization_weight = 0.001;
LOAD STATE gender_model TYPE jubaclassifier_arow
    OR CREATE IF NOT SAVED
    WITH label_field = "gender", regularization_weight = 0.001;
```

In SensorBee, machine learning models are expressed as user-defined states (UDSs). In the statement above, two models are loaded: `age_model` and `gender_model`. These models contain the necessary information to classify gender and age of the user of each tweet. The model files are located in the `uds` directory that was copied from the package's `config` directory beforehand:

```
/path/to/sbml$ ls uds
twitter-age_model-default.state
twitter-gender_model-default.state
```

These filenames were automatically assigned by SensorBee server when the `SAVE STATE` statement was issued. It will be described later.

Both models have the type `jubaclassifier_arow` imported from Jubatus. The UDS type is implemented in the github.com/sensorbee/jubatus/classifier package. `jubaclassifier_arow` implements the AROW online linear

classification algorithm *[Crammer09]*. Parameters specified in the `WITH` clause are related to training and will be described later.

After loading the models as UDSs, the machine learning algorithm is ready to work:

```
CREATE STREAM labeled_tweets AS
    SELECT RSTREAM
        juba_classified_label(jubaclassify("age_model", feature_vector)) AS age,
        juba_classified_label(jubaclassify("gender_model", feature_vector)) AS gender,
        tag, id, screen_name, lang, text, description
    FROM fv_tweets [RANGE 1 TUPLES];
```

The `labeled_tweets` stream emits tweets with `age` and `gender` labels. The `jubaclassify` UDF performs classification based on the given model.

```
twitter> EVAL jubaclassify("gender_model", {
    "text": {"i": 1, "wanna": 1, "eat":1, "sushi":1},
    "description": {"i": 1, "need": 1, "sushi": 1}
});
{"male":0.021088751032948494,"female":-0.020287269726395607}
```

`jubaclassify` returns a map of labels and their scores as shown above. The higher the score of a label, the more likely a tweet has the label. To choose the label having the highest score, the `juba_classified_label` function is used:

```
twitter> EVAL juba_classified_label({
    "male":0.021088751032948494,"female":-0.020287269726395607});
"male"
```

`jubaclassify` and `juba_classified_label` functions are also defined in the `github.com/sensorbee/jubatus/classifier` package.

Inserting Labeled Tweets Into Elasticsearch via Fluentd

Finally, tweets labeled by machine learning need to be inserted into Elasticsearch for visualization. This is done via fluentd which was previously set up.

```
CREATE SINK fluentd TYPE fluentd;
INSERT INTO fluentd from labeled_tweets;
```

SensorBee provides `fluentd` plugins in the `github.com/sensorbee/fluentd` package. The `fluentd` sink write tuples into fluentd's `forward` input plugin running on the same host.

After creating the sink, the `INSERT INTO` statement starts writing tuples from a source or a stream into it. This statement is the last one in the `twitter.bql` file and also concludes this section. All the steps from connecting to the Twitter API, transforming tweets and analyzing them using Jubatus have been shown in this section. As the last part of this tutorial, it will be shown how the training of the previously loaded model files has been done.

Training

The previous section used the machine learning models that were already trained but it was not described how to train them. This section explains how machine learning models can be trained with BQL and the `sensorbee` command.

Preparing Training Data

Because the machine learning algorithm used in this tutorial is supervised learning, it requires a training data set to create models. Training data is a pair of original data and its label. There is no common format of a training data set and a format can vary depending on use cases. In this tutorial, a training data set consists of multiple lines each of which has exactly one JSON object.

```
{"description":"I like sushi.", ...}
{"text":"I wanna eat sushi.", ...}
...
```

In addition, each JSON object needs to have two fields "age" and "gender":

```
{"age":"10-19","gender":"male", ...other original fields...}
{"age":"20-29","gender":"female", ...other original fields...}
...
```

In the pre-trained model, age and gender have following labels:

- age
  - `10-19`
  - `20-29`
  - `30-39`
  - `40-49`
  - `50<`
- gender
  - `male`
  - `female`

Both age and gender can have additional labels if necessary. Labels can be empty if they are not known for sure. After annotating each tweet, the training data set needs to be saved as `training_tweets.json` in the `/path/to/sbml` directory.

The training data set used for the pre-trained models contains 4974 gender labels and 14747 age labels.

Training

Once the training data set has been prepared, the models can be trained with the following command:

```
/path/to/sbml$ ./sensorbee runfile -t twitter -c sensorbee.yaml -s '' train.bql
```

`sensorbee runfile` executes BQL statements written in a given file, e.g. `train.bql` in the command above. `-t twitter` means the name of the topology is `twitter`. The name is used for the filenames of saved models later. `-c sensorbee.yaml` passes the same configuration file as the one used previously. `-s ''` means `sensorbee runfile` saves all UDSs after the topology stops.

After running the command above, two models (UDSs) are saved in the `uds` directory. The saved model can be loaded by the `LOAD STATE` statement.

BQL Statements

All BQL statements for training are written in `train.bql`. Most statements in the file overlap with `twitter.bql`, so only differences will be explained.

```
CREATE STATE age_model TYPE jubaclassifier_arow
    WITH label_field = "age", regularization_weight = 0.001;
CREATE SINK age_model_trainer TYPE uds WITH name = "age_model";

CREATE STATE gender_model TYPE jubaclassifier_arow
    WITH label_field = "gender", regularization_weight = 0.001;
CREATE SINK gender_model_trainer TYPE uds WITH name = "gender_model";
```

These statements create UDSs for machine learning models of age and gender classifications. `CREATE STATE` statements are same as ones in `twitter.bql`. The `CREATE SINK` statements above create new sinks with the type `uds`. The `uds` sink writes tuples into the UDS specified as `name` if the UDS supports it. `jubaclassifier_arow` supports writing tuples. When a tuple is written to it, it trains the model with the tuple having training data. It assumes that the tuple has two fields: a feature vector field and a label field. By default, a feature vector and a label are obtained by the `feature_vector` field and the `label` field in a tuple, respectively. In this tutorial, each tuple has two labels: `age` and `gender`. Therefore, the field names of those fields need to be customized. The field names can be specified by the `label_field` parameter in the `WITH` clause of the `CREATE STATE` statement. In the statements above, `age_model` and `gender_model` UDSs obtain labels from the `age` field and the `gender` field, respectively.

```
CREATE PAUSED SOURCE training_data TYPE file WITH path = "training_tweets.json";
```

This statement creates a source which inputs tuples from a file. `training_tweets.json` is the file prepared previously and contains training data. The source is created with the `PAUSED` flag, so it doesn't emit any tuple until all other components in the topology are set up and the `RESUME SOURCE` statement is issued.

`en_tweets`, `preprocessed_tweets`, and `fv_tweets` streams are same as ones in `twitter.bql` except that the tweets are emitted from the `file` source rather than the `twitter_public_stream` source.

```
CREATE STREAM age_labeled_tweets AS
    SELECT RSTREAM * FROM fv_tweets [RANGE 1 TUPLES] WHERE age != "";
CREATE STREAM gender_labeled_tweets AS
    SELECT RSTREAM * FROM fv_tweets [RANGE 1 TUPLES] WHERE gender != "";
```

These statements create new sources that only emit tuples having a label for training.

```
INSERT INTO age_model_trainer FROM age_labeled_tweets;
INSERT INTO gender_model_trainer FROM gender_labeled_tweets;
```

Then, those filtered tuples are written into models (UDSs) via the `uds` sinks created earlier.

```
RESUME SOURCE training_data;
```

All streams are set up and the `training_data` source is finally resumed. With the `sensorbee runfile` command, all statements run until all tuples emitted from the `training_data` source are processed.

When BQL statements are run on the server, the `SAVE STATE` statement is usually used to save UDSs. However, `sensorbee runfile` optionally saves UDSs after the topology is stopped. Therefore, `train.bql` doesn't issue `SAVE STATE` statements.

Evaluation

Evaluation tools are being developed.

Online Training

All machine learning algorithms provided by Jubatus are online algorithms, that is, models can incrementally be trained every time a new training data is given. In contrast to online algorithms, batch algorithms requires all training data for each training. Since online machine learning algorithms don't have to store training data locally, they can train models from streaming data.

If training data can be obtained by simple rules, training and classification can be applied to streaming data concurrently in the same SensorBee server. In other words, a UDS can be used for training and classification.

Part III

The BQL Language

This part describes the use of the BQL Language in SensorBee. It starts with describing the general syntax of BQL, then explain how to create the structures for data in-/output and stateful operations. After that, the general processing model and the remaining BQL query types are explained. Finally, a list of operators and functions that can be used in BQL expressions is provided.

CHAPTER 6

BQL Syntax

Lexical Structure

BQL has been designed to be easy to learn for people who have used SQL before. While keywords and commands differ in many cases, the basic structure, set of tokens, operators etc. is the same. For example, the following is (syntactically) valid BQL input:

```
SELECT RSTREAM given_name, last_name FROM persons [RANGE 1 TUPLES] WHERE age > 20;

CREATE SOURCE s TYPE fluentd WITH host="example.com", port=12345;

INSERT INTO file FROM data;
```

This is a sequence of three commands, one per line (although this is not required; more than one command can be on a line, and commands can span multiple lines where required). Additionally, comments can occur in BQL input. They are effectively equivalent to whitespace.

The type of commands that can be used in BQL is described in *Input/Output/State Definition* and *Queries*.

Identifiers and Keywords

Tokens such as `SELECT`, `CREATE`, or `INTO` in the example above are examples of *keywords*, that is, words that have a fixed meaning in the BQL language. The tokens `persons` and `file` are examples of identifiers. They identify names of streams, sources, or other objects, depending on the command they are used in. Therefore they are sometimes simply called "names". Keywords and identifiers have the same lexical structure, meaning that one cannot know whether a token is an identifier or a keyword without knowing the language.

BQL identifiers and keywords must begin with a letter (`a-z`). Subsequent characters can be letters, underscores, or digits (`0-9`). Keywords and unquoted identifiers are in general case insensitive.

However, there is one important difference between SQL and BQL when it comes to "column identifiers". In BQL, there are no "columns" with names that the user can pick herself, but "field selectors" that describe the path to a value in a JSON-like document imported from outside the system. Therefore field selectors are case-sensitive (in order to

be able to deal with input of the form `{"a": 1, "A": 2}`) and also there is a form that allows to use special characters; see *Field Selectors* for details.

Note: There is a list of reserved words that cannot be used as identifiers to avoid confusion. This list can be found at https://github.com/sensorbee/sensorbee/blob/master/core/reservedwords.go. However, this restriction does not apply to field selectors.

Constants

There are multiple kinds of implicitly-typed constants in BQL: strings, decimal numbers (with and without fractional part) and booleans. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. These alternatives are discussed in the following subsections.

String Constants

A string constant in BQL is an arbitrary sequence of characters bounded by double quotes (`"`), for example `"This is a string"`. To include a double-quote character within a string constant, write two adjacent double quotes, e.g., `"Dianne""s horse"`.

No escaping for special characters is supported at the moment, but any valid UTF-8 encoded byte sequence can be used. See *the string data type reference* for details.

Numeric Constants

There are two different numeric data types in BQL, `int` and `float`, representing decimal numbers without and with fractional part, respectively.

An `int` constant is written as

```
[-]digits
```

A `float` constant is written as

```
[-]digits.digits
```

Scientific notation (`1e+10`) as well as Infinity and NaN cannot be used in BQL statements.

Some example of valid numerical constants:

```
42
3.5
-36
```

See the type references for *int* and *float* for details.

Note: For some operations/functions it makes a difference whether `int` or `float` is used (e.g., `2/3` is `0`, but `2.0/3` is `0.666666`). Be aware of that when writing constants in BQL statements.

Boolean Constants

There are two keywords for the two possible boolean values, namely `true` and `false`.

See *the bool data type reference* for details.

Operators

An operator is a sequence of the items from the following list:

```
+
-
*
/
<
>
=
!
%
```

See the *chapter on Operators* for the complete list of operators in BQL. There are no user-defined operators at the moment.

Special Characters

Some characters that are not alphanumeric have a special meaning that is different from being an operator. Details on the usage can be found at the location where the respective syntax element is described. This section only exists to advise the existence and summarize the purposes of these characters.

- Parentheses (`()`) have their usual meaning to group expressions and enforce precedence. In some cases parentheses are required as part of the fixed syntax of a particular BQL command.
- Brackets (`[]`) are used in *Array Constructors* and in *Field Selectors*, as well as in *Stream-to-Relation Operators*.
- Curly brackets (`{}`) are used in *Map Constructors*
- Commas (`,`) are used in some syntactical constructs to separate the elements of a list.
- The semicolon (`;`) terminates a BQL command. It cannot appear anywhere within a command, except within a string constant or quoted identifier.
- The colon (`:`) is used to separate stream names and field selectors, and within field selectors to select array slices (see *Extended Descend Operators*).
- The asterisk (`*`) is used in some contexts to denote all the fields of a table row (see *Notes on Wildcards*). It also has a special meaning when used as the argument of an aggregate function, namely that the aggregate does not require any explicit parameter.
- The period (`.`) is used in numeric constants and to denote descend in field selectors.

Comments

A comment is a sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

```
-- This is a standard BQL comment
```

C-style (multi-line) comments cannot be used.

Operator Precedence

The following table shows the operator precedence in BQL:

Operator/Element	Description
`::`	typecast
`-`	unary minus
`* / %`	multiplication, division, modulo
`+ -`	addition, subtraction
`IS`	`IS NULL` etc.
(any other operator)	e.g., `||`
`= != <> <= < >= >`	comparison operator
`NOT`	logical negation
`AND`	logical conjunction
`OR`	logical disjunction

Value Expressions

Value expressions are used in a variety of contexts, such as in the target list or filter condition of the `SELECT` command. The expression syntax allows the calculation of values from primitive parts using arithmetic, logical, set, and other operations.

A value expression is one of the following:

- A constant or literal value
- A field selector
- A row metadata reference
- An operator invocation
- A function call
- An aggregate expression
- A type cast
- An array constructor
- A map constructor
- Another value expression in parentheses (used to group subexpressions and override precedence)

The first option was already discussed in *Constants*. The following sections discuss the remaining options.

Field Selectors

In SQL, each table has a well-defined schema with columns, column names and column types. Therefore, a column name is enough to check whether that column exists, what type it has and if the type that will be extracted matches the type expected by the surrounding expression.

In BQL, each row corresponds to a JSON-like object, i.e., a map with string keys and values that have one of several data types (see *Data Types and Conversions*). In particular, nested maps and arrays are commonplace in the data streams used with BQL. For example, a row could look like:

```
{"ids": [3, 17, 21, 5],
 "dists": [
  {"other": "foo", "value": 7},
  {"other": "bar", "value": 3.5}
 ],
 "found": true}
```

To deal with such nested data structures, BQL uses a subset of JSON Path to address values in a row.

Basic Descend Operators

In general, a JSON Path describes a path to a certain element of a JSON document. Such a document is looked at as a rooted tree and each element of the JSON Path describes how to descend from the current node to a node one level deeper, with the start node being the root. The basic rules are:

- If the current node is a map, then

  ```
  .child_key
  ```

  or

  ```
  ["child_key"]
  ```

  mean "descend to the child node with the key `child_key`". The second form must be used if the key name has a non-identifier shape (e.g., contains spaces, dots, brackets or similar). It is an error if the current node is not a map. It is an error if the current node does not have such a child node.

- If the current node is an array, then

  ```
  [k]
  ```

  means "descend to the (zero-based) k-th element in the array". Negative indices count from the end end of the array (as in Python). It is an error if the current node is not an array. It is an error if the given index is out of bounds.

The first element of a JSON Path must always be a "map access" component (since the document is always a map) and the leading dot must be omitted.

For example, `ids[1]` in the document given above would return `17`, `dists[-2].other` would return `foo` and just `dists` would return the array `[{"other": "foo", "value": 7}, {"other": "bar", "value": 3.5}]`.

Extended Descend Operators

There is limited support for array slicing and recursive descend:

- If the current node is a map or an array, then

  ```
  ..child_key
  ```

  returns an array of all values below the current node that have the key `child_key`. However, once a node with key `child_key` has been found, it will be returned as is, even if it may possibly itself contain that key again.

  This selector cannot be used as the first component of a JSON Path. It is an error if the current node is not a map or an array. It is *not* an error if there is no child element with the given key.

- If the current node is an array, then

```
[start:end]
```

returns an array of all values with the indexes in the range $[\text{start}, \text{end} - 1]$. One or both of `start` and `end` can be omitted, meaning "from the first element" and "until the last element", respectively.

```
[start:end:step]
```

returns an array of all elements with the indexes $[\text{start}, \text{start}+\text{step}, \text{start}+2\cdot\text{step}, \cdots, \text{end}-1]$ if `step` is positive, or $[\text{start}, \text{start} - \text{step}, \text{start} - 2 \cdot \text{step}, \cdots, \text{end} + 1]$ if it is negative. (This description is only true for positive indices, but in fact also negative indices can be used, again counting from the end of the array.) In general, the behavior has been implemented to be very close to Python's list slicing.

These selectors cannot be used as the first component of a JSON Path. It is an error if it can be decided independent of the input data that the specified values do not make sense (e.g., `step` is 0, or `end` is larger than `start` but `step` is negative), but slices that will always be empty (e.g., `[2:2]`) are valid. Also, if it depends on the input data whether a slice specification is valid or not (e.g., `[4:-4]`) it is not an error, but an empty array is returned.

- If the slicing or recursive descend operators are followed by ordinary JSON Path operators as described before, their meaning changes to "... for every element in the array". For example, `list[1:3].foo` has the same result as `[list[1].foo, list[2].foo, list[3].foo]` (except that the latter would fail if `list` is not long enough) or a Python list comprehension such as `[x.foo for x in list[1:3]]`. However, it is not possible to chain multiple list-returning operators: `list[1:3]..foo` or `foo..bar..hoge` are invalid.

Examples

Given the input data

```
{
    "foo": [
        {"hoge": [
            {"a": 1, "b": 2},
            {"a": 3, "b": 4} ],
         "bar": 5},
        {"hoge": [
            {"a": 5, "b": 6},
            {"a": 7, "b": 8} ],
         "bar": 2},
        {"hoge": [
            {"a": 9, "b": 10} ],
         "bar": 8}
    ],
    "nantoka": {"x": "y"}
}
```

the following table is supposed to illustrate the effect of various JSON Path expressions.

Path	Result
`nantoka`	`{"x": "y"}`
`nantoka.x`	`"y"`
`nantoka["x"]`	`"y"`
`foo[0].bar`	`5`
`foo[0].hoge[-1].a`	`3`
`["foo"][0]["hoge"][-1]["a"]`	`3`
`foo[1:2].bar`	`[2, 8]`
`foo..bar`	`[5, 2, 8]`
`foo..hoge[0].b`	`[2, 6, 10]`

Row Metadata References

Metadata is the data that is attached to a tuple but which cannot be accessed as part of the normal row data.

Tuple Timestamp

At the moment, the only metadata that can be accessed from within BQL is a tuple's system timestamp (the time that was set by the source that created it). This timestamp can be accessed using the `ts()` function. If multiple streams are joined, a stream prefix is required to identify the input tuple that is referred to, i.e.,

```
stream_name:ts()
```

Operator Invocations

There are three possible syntaxes for an operator invocation:

```
expression  operator  expression

operator  expression

expression  operator
```

See the section *Operators* for details.

Function Calls

The syntax for a function call is the name of a function, followed by its argument list enclosed in parentheses:

```
function_name([expression [, expression ... ]])
```

For example, the following computes the square root of 2:

```
sqrt(2);
```

The list of built-in functions is described in section *Functions*.

Aggregate Expressions

An aggregate expression represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is the following:

```
function_name(expression [, ... ] [ order_by_clause ])
```

where `function_name` is a previously defined aggregate and expression is any value expression that does not itself contain an aggregate expression. The optional `order_by_clause` is described below.

In BQL, aggregate functions can take aggregate and non-aggregate parameters. For example, the `string_agg` function can be called like

```
string_agg(name, ", ")
```

to return a comma-separated list of all names in the respective group. However, the second parameter is not an aggregation parameter, so for a statement like

```
SELECT RSTREAM string_agg(name, sep) FROM ...
```

`sep` must be mentioned in the `GROUP BY` clause.

For many aggregate functions (e.g., `sum` or `avg`), the order of items in the group does not matter. However, for other functions (e.g., `string_agg`) the user has certain expectations with respect to the order that items should be fed into the aggregate function. In this case, the `order_by_clause` with the syntax

```
ORDER BY expression [ASC | DESC] [ , expression [ASC | DESC] ... ]
```

can be used. The rows that are fed into the aggregate function are sorted by the values of the given expression in ascending (default) or descending mode. For example,

```
string_agg(first_name || " " || last_name, "," ORDER BY last_name)
```

will create a comma-separated list of names, ordered ascending by the last name.

See *Aggregate Functions* for a list of built-in aggregate functions.

Type Casts

A type cast specifies a conversion from one data type to another. BQL accepts two equivalent syntaxes for type casts:

```
CAST(expression AS type)
expression::type
```

When a cast is applied to a value expression, it represents a run-time type conversion. The cast will succeed only if a suitable type conversion operation has been defined, see *Conversions*.

Array Constructors

An array constructor is an expression that builds an array value using values for its member elements. A simple array constructor consists of a left square bracket `[`, a list of expressions (separated by commas) for the array element values, and finally a right square bracket `]`. For example:

```
SELECT RSTREAM [7, 2 * stream:a, true, "blue"] FROM ...
```

Each element of the array can have a different type. In particular, the wildcard is also allowed as an expression and will include the whole current row (i.e., a map) as an array element.

Note: Single-element arrays of strings could also be interpreted as JSON Paths and are therefore required to have a trailing comma after their only element: `["foo",]`

Map Constructors

A map constructor is an expression that builds a map value using string keys and arbitrary values for its member elements. A simple map constructor consists of a left curly bracket `{`, a list of `"key": value` pairs (separated by commas) for the map elements, and finally a right curly bracket `}`. For example:

```
SELECT RSTREAM {"a_const": 7, "prod": 2 * stream:a} FROM ...
```

The keys must be string literals (i.e., they can not be computed expressions); in particular they must be written using double quotes. The values can be arbitrary expressions, including a wildcard.

Expression Evaluation Rules

The order of evaluation of subexpressions is not defined. In particular, the inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order.

Furthermore, if the result of an expression can be determined by evaluating only some parts of it, then other subexpressions might not be evaluated at all. For instance, if one wrote:

```
true OR somefunc()
```

then `somefunc()` would (probably) not be called at all. The same would be the case if one wrote:

```
somefunc() OR true
```

Note that this is *not* the same as the left-to-right "short-circuiting" of Boolean operators that is found in some programming languages.

Calling Functions

BQL allows functions to be called using only the positional notation. In positional notation, a function call is written with its argument values in the same order as they are defined in the function declaration. Therefore, while some parameters of a function can be optional, these parameters can only be omitted *at the end* of the parameter list.

For example,

```
log(100)
log(100, 2)
```

are both valid function calls computing the logarithm of a function. The first one uses the default value 10 for the logarithm base, the second one uses the given value 2.

CHAPTER 7

Input/Output/State Definition

To process streams of data, that data needs to be imported into SensorBee and the processing results have to be exported from it. This chapter introduces input and output components in BQL. It also describes how BQL supports stateful data processing using user-defined states (UDSs).

Data Input

BQL inputs a stream of data using a **source**. A source receives data defined and generated outside SensorBee, converts that data into tuples, and finally emits tuples for further processing. This section describes how a source can be created, operated, and dropped.

Creating a Source

A source can be created using the *CREATE SOURCE* statement.

```
CREATE SOURCE logs TYPE file WITH path = "access_log.jsonl";
```

In this example, a source named `logs` is created and it has the type `file`. The `file` source type has a required parameter called `path`. The parameter is specified in the `WITH` clause. Once a source is created, other components described later can read tuples from the source and compute results based on them.

When multiple parameters are required, they should be separated by commas:

```
CREATE SOURCE src TYPE some_type
    WITH param1 = val1, param2 = val2, param3 = val3;
```

Each source type has its own parameters and there is no parameter that is common to all source types.

Source types can be registered to the SensorBee server as plugins. To learn how to develop and register a source plugin, see *Source Plugins*.

Built-in Sources

BQL has a number of built-in source types.

`file`

The `file` type provides a source that inputs tuples from an existing file.

`node_statuses`

The `node_statuses` source periodically emits tuples with information about nodes in a topology. The status includes connected nodes, number of tuples emitted from or written to the node, and so on.

`edge_statuses`

The `edge_statuses` source periodically emits tuples with information about each edge (a.k.a. pipe) that connects a pair of nodes. Although this information is contained in tuples emitted from `node_statuses` source, the `edge_statuses` source provides more edge-centric view of IO statuses.

`dropped_tuples`

The `dropped_tuples` emits tuples dropped from a topology. It only reports once per tuple. Tuples are often dropped from a topology because a source or a stream is not connected to any other node or a `SELECT` statement tries to look up a nonexistent field of a tuple.

Pausing and Resuming a Source

By default, a source starts emitting tuples as soon as it is created. By adding the `PAUSED` keyword to the `CREATE SOURCE` statement, it creates a source that is paused on startup:

```
CREATE PAUSED SOURCE logs TYPE file WITH path = "access_log.jsonl";
```

The *RESUME SOURCE* statement makes a paused source emit tuples again:

```
RESUME SOURCE logs;
```

The statement takes the name of the source to be resumed.

A source can be paused after it is created by the *PAUSE SOURCE* statement:

```
PAUSE SOURCE logs;
```

The statement also takes the name of the source to be paused.

Not all sources support `PAUSE SOURCE` and `RESUME SOURCE` statements. Issuing statements to those sources results in an error.

Rewinding a Source

Some sources can be rewound, that is, they emit tuples again starting from the beginning. The *REWIND SOURCE* statement rewinds a source if the source supports the statement:

```
REWIND SOURCE logs;
```

The statement takes the name of the source to be rewound. Issuing the statement to sources that don't support rewinding results in an error.

Dropping a Source

The *DROP SOURCE* statement drops (i.e. removes) a source from a topology:

```
DROP SOURCE logs;
```

The statement takes the name of the source to be dropped. Other nodes in a topology cannot refer to the source once it's dropped. Also, nodes connected to a source may be stopped cascadingly when the source gets dropped.

Data Output

Results of tuple processing need to be emitted to systems or services running outside the SensorBee server in order to work with them as part of a larger system. A **sink** receives the results of computations performed within the SensorBee server and sends them to the outside world. This section explains how sinks are operated in BQL.

Creating a Sink

A sink can be created by the *CREATE SINK* statement:

```
CREATE SINK filtered_logs TYPE file WITH path = "filtered_access_log.jsonl";
```

The statement is very similar to the `CREATE SOURCE` statement. It takes the name of the new sink, its type, and parameters. Multiple parameters can also be provided as a list separated by commas. Each sink type has its own parameters and there is no parameter that is common to all sink types.

Sink types can also be registered to the SensorBee server as plugins. To learn how to develop and register a sink plugin, see *Sink Plugins*.

Built-in Sinks

BQL has a number of built-in sink types.

`file`

The `file` type provides a sink that writes tuples to a file.

`stdout`

A `stdout` sink writes output tuples to stdout.

`uds`

A `uds` sink passes tuples to user-defined states, which is described later.

Writing Data to a Sink

The *INSERT INTO* statement writes data to a sink:

```
INSERT INTO filtered_logs FROM filtering_stream;
```

The statement takes the name of sink to be written and the name of a source or a stream, which will be described in following chapters.

Dropping a Sink

The *DROP SINK* statement drops a sink from a topology:

```
DROP SINK filtered_logs;
```

The statement takes the name of the sink to be dropped. The sink cannot be accessed once it gets dropped. All `INSERT INTO` statements writing to the dropped sink are also stopped.

Stateful Data Processing

SensorBee supports user-defined states (UDSs) to perform stateful streaming data processing. Such processing includes not only aggregates such as counting but also machine learning, adaptive sampling, and so on. In natural language processing, dictionaries or configurations for tokenizers can also be considered as states.

This section describes operations involving UDSs. Use cases of UDSs are described in the *tutorials* and how to develop a custom UDS is explained in the *server programming* part.

Creating a UDS

A UDS can be created using the *CREATE STATE* statement:

```
CREATE STATE age_classifier TYPE jubaclassifier_arow
    WITH label_field = "age", regularization_weight = 0.001;
```

This statement creates a UDS named `age_classifier` with the type `jubaclassifier_arow`. It has two parameters: `label_field` and `regularization_weight`. Each UDS type has its own parameters and there is no parameter that is common to all UDS types.

A UDS is usually used via user-defined functions (UDFs) that know about the internals of a specific UDS type. See *server programming* part for details.

Saving a State

The *SAVE STATE* statement persists a UDS:

```
SAVE STATE age_classifier;
```

The statement takes the name of the UDS to be saved. After the statement is issued, SensorBee saves the state based on the given configuration. The location and the format of saved data depend on the run-time configuration and are unknown to users.

The `SAVE STATE` statement may take a `TAG` to support versioning of the saved data:

```
SAVE STATE age_classifier TAG initial;
-- or
SAVE STATE age_classifier TAG trained;
```

When the `TAG` clause is omitted, `default` will be the default tag name.

Loading a State

The *LOAD STATE* loads a UDS that was previously saved with the `SAVE STATE` statement:

```
LOAD STATE age_classifier TYPE jubaclassifier_arow;
```

The statement takes the name of the UDS to be loaded and its type name.

The `LOAD STATE` statements may also take a `TAG`:

```
LOAD STATE age_classifier TYPE jubaclassifier_arow TAG initial;
-- or
LOAD STATE age_classifier TYPE jubaclassifier_arow TAG trained;
```

The UDS needs to have been saved with the specified tag before. When the `TAG` clause is omitted, it's same as:

```
LOAD STATE age_classifier TYPE jubaclassifier_arow TAG default;
```

The `LOAD STATE` statement fails if no saved state with the given name and type exists. In that case, to avoid failure and instead create a new "empty" instance, the `OR CREATE IF NOT SAVED` clause can be added:

```
LOAD STATE age_classifier TYPE jubaclassifier_arow
    OR CREATE IF NOT SAVED
        WITH label_field = "age", regularization_weight = 0.001;
```

If there is a saved state, this statement will load it, otherwise create a new state with the given parameters. This variant, too, can be used with the `TAG` clause:

```
LOAD STATE age_classifier TYPE jubaclassifier_arow TAG trained
    OR CREATE IF NOT SAVED
        WITH label_field = "age", regularization_weight = 0.001;
```

Dropping a State

The *DROP STATE* statement drops a UDS from a topology:

```
DROP STATE age_classifier;
```

The statement takes the name of the UDS to be dropped. Once a UDS is dropped, it can no longer be referred to by any statement unless it is cached somewhere.

CHAPTER 8

Queries

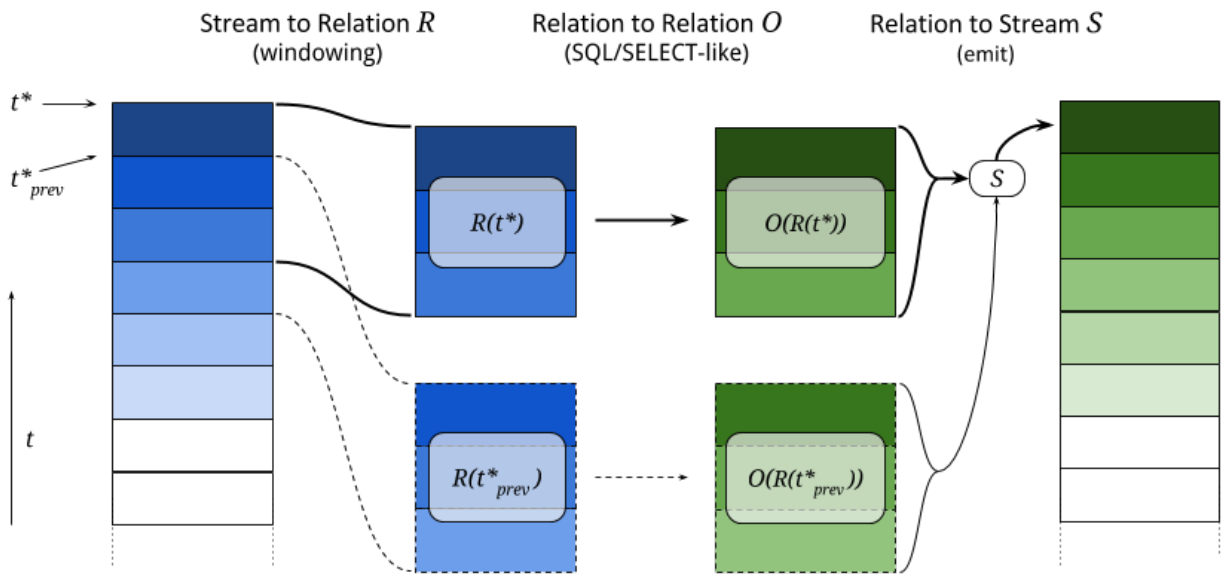
The previous chapters described how to define data sources and sinks to communicate with the outside world. Now it is discussed how to transform the data stream from those sources and write it to the defined sinks - that is, how to actually process data.

Processing Model

Overview

The processing model in BQL is similar to what is explained in *[cql]*. In this model, each tuple in a stream has the shape (t, d), where t is the original timestamp and d the data contained.

In order to execute SQL-like queries, a finite set of tuples from the possibly unbounded stream, a *relation*, is required. In the processing step at time t^*, a *stream-to-relation* operator R that converts a certain set of tuples in the stream to a relation $R(t^*)$ is used. This relation is then processed with a *relation-to-relation* operator O that is expressed in a form very closely related to an SQL `SELECT` statement. Finally, a *relation-to-stream* operator S will emit certain rows from the output relation $O(R(t^*))$ into the output stream, possibly taking into account the results of the previous execution step $O(R(t^*_{\text{prev}}))$. This process is illustrated in the following figure:



This three-step pipeline is executed for each tuple, but only for one tuple at a time. Therefore, during execution there is a well-defined "current tuple". This also means that if there is no tuple in the input stream for a long time, transformation functions will not be called.

Now the kind of stream-to-relation and relation-to-stream operators that can be used in BQL is explained.

Stream-to-Relation Operators

In BQL, there are two different stream-to-relation operators, a time-based one and a tuple-based one. They are also called "window operators", since they define a sliding window on the input stream. In terms of BQL syntax, the window operator is given after a stream name in the `FROM` clause within brackets and using the `RANGE` keyword, for example:

```
... FROM events [RANGE 5 SECONDS] ...
... FROM data [RANGE 10 TUPLES] ...
... FROM left [RANGE 2 SECONDS], right [RANGE 5 TUPLES] ...
```

From an SQL point of view, it makes sense to think of `stream [RANGE window-spec]` as the table to operate on.

The **time-based operator** is used with a certain time span I (such as 60 seconds) and at point in time t^* uses all tuples in the range $[t^* - I, t^*]$ to create the relation $R(t^*)$.

Valid time spans are positive integer or float values, followed by the `SECONDS` or `MILLISECONDS` keyword, for example `[RANGE 3.5 SECONDS]` or `[RANGE 200 MILLISECONDS]` are valid specifications. The maximal allowed values are 86,400 for `SECONDS` and 86,400,000 for `MILLISECONDS`, i.e., the maximal window size is one day.

Note:

- The point in time t^* is *not* the "current time" (however that would be defined), but it is equal to the timestamp of the current tuple. This approach means that a stream can be reprocessed with identical results independent of the system clock of some server. Also it is not necessary to worry about a delay until a tuple arrives in the system and is processed there.

- It is assumed that the tuples in the input stream arrive in the order of their timestamps. If timestamps are out of order, the window contents are not well-defined.
- The sizes of relations $R(t_1^*)$ and $R(t_2^*)$ can be different, since there may be more or less tuples in the given time span. However, there is always at least one tuple in the relation (the current one).

The **tuple-based operator** is used with a number k and uses the last k tuples that have arrived (or *all* tuples that have arrived when this number is less than k) to create the relation $R(t^*)$. The example figure above shows a tuple-based window with $k = 3$.

Valid ranges are positive integral values, followed by the `TUPLES` keyword, for example `[RANGE 10 TUPLES]` is a valid specification. The maximal allowed value is 1,048,575.

Note:

- The timestamps of tuples do not have any effect with this operator, they can also be out of order. Only the order in which the tuples arrived is important. (Note that for highly concurrent systems, "order" is not always a well-defined term.)
- At the beginning of stream processing, when less than k tuples have arrived, the size of the relation will be less than k.[1] As soon as k tuples have arrived, the relation size will be constant.

Relation-to-Stream Operators

Once a resulting relation $O(R(t^*))$ is computed, tuples in the relation need to be output as a stream again. In BQL, there are three relation-to-stream operators, `RSTREAM`, `ISTREAM` and `DSTREAM`. They are also called "emit operators", since they control how tuples are emitted into the output stream. In terms of BQL syntax, the emit operator keyword is given after the `SELECT` keyword, for example:

```
SELECT ISTREAM uid, msg FROM ...
```

The following subsections describe how each operator works. To illustrate the effects of each operator, a visual example is provided afterwards.

`RSTREAM` Operator

When `RSTREAM` is specified, all tuples in the relation are emitted. In particular, a combination of `RSTREAM` with a `RANGE 1 TUPLES` window operator leads to 1:1 input/output behavior and can be processed by a faster execution plan than general statements.

In contrast,

```
SELECT RSTREAM * FROM src [RANGE 100 TUPLES];
```

emits (at most) 100 tuples for every tuple in `src`.

`ISTREAM` Operator

When `ISTREAM` is specified, all tuples in the relation *that have not been in the previous relation* are emitted. (The "I" in `ISTREAM` stands for "insert".) Here, "previous" refers to the relation that was computed for the tuple just before

[1] Sometimes this leads to unexpected effects or complicated workarounds, while the cases where this is a useful behavior may be few. Therefore this behavior may change in future version.

the current tuple. Therefore the current relation can contain at most one row that was not in the previous relation and thus `ISTREAM` can emit at most one row in each run.

In section 4.3.2 of *[streamsql]*, it is highlighted that for the "is contained in previous relation" check, a notion of equality is required; in particular there are various possibilities how to deal with multiple tuples that have the same value. In BQL tuples with the same value are considered equal, so that if the previous relation contains the values $\{a, b\}$ and the current relation contains the values $\{b, a\}$, then nothing is emitted. However, multiplicities are respected, so that if the previous relation contains the values $\{b, a, b, a\}$ and the current relation contains $\{a, b, a, a\}$, then one a is emitted.

As an example for a typical use case,

```
SELECT ISTREAM * FROM src [RANGE 1 TUPLES];
```

will drop subsequent duplicates, i.e., emit only the first occurrence of a series of tuples with identical values.

To illustrate the multiplicity counting,

```
SELECT ISTREAM 1 FROM src [RANGE 3 TUPLES];
```

will emit three times 1 and then nothing (because after the first three tuples processed, both the previous and the current relation always look like $\{1, 1, 1\}$.)

`DSTREAM` Operator

The `DSTREAM` operator is very similar to `ISTREAM`, except that it emits all tuples in the *previous* relation that are not also contained in the current relation. (The "D" in `DSTREAM` stands for "delete".) Just as `ISTREAM`, equality is computed using value comparison and multiplicity counting is used: If the previous relation contains the values $\{a, a, b, a\}$ and the current relation contains $\{b, b, a, a\}$, then one a is emitted.

As an example for a typical use case,

```
SELECT DSTREAM * FROM src [RANGE 1 TUPLES];
```

will emit only the last occurrence of a series of tuples with identical values.

To illustrate the multiplicity counting,

```
SELECT DSTREAM 1 FROM src [RANGE 3 TUPLES];
```

will never emit anything.

Examples

To illustrate the difference between the three emit operators, a concrete example shall be presented. Consider the following statement (where `*STREAM` is a placeholder for one of the emit operators):

```
SELECT *STREAM id, price FROM stream [RANGE 3 TUPLES] WHERE price < 8;
```

This statement just takes the `id` and `price` key-value pairs of every tuple and outputs them untransformed.

In the following table, the leftmost column shows the data of the tuple in the stream, next to that is the contents of the current window $R(t^*)$, then the results of the relation-to-relation operator $O(R(t^*))$. In the table below, there is the list of items that would be output by the respective emit operator.

Internal Transformations

Current Tuple's Data	Current Window $R(t^*)$ (last three tuples)	Output Relation $O(R(t^*))$
{"id": 1, "price": 3.5}	{"id": 1, "price": 3.5}	{"id": 1, "price": 3.5}
{"id": 2, "price": 4.5}	{"id": 1, "price": 3.5} {"id": 2, "price": 4.5}	{"id": 1, "price": 3.5} {"id": 2, "price": 4.5}
{"id": 3, "price": 10.5}	{"id": 1, "price": 3.5} {"id": 2, "price": 4.5} {"id": 3, "price": 10.5}	{"id": 1, "price": 3.5} {"id": 2, "price": 4.5}
{"id": 4, "price": 8.5}	{"id": 2, "price": 4.5} {"id": 3, "price": 10.5} {"id": 4, "price": 8.5}	{"id": 2, "price": 4.5}
{"id": 5, "price": 6.5}	{"id": 3, "price": 10.5} {"id": 4, "price": 8.5} {"id": 5, "price": 6.5}	{"id": 5, "price": 6.5}

Emitted Tuple Data

RSTREAM	ISTREAM	DSTREAM
{"id": 1, "price": 3.5}	{"id": 1, "price": 3.5}	
{"id": 1, "price": 3.5} {"id": 2, "price": 4.5}	{"id": 2, "price": 4.5}	
{"id": 1, "price": 3.5} {"id": 2, "price": 4.5}		
{"id": 2, "price": 4.5}		{"id": 1, "price": 3.5}
{"id": 5, "price": 6.5}	{"id": 5, "price": 6.5}	{"id": 2, "price": 4.5}

Selecting and Transforming Data

In the previous section, it was explained how BQL converts stream data into relations and back. This section is about how this relational data can be selected and transformed. This functionality is exactly what SQL's `SELECT` statement was designed to do, and so in BQL the `SELECT` syntax is mimicked as much as possible. (Some basic knowledge of what the SQL `SELECT` statement does is assumed.) However, as opposed to the SQL data model, BQL's input data is assumed to be JSON-like, i.e., with varying shapes, nesting levels, and data types; therefore the BQL `SELECT` statement has a number of small differences to SQL's `SELECT`.

Overview

The general syntax of the `SELECT` command is

```
SELECT emit_operator select_list FROM table_expression;
```

The `emit_operator` is one of the operators described in *Relation-to-Stream Operators*. The following subsections describe the details of `select_list` and `table_expression`.

Table Expressions

A *table expression* computes a table. The table expression contains a `FROM` clause that is optionally followed by `WHERE`, `GROUP BY`, and `HAVING` clauses:

```
... FROM table_list [WHERE filter_expression]
    [GROUP BY group_list] [HAVING having_expression]
```

The `FROM` Clause

The `FROM` clause derives a table from one or more other tables given in a comma-separated table reference list.

```
FROM table_reference [, table_reference [, ...]]
```

In SQL, each `table_reference` is (in the simplest possible case) an identifier that refers to a pre-defined table, e.g., `FROM users` or `FROM names, addresses, cities` are valid SQL `FROM` clauses.

In BQL, only streams have identifiers, so in order to get a well-defined relation, a window specifier as explained in *Stream-to-Relation Operators* must be added. In particular, the examples just given for SQL `FROM` clauses are all *not* valid in BQL, but the following are:

```
FROM users [RANGE 10 TUPLES]

FROM names [RANGE 2 TUPLES], addresses [RANGE 1.5 SECONDS], cities [RANGE 200␣
↪MILLISECONDS]
```

Using Stream-Generating Functions

BQL also knows "user-defined stream-generating functions" (UDSFs) that transform a stream into another stream and can be used, for example, to output multiple output rows per input row; something that is not possible with standard `SELECT` features. (These are similar to "Table Functions" in PostgreSQL.) Such UDSFs can also be used in the `FROM` clause: Instead of using a stream's identifier, use the function call syntax `function(param, param, ...)` with the UDSF name as the function name and the base stream's identifiers as parameters (as a string, i.e., in double quotes), possibly with other parameters. For example, if there is a UDSF called `duplicate` that takes the input stream's name as the first parameter and the number of copies of each input tuple as the second, this would look as follows:

```
FROM duplicate("products", 3) [RANGE 10 SECONDS]
```

Table Joins

If more than one table reference is listed in the `FROM` clause, the tables are cross-joined (that is, the Cartesian product of their rows is formed). The syntax `table1 JOIN table2 ON (...)` is not supported in BQL. The result of the `FROM` list is an intermediate virtual table that can then be subject to transformations by the `WHERE`, `GROUP BY`, and `HAVING` clauses and is finally the result of the overall table expression.

Table Aliases

A temporary name can be given to tables and complex table references to be used for references to the derived table in the rest of the query. This is called a "table alias". To create a table alias, write

```
FROM table_reference AS alias
```

The use of table aliases is optional, but helps to shorten statements. By default, each table can be addressed using the stream name or the UDSF name, respectively. Therefore, table aliases are only mandatory if the same stream/UDSF is used multiple times in a join. Taking aliases into account, each name must uniquely refer to one table. `FROM stream [RANGE 1 TUPLES], stream [RANGE 2 TUPLES]` or `FROM streamA [RANGE 1 TUPLES], streamB [RANGE 2 TUPLES] AS streamA` are not valid, but `FROM stream [RANGE 1 TUPLES] AS streamA, stream [RANGE 2 TUPLES] AS streamB` and also `FROM stream [RANGE 1 TUPLES], stream [RANGE 2 TUPLES] AS other` are.

The `WHERE` Clause

The syntax of the `WHERE` clause is

```
WHERE filter_expression
```

where `filter_expression` is any expression with a boolean value. (That is, `WHERE 6` is not a valid filter, but `WHERE 6::bool` is.)

After the processing of the `FROM` clause is done, each row of the derived virtual table is checked against the search condition. If the result of the condition is true, the row is kept in the output table, otherwise (i.e., if the result is false or null) it is discarded. The search condition typically references at least one column of the table generated in the `FROM` clause; this is not required, but otherwise the `WHERE` clause will be fairly useless.

As BQL does not support the `table1 JOIN table2 ON (condition)` syntax, any join condition must always be given in the `WHERE` clause.

The `GROUP BY` and `HAVING` Clauses

After passing the `WHERE` filter, the derived input table might be subject to grouping, using the `GROUP BY` clause, and elimination of group rows using the `HAVING` clause. They basically have the same semantics as explained in the PostgreSQL Documentation, section 7.2.3

One current limitation of BQL row grouping is that only simple columns can be used in the `GROUP BY` list, no complex expressions are allowed. For example, `GROUP BY round(age/10)` cannot be used in BQL at the moment.

Select Lists

As shown in the previous section, the table expression in the SELECT command constructs an intermediate virtual table by possibly combining tables, views, eliminating rows, grouping, etc. This table is finally passed on to processing by the "select list". The select list determines which elements of the intermediate table are actually output.

Select-List Items

As in SQL, the select list contains a number of comma-separated expressions:

```
SELECT emit_operator expression [, expression] [...] FROM ...
```

In general, items of a select list can be arbitrary *Value Expressions*. In SQL, tables are strictly organized in "rows" and "columns" and the most important elements in such expressions are therefore column references.

In BQL, each input tuple can be considered a "row", but the data can also be unstructured and the notion of a "column" is not sufficient. (In fact, each row corresponds to a *map* object.) Therefore, BQL uses JSON Path to address data in

each row. If only one table is used in the `FROM` clause and only top-level keys of each JSON-like row are referenced, the BQL select list looks the same as in SQL:

```
SELECT RSTREAM a, b, c FROM input [RANGE 1 TUPLES];
```

If the input data has the form `{"a": 7, "b": "hello", "c": false}`, then the output will look exactly the same. However, JSON Path allows to access nested elements as well:

```
SELECT RSTREAM a.foo.bar FROM input [RANGE 1 TUPLES];
```

If the input data has the form `{"a": {"foo": {"bar": 7}}}`, then the output will be `{"col_0": 7}`. (See paragraph *Column Labels* below for details on output key naming, and the section *Field Selectors* for details about the available syntax for JSON Path expressions.)

Table Prefixes

Where SQL uses the dot in `SELECT left.a, right.b` to specify the table from which to use a column, JSON Path uses the dot to describe a child relation in a single JSON element as shown above. Therefore to avoid ambiguity, BQL uses the colon (`:`) character to separate table and JSON Path:

```
SELECT RSTREAM left:foo.bar, right:hoge FROM ...
```

If there is just one table to select from, the table prefix can be omitted, but then it must be omitted in *all* expressions of the statement. If there are multiple tables in the `FROM` clause, then table prefixes must be used.

Column Labels

The result value of every expression in the select list will be assigned to a key in the output row. If not explicitly specified, these output keys will be `"col_0"`, `"col_1"`, etc. in the order the expressions were specified in the select list. However, in some cases a more meaningful output key is chosen by default, as already shown above:

- If the expression is a single top-level key (like `a`), then the output key will be the same.
- If the expression is a simple function call (like `f(a)`), then the output key will be the function name.
- If the expression refers the timestamp of a tuple in a stream (using the `stream:ts()` syntax), then the output key will be `ts`.
- If the expression is the wildcard (`*`), then the input will be copied, i.e., all keys from the input document will be present in the output document.

The output key can be overridden by specifying an `... AS output_key` clause after an expression. For the example above,

```
SELECT RSTREAM a.foo.bar AS x FROM input [RANGE 1 TUPLES];
```

will result in an output row that has the shape `{"x": 7}` instead of `{"col_0": 7}`. Note that it is possible to use the same column label multiple times, but in this case it is undefined which of the values with the same alias will end up in that output key.

To place values at other places than the top level of an output row map, a subset of the JSON Path syntax described in *Field Selectors* can be used for column labels as well. Where such a selector describes the position in a map uniquely, the value will be placed at that location. For the input data example above,

```
SELECT RSTREAM a.foo.bar AS x.y[3].z FROM input [RANGE 1 TUPLES];
```

will result in an output document with the following shape:

```
{"x": {"y": [null, null, null, {"z": 7}]}}
```

That is, a string `child_key` in the column label hierarchy will assume a map at the corresponding position and put the value in that map using `child_key` as a key; a numeric index `[n]` will assume an array and put the value in the `n`-th position, padded with `NULL` items before if required. Negative list indices cannot be used. Also, *Extended Descend Operators* cannot be used.

It is safe to assign multiple values to non-overlapping locations of an output row created this way, as shown below:

```
SELECT RSTREAM 7 AS x.y[3].z, "bar" AS x.foo, 17 AS x.y[0]
  FROM input [RANGE 1 TUPLES];
```

This will create the following output row:

```
{"x": {"y": [17, null, null, {"z": 7}], "foo": "bar"}}
```

However, as the order in which the items of the select list are processed is not defined, it is not safe to override values placed by one select list item from another select list item. For example,

```
SELECT RSTREAM [1, 2, 3] AS x, 17 AS x[1] ...
```

does *not* guarantee a particular output. Also, statements such as

```
SELECT RSTREAM 1 AS x.y, 2 AS x[1] ...
```

will lead to errors because *x* can not be a map and an array at the same time.

Notes on Wildcards

In SQL, the wildcard (`*`) can be used as a shorthand expression for all columns of an input table. However, due to the strong typing in SQL's data model, name and type conflicts can still be checked at the time the statement is analyzed. In BQL's data model, there is no strong typing, therefore the wildcard operator must be used with a bit of caution. For example, in

```
SELECT RSTREAM * FROM left [RANGE 1 TUPLES], right [RANGE 1 TUPLES];
```

if the data in the `left` stream looks like `{"a": 1, "b": 2}` and the data in the `right` stream looks like `{"b": 3, "c": 4}`, then the output document will have the keys `a`, `b`, and `c`, but the value of the `b` key is undefined.

To select all keys from only one stream, the colon notation (`stream:*`) as introduced above can be used.

The wildcard can be used with a column alias as well. The expression `* AS foo` will nest the input document under the given key `foo`, i.e., input `{"a": 1, "b": 2}` is transformed to `{"foo": {"a": 1, "b": 2}}`.

On the other hand, it is also possible to use the wildcard as an alias, as in `foo AS *`. This will have the opposite effect, i.e., it takes the contents of the `foo` key (which *must* be a map itself) and pulls them up to top level, i.e., `{"foo": {"a": 1, "b": 2}}` is transformed to `{"a": 1, "b": 2}`.

Note that any name conflicts that arise due to the use of the wildcard operator (e.g., in `*`, `a:*`, `b:*`, `foo AS *`, `bar AS *`) lead to undefined values in the column with the conflicting name. However, if there is an explicitly specified output key, this will always be prioritized over a key originating from a wildcard expression.

Examples

Single Input Stream

Select List	Input Row	Output Row
`a`	`{"a": 1, "b": 2}`	`{"a": 1}`
`a, b`	`{"a": 1, "b": 2}`	`{"a": 1, "b": 2}`
`a + b`	`{"a": 1, "b": 2}`	`{"col_0": 3}`
`a, a + b`	`{"a": 1, "b": 2}`	`{"a": 1, "col_1": 3}`
`*`	`{"a": 1, "b": 2}`	`{"a": 1, "b": 2}`

Join on Two Streams `l` and `r`

Select List	Input Row (`l`)	Input Row (`r`)	Output Row
`l:a`	`{"a": 1, "b": 2}`	`{"c": 3, "d": 4}`	`{"a": 1}`
`l:a, r:c`	`{"a": 1, "b": 2}`	`{"c": 3, "d": 4}`	`{"a": 1, "c": 3}`
`l:a + r:c`	`{"a": 1, "b": 2}`	`{"c": 3, "d": 4}`	`{"col_0": 4}`
`l:*`	`{"a": 1, "b": 2}`	`{"c": 3, "d": 4}`	`{"a": 1, "b": 2}`
`l:*, r:c AS b`	`{"a": 1, "b": 2}`	`{"c": 3, "d": 4}`	`{"a": 1, "b": 3}`
`l:*, r:*`	`{"a": 1, "b": 2}`	`{"c": 3, "d": 4}`	`{"a": 1, "b": 2, "c": 3, "d": 4}`
`*`	`{"a": 1, "b": 2}`	`{"c": 3, "d": 4}`	`{"a": 1, "b": 2, "c": 3, "d": 4}`
`*`	`{"a": 1, "b": 2}`	`{"b": 3, "d": 4}`	`{"a": 1, "b": (undef.), "d": 4}`

Building Processing Pipelines

The `SELECT` statement as described above returns a data stream (where the transport mechanism depends on the client in use), but often an unattended processing pipeline (i.e., running on the server without client interaction) needs to set up. In order to do so, a stream can be created from the results of a `SELECT` query and then used afterwards like an input stream. (The concept is equivalent to that of an SQL `VIEW`.)

The statement used to create a stream from an SELECT statement is:

```
CREATE STREAM stream_name AS select_statement;
```

For example:

```
CREATE STREAM odds AS SELECT RSTREAM * FROM numbers [RANGE 1 TUPLES] WHERE id % 2 = 1;
```

If that statement is issued correctly, subsequent statements can refer to `stream_name` in their `FROM` clauses.

If a stream thus created is no longer needed, it can be dropped using the `DROP STREAM` command:

```
DROP STREAM stream_name;
```

Expression Evaluation

To evaluate expressions outside the context of a stream, the `EVAL` command can be used. The general syntax is

```
EVAL expression;
```

and `expression` can generally be any expression, but it cannot contain references to any columns, aggregate functions or anything that only makes sense in a stream processing context.

For example, in the SensorBee Shell, the following can be done:

```
> EVAL "foo" || "bar";
foobar
```

CHAPTER 9

Data Types and Conversions

This chapter describes data types defined in BQL and how their type conversion works.

Overview

BQL has following data types:

Type name	Description	Example
`null`	Null type	`NULL`
`bool`	Boolean	`true`
`int`	64-bit integer	`12`
`float`	64-bit floating point number	`3.14`
`string`	String	`"sensorbee"`
`blob`	Binary large object	A blob value cannot directly be written in BQL.
`timestamp`	Datetime information in UTC	A timestamp value cannot directly be written in BQL.
`array`	Array	`[1, "2", 3.4]`
`map`	Map with string keys	`{"a": 1, "b": "2", "c": 3.4}`

These types are designed to work well with JSON. They can be converted to or from JSON with some restrictions.

Note: User defined types are not available at the moment.

Types

This section describes the detailed specification of each type.

null

The type `null` only has one value: `NULL`, which represents an empty or undefined value.

`array` can contain `NULL` as follows:

```
[1, NULL, 3.4]
```

`map` can also contain `NULL` as its value:

```
{
    "some_key": NULL
}
```

This map is different from an empty map `{}` because the key `"some_key"` actually exists in the map but the empty map doesn't even have a key.

`NULL` is converted to `null` in JSON.

bool

The type `bool` has two values: `true` and `false`. In terms of a three-valued logic, `NULL` represents the third state, "unknown".

`true` and `false` are converted to `true` and `false` in JSON, respectively.

int

The type `int` is a 64-bit integer type. Its minimum value is `-9223372036854775808` and its maximum value is `+9223372036854775807`. Using an integer value out of this range result in an error.

Note: Due to bug #56 the current minimum value that can be parsed is actually `-9223372036854775807`.

An `int` value is converted to a number in JSON.

Note: Some implementations of JSON use 64-bit floating point number for all numerical values. Therefore, they might not be able to handle integers greater than or equal to 9007199254740992 (i.e. `2^53`) accurately.

---

float

The type `float` is a 64-bit floating point type. Its implementation is IEEE 754 on most platforms but some platforms could use other implementations.

A `float` value is converted to a number in JSON.

Note: Some expressions and functions may result in an infinity or a NaN. Because JSON doesn't have an infinity or a NaN notation, they will become `null` when they are converted to JSON.

---

string

The type `string` is similar to SQL's type `text`. It may contain an arbitrary length of characters. It may contain any valid UTF-8 character including a null character.

A `string` value is converted to a string in JSON.

blob

The type `blob` is a data type for any variable length binary data. There is no way to write a value directly in BQL yet, but there are some ways to use `blob` in BQL:

- Emitting a tuple containing a `blob` value from a source
- Casting a `string` encoded in base64 to `blob`
- Calling a function returning a `blob` value

A `blob` value is converted to a base64-encoded string in JSON.

timestamp

The type `timestamp` has date and time information in UTC. `timestamp` only guarantees precision in microseconds. There is no way to write a value directly in BQL yet, but there are some ways to use `blob` in BQL:

- Emitting a tuple containing a `timestamp` value from a source
- Casting a value of a type that is convertible to `timestamp`
- Calling a function returning a `timestamp` value

A `timestamp` value is converted to a string in RFC3339 format with nanosecond precision in JSON: `"2006-01-02T15:04:05.999999999Z07:00"`. Although the format can express nanoseconds, `timestamp` in BQL only guarantees microsecond precision as described above.

array

The type `array` provides an ordered sequence of values of any type, for example:

```
[1, "2", 3.4]
```

An `array` value can also contain another `array` or `map` as a value:

```
[
    [1, "2", 3.4],
    [
        ["4", 5.6, 7],
        [true, false, NULL],
        {"a": 10}
    ],
    {
        "nested_array": [12, 34.5, "67"]
    }
]
```

An `array` value is converted to an array in JSON.

map

The type `map` represents an unordered set of key-value pairs. A key needs to be a `string` and a value can be of any type:

```
{
    "a": 1,
    "b": "2",
    "c": 3.4
}
```

A `map` value can contain another `map` or `array` as its value:

```
{
    "a": {
        "aa": 1,
        "ab": "2",
        "ac": 3.4
    },
    "b": {
        "ba": {"a": 10},
        "bb": ["4", 5.6, 7],
        "bc": [true, false, NULL]
    },
    "c": [12, 34.5, "67"]
}
```

A `map` is converted to an object in JSON.

Conversions

BQL provides a `CAST(value AS type)` operator, or `value::type` as syntactic sugar, that converts the given value to a corresponding value in the given type, if those types are convertible. For example, `CAST(1 AS string)`, or `1::string`, converts an `int` value `1` to a `string` value and results in `"1"`. Converting to the same type as the value's type is valid. For instance, `"str"::string` does not do anything and results in `"str"`.

The following types are valid for the target type of `CAST` operator:

- `bool`
- `int`
- `float`
- `string`
- `blob`
- `timestamp`

Specifying `null`, `array`, or `map` as the target type results in an error.

This section describes how type conversions work in BQL.

Note: Converting a `NULL` value into any type results in `NULL` and it is not explicitly described in the subsections.

To `bool`

Following types can be converted to `bool`:

- `int`
- `float`
- `string`
- `blob`
- `timestamp`
- `array`
- `map`

From `int`

`0` is converted to `false`. Other values are converted to `true`.

From `float`

`0.0`, `-0.0`, and NaN are converted to `false`. Other values *including infinity* result in `true`.

From `string`

Following values are converted to `true`:

- `"t"`
- `"true"`
- `"y"`
- `"yes"`
- `"on"`
- `"1"`

Following values are converted to `false`:

- `"f"`
- `"false"`
- `"n"`
- `"no"`
- `"off"`
- `"0"`

Comparison is case-insensitive and leading and trailing whitespaces in a value are ignored. For example, `" tRuE "::bool` is `true`. Converting a value that is not mentioned above results in an error.

From `blob`

An empty `blob` value is converted to `false`. Other values are converted to `true`.

From `timestamp`

January 1, year 1, 00:00:00 UTC is converted to `false`. Other values are converted to `true`.

From `array`

An empty `array` is converted to `false`. Other values result in `true`.

From `map`

An empty `map` is converted to `false`. Other values result in `true`.

To `int`

Following types can be converted to `int`:

- `bool`
- `float`
- `string`
- `timestamp`

From `bool`

`true::int` results in 1 and `false::int` results in 0.

From `float`

Converting a `float` value into a `int` value truncates the decimal part. That is, for positive numbers it results in the greatest `int` value less than or equal to the `float` value, for negative numbers it results in the smallest `int` value greater than or equal to the `float` value:

```
1.0::int  -- => 1
1.4::int  -- => 1
1.5::int  -- => 1
2.01::int -- => 2
(-1.0)::int  -- => -1
(-1.4)::int  -- => -1
(-1.5)::int  -- => -1
(-2.01)::int -- => -2
```

The conversion results in an error when the `float` value is out of the valid range of `int` values.

From `string`

When converting a `string` value into an `int` value, `CAST` operator tries to parse it as an integer value. If the string contains a `float`-shaped value (even if it is `"1.0"`), conversion fails.

```
"1"::int   -- => 1
```

The conversion results in an error when the `string` value contains a number that is out of the valid range of `int` values, or the value isn't a number. For example, `"1a"::string` results in an error even though the value starts with a number.

From `timestamp`

A `timestamp` value is converted to an `int` value as the number of full seconds elapsed since January 1, 1970 UTC:

```
("1970-01-01T00:00:00Z"::timestamp)::int        -- => 0
("1970-01-01T00:00:00.123456Z"::timestamp)::int -- => 0
("1970-01-01T00:00:01Z"::timestamp)::int         -- => 1
("1970-01-02T00:00:00Z"::timestamp)::int        -- => 86400
("2016-01-18T09:22:40.123456Z"::timestamp)::int -- => 1453108960
```

To `float`

Following types can be converted to `float`:

- `bool`
- `int`
- `string`
- `timestamp`

From `bool`

`true::float` results in 1.0 and `false::float` results in 0.0.

From `int`

`int` values are converted to the nearest `float` values:

```
1::float -- => 1.0
((9000000000000012345::float)::int)::string -- => "9000000000000012288"
```

From `string`

A `string` value is parsed and converted to the nearest `float` value:

```
"1.1"::float   -- => 1.1
"1e-1"::float  -- => 0.1
"-1e+1"::float -- => -10.0
```

From `timestamp`

A `timestamp` value is converted to a `float` value as the number of seconds (including a decimal part) elapsed since January 1, 1970 UTC. The integral part of the result contains seconds and the decimal part contains microseconds:

```
("1970-01-01T00:00:00Z"::timestamp)::float        -- => 0.0
("1970-01-01T00:00:00.000001Z"::timestamp)::float -- => 0.000001
("1970-01-02T00:00:00.000001Z"::timestamp)::float -- => 86400.000001
```

To `string`

Following types can be converted to `string`:

- `bool`
- `int`
- `float`
- `blob`
- `timestamp`
- `array`
- `map`

From `bool`

`true::string` results in `"true"`, `false::string` results in `"false"`.

Note: Keep in mind that casting the string `"false"` back to boolean results in the `true` value as described above.

From `int`

A `int` value is formatted as a signed decimal integer:

```
1::string     -- => "1"
(-24)::string -- => "-24"
```

From `float`

A `float` value is formatted as a signed decimal floating point. Scientific notation is used when necessary:

```
1.2::string           -- => "1.2"
10000000000.0::string -- => "1e+10"
```

From `blob`

A `blob` value is converted to a `string` value encoded in base64.

Note: Keep in mind that the `blob`/`string` conversion using `CAST` *always* involves base64 encoding/decoding. It is not possible to see the single bytes of a `blob` using only the `CAST` operator. If there is a source that emits `blob` data where it is *known* that this is actually a valid UTF-8 string (for example, JSON or XML data), the interpretation "as a string" (as opposed to "to string") must be performed by a UDF.

From `timestamp`

A `timestamp` value is formatted in RFC3339 format with nanosecond precision: `"2006-01-02T15:04:05.999999999Z07:00"`.

From `array`

An `array` value is formatted as a JSON array:

```
[1, "2", 3.4]::string -- => "[1,""2"",3.4]"
```

From `map`

A `map` value is formatted as a JSON object:

```
{"a": 1, "b": "2", "c": 3.4}::string -- => "{""a"":1,""b"":""2"",""c"":3.4}"
```

To `timestamp`

Following types can be converted to `timestamp`:

- `int`
- `float`
- `string`

From `int`

An `int` value to be converted to a `timestamp` value is assumed to have the number of seconds elapsed since January 1, 1970 UTC:

```
0::timestamp          -- => 1970-01-01T00:00:00Z
1::timestamp          -- => 1970-01-01T00:00:01Z
1453108960::timestamp -- => 2016-01-18T09:22:40Z
```

From `float`

An `float` value to be converted to a `timestamp` value is assumed to have the number of seconds elapsed since January 1, 1970 UTC. Its integral part should have seconds and decimal part should have microseconds:

```
0.0::timestamp -- => 1970-01-01T00:00:00Z
0.000001::timestamp -- => 1970-01-01T00:00:00.000001Z
86400.000001::timestamp -- => 1970-01-02T00:00:00.000001Z
```

From `string`

A `string` value is parsed in RFC3339 format, or RFC3339 with nanosecond precision format:

```
"1970-01-01T00:00:00Z"::timestamp        -- => 1970-01-01T00:00:00Z
"1970-01-01T00:00:00.000001Z"::timestamp -- => 1970-01-01T00:00:00.000001Z
"1970-01-02T00:00:00.000001Z"::timestamp -- => 1970-01-02T00:00:00.000001Z
```

Converting ill-formed `string` values to `timestamp` results in an error.

CHAPTER 10

Operators

This chapter introduces operators used in BQL.

Arithmetic Operators

BQL provides the following arithmetic operators:

Operator	Description	Example	Result
`+`	Addition	`6 + 1`	`7`
`-`	Subtraction	`6 - 1`	`5`
`+`	Unary plus	`+4`	`4`
`-`	Unary minus	`-4`	`-4`
`*`	Multiplication	`3 * 2`	`6`
`/`	Division	`7 / 2`	`3`
`%`	Modulo	`5 % 3`	`2`

All operators accept both integers and floating point numbers. Integers and floating point numbers can be mixed in a single arithmetic expression. For example, `3 + 5 * 2.5` is valid.

Note: Unary minus operators can be applied to a value multiple times. However, each unary minus operators must be separated by a space like `- - -3` because `--` and succeeding characters are parsed as a comment. For example, `---3` is parsed as `--` and a comment body `-3`.

String Operators

BQL provides the following string operators:

Operator	Description	Example	Result
`||`	Concatenation	`"Hello" || ", world"`	`"Hello, world"`

`||` only accepts strings and `NULL`. For example, `"1" || 2` results in an error. When one operand is `NULL`, the result is also `NULL`. For instance, `NULL || "str"`, `"str" || NULL`, and `NULL || NULL` result in `NULL`.

Comparison Operators

BQL provides the following comparison operators:

Operator	Description	Example	Result
<	Less than	`1 < 2`	`true`
>	Greater than	`1 > 2`	`false`
<=	Less than or equal to	`1 <= 2`	`true`
>=	Greater than or equal to	`1 >= 2`	`false`
=	Equal to	`1 = 2`	`false`
<> or !=	Not equal to	`1 != 2`	`true`
`IS NULL`	Null check	`false IS NULL`	`false`
`IS NOT NULL`	Non-null check	`false IS NOT NULL`	`true`

All comparison operators return a boolean value.

<, >, <=, and >= are only valid when

1. either both operands are numeric values (i.e. integers or floating point numbers)
2. or have the same type *and* that type is comparable.

The following types are comparable:

- `null`
- `int`
- `float`
- `string`
- `timestamp`

Valid examples are as follows:

- `1 < 2.1`
  - Integers and floating point numbers can be compared.
- `"abc" > "def"`
- `1::timestamp <= 2::timestamp`
- **`NULL > "a"`**
  - This expression is valid although it always results in `NULL`. See *NULL Comparison* below.

=, <>, and != are valid for any type even if both operands have different types. When the types of operands are different, = results in `false`; <> and != return `true`. (However, integers and floating point numbers can be compared, for example `1 = 1.0` returns `true`.) When operands have the same type, = results in `true` if both values are equivalent and others return `false`.

Note: Floating point values with the value `NaN` are treated specially as per the underlying floating point implementation. In particular, = comparison will always be false if one or both of the operands is `NaN`.

NULL Comparison

In a three-valued logic, comparing any value with NULL results in NULL. For example, all of following expressions result in NULL:

- 1 < NULL
- 2 > NULL
- "a" <= NULL
- 3 = NULL
- NULL = NULL
- NULL <> NULL

Therefore, do **not** look for NULL values with expression = NULL. To check if a value is NULL or not, use IS NULL or IS NOT NULL operator. expression IS NULL operator returns true only when an expression is NULL.

Note: [NULL] = [NULL] and {"a": NULL} = {"a": NULL} result in true although it contradict the three-valued logic. This specification is provided for convenience. Arrays or maps often have NULL to indicate that there's no value for a specific key but the key actually exists. In other words, {"a": NULL, "b": 1} and {"b": 1} are different. Therefore, NULL in arrays and maps are compared as if it's a regular value. Unlike NULL, comparing NaN floating point values always results in false.

Presence/Absence Check

In BQL, the JSON object {"a": 6, "b": NULL} is different from {"a": 6}. Therefore, when accessing b in the latter object, the result is not NULL but an error. To check whether a key is present in a map, the following operators can be used:

Operator	Description	Example	Example Input	Result
IS MISSING	Absence Check	b IS MISSING	{"a": 6}	true
IS NOT MISSING	Presence Check	b IS NOT MISSING	{"a": 6}	false

Since the presence/absence check is done before the value is actually extracted from the map, only JSON Path expressions can be used with IS [NOT] MISSING, not arbitrary expressions. For example, a + 2 IS MISSING is not a valid expression.

Logical Operators

BQL provides the following logical operators:

Operator	Description	Example	Result
AND	Logical and	1 < 2 AND 2 < 3	true
OR	Logical or	1 < 2 OR 2 > 3	true
NOT	Logical negation	NOT 1 < 2	false

Logical operators also follow the three-valued logic. For example, true AND NULL and NULL OR false result in NULL.

CHAPTER 11

Functions

BQL provides a number of built-in functions that are described in this chapter. Function names and meaning of parameters have been heavily inspired by PostgreSQL. However, be aware that the accepted and returned types may differ as there is no simple mapping between BQL and SQL data types. See the *Function Reference* for details about each function's behavior.

Numeric Functions

General Functions

The table below shows some common mathematical functions that can be used in BQL.

Function	Description
abs(x)	absolute value
cbrt(x)	cube root
ceil(x)	round up to nearest integer
degrees(x)	radians to degrees
div(y, x)	integer quotient of `y`/`x`
exp(x)	exponential
floor(x)	round down to nearest integer
ln(x)	natural logarithm
log(x)	base 10 logarithm
log(b, x)	logarithm to base b
mod(y, x)	remainder of `y`/`x`
pi()	"π" constant
power(a, b)	`a` raised to the power of `b`
radians(x)	degrees to radians
round(x)	round to nearest integer
sign(x)	sign of the argument (-1, 0, +1)
sqrt(x)	square root
trunc(x)	truncate toward zero
width_bucket(x, l, r, c)	bucket of `x` in a histogram

Pseudo-Random Functions

The table below shows functions for generating pseudo-random numbers.

Function	Description
random()	random value in the range $0.0 <= x < 1.0$
setseed(x)	set seed ($-1.0 <= x <= 1.0$) for subsequent `random()` calls

Trigonometric Functions

Finally, the table below shows the available trigonometric functions.

Function	Description
acos(x)	inverse cosine
asin(x)	inverse sine
atan(x)	inverse tangent
cos(x)	cosine
cot(x)	cotangent
sin(x)	sine
tan(x)	tangent

String Functions

The table below shows some common functions for strings that can be used in BQL.

Function	Description
bit_length(s)	number of bits in string
btrim(s)	remove whitespace from the start/end of `s`
btrim(s, chars)	remove `chars` from the start/end of `s`
char_length(s)	number of characters in `s`
concat(s [, ...])	concatenate all arguments
concat_ws(sep, s [, ...])	concatenate arguments `s` with separator
format(s, [x, ...])	format arguments using a format string
lower(s)	convert `s` to lower case
ltrim(s)	remove whitespace from the start of `s`
ltrim(s, chars)	remove `chars` from the start of `s`
md5(s)	MD5 hash of `s`
octet_length(s)	number of bytes in `s`
overlay(s, r, from)	replace substring
overlay(s, r, from, for)	replace substring
rtrim(s)	remove whitespace from the end of `s`
rtrim(s, chars)	remove `chars` from the end of `s`
sha1(s)	SHA1 hash of `s`
sha256(s)	SHA256 hash of `s`
strpos(s, t)	location of substring `t` in `s`
substring(s, r)	extract substring matching regex `r` from `s`
substring(s, from)	extract substring
substring(s, from, for)	extract substring
upper(s)	convert `s` to upper case

Time Functions

Function	Description
distance_us(u, v)	signed temporal distance from `u` to `v` in microseconds
clock_timestamp()	current date and time (changes during statement execution)
now()	date and time when processing of current tuple was started

Array Functions

Function	Description
array_length(a)	number of elements in an array

Other Scalar Functions

Function	Description
coalesce(x [, ...])	return first non-null input parameter

Aggregate Functions

Aggregate functions compute a single result from a set of input values. The built-in normal aggregate functions are listed in the table below. The special syntax considerations for aggregate functions are explained in *Aggregate*

Expressions.

Function	Description
array_agg(x)	input values, including nulls, concatenated into an array
avg(x)	the average (arithmetic mean) of all input values
bool_and(x)	true if all input values are true, otherwise false
bool_or(x)	true if at least one input value is true, otherwise false
count(x)	number of input rows for which `x` is not null
count(*)	number of input rows
json_object_agg(k, v)	aggregates name/value pairs as a map
max(x)	maximum value of `x` across all input values
median(x)	the median of all input values
min(x)	minimum value of `x` across all input values
string_agg(x, sep)	input values concatenated into a string, separated by `sep`
sum(x)	sum of `x` across all input values

Part IV

Server Programming

This part describes the extensibility of the SensorBee server. Topics covered in this part are advanced and should be read after understanding the basics of SensorBee and BQL.

Because SensorBee is mainly written in The Go Programming Language, understanding the language before reading this part is also recommended. A Tour of Go is an official tutorial and is one of the best tutorials of the language. It runs on web browsers and does not require any additional software installation. After learning the language, How to Write Go Code helps to understand how to use the `go` tool and the standard way to develop Go packages and applications.

This part assumes that the `go` tool is installed and the development environment including Go's environment variables like `GOROOT` or `GOPATH` is appropriately set up. SensorBee requires Go 1.4 or later.

CHAPTER 12

Extending the SensorBee Server and BQL

Many features of the server and BQL can be extended by plugins. This chapter describes what types of features can be extended by users. Following chapters in this part describes how to develop those features in specific programming languages.

User-Defined Functions

A user-defined function (UDF) is a function that is implemented by a user and registered in the SensorBee server. Once it is registered, it be called from BQL statements:

```
SELECT RSTREAM my_udf(field) FROM src [RANGE 1 TUPLES];
```

A UDF behaves just like a built-in function. A UDF can take an arbitrary number of arguments. Each argument can be any of *built-in types* and can receive multiple types of values. A UDF can also support a variable-length argument. A UDF has a single return value of any built-in type. When multiple return values are required, a UDF can return the value as an `array`.

Note: BQL currently does not support `CREATE FUNCTION` statements like well-known RDBMSs. UDFs can only be added through Go programs.

User-Defined Aggregate Functions

A user-defined aggregate function (UDAF) is a function similar to a UDF but can take aggregation parameters (see *Aggregate Expressions*) as arguments in addition to regular arguments.

User-Defined Stream-Generating Functions

Stream-generating functions can also be user-defined. There are two types of UDSFs. The first type behaves like a source, which is not connected to any input stream and generates tuples proactively:

```
... FROM my_counter() [RANGE ...
```

`my_counter` above may emit tuples like `{"count": 1}`.

This type of UDSFs are called source-like UDSFs.

The other type is called a stream-like UDSF and behaves just like a stream, which receives tuples from one or more incoming streams or sources. It receives names of streams or sources as its arguments:

```
... FROM my_udsf("another_stream", "yet_another_stream", other_params) [RANGE ...
```

Note that there is no rule on how to define UDFS's arguments. Thus, the order and the use of arguments depend on each UDFS. For example, a UDFS might take an `array` of `string` containing names of input streams as its first argument: `my_union(["stream1", "stream2", "stream3"])`. Names of input stream do not even need to be located at the beginning of the argument list: `my_udfs2(1, "another_stream")` is also possible.

Using UDSFs is a very powerful way to extend BQL since they can potentially do anything that the `SELECT` cannot do.

User-Defined States

A user-defined state (UDS) can be provided to support stateful data processing (see *Stateful Data Processing*). A UDS is usually provided with a set of UDFs that manipulate the state. Those UDFs take the name of the UDS as a `string` argument:

```
CREATE STATE event_id_seq TYPE snowflake_id WITH machine_id = 1;
CREATE STREAM events_with_id AS
    SELECT snowflake_id("event_id_seq"), * FROM events [RANGE 1 TUPLES];
```

In the example above, a UDS `event_id_seq` is created with the type `snowflake_id`. Then, the UDS is passed to the UDF `snowflake_id`, which happens to have the same name as the type name of the UDS. The UDF looks up the UDS `event_id_seq` and returns a value computed based on the state.

Source Plugins

A source type developed by a user can be added to the SensorBee server as a plugin so that it can be used in `CREATE SOURCE` statements. A source type can have any number of required and optional parameters. Each parameter can have any of *built-in types*.

Sink Plugins

A sink type developed by a user can be added to the SensorBee server as a plugin so that it can be used in `CREATE SINK` statement. A sink type can have any number of required and optional parameters. Each parameter can have any of *built-in types*.

CHAPTER 13

Extensions in Go

This chapter describes how to extend the SensorBee server in the Go programming language.

Development Flow of Components in Go

The typical development flow of components like a UDF, a UDS type, a source type, or a sink type should be discussed before looking into details of each component.

The basic flow is as follows:

1. Create a git repository for components
2. Implement components
3. Create a plugin subpackage in the repository

Create a Git Repository for Components

Components are written in Go, so they need to be in a valid git repository (or a repository of a different version control system). One repository may provide multiple types of components. For example, a repository could have 10 UDFs, and 5 UDS types, 2 source types, and 1 sink type. However, since Go is very well designed to provide packages in a fine-grained manner, each repository should only provide a minimum set of components that are logically related and make sense to be in the same repository.

Implement Components

The next step is to implement components. There is no restriction on which standard or 3rd party packages to depend on.

Functions or structs that are to be registered to the SensorBee server need to be referred to by the plugin subpackage, which is described in the next subsection. Thus, names of those symbols need to start with a capital letter.

In this step, components should not be registered to the SensorBee server yet.

Create a Plugin Subpackage in the Repository

It is highly recommended that the repository has a separate package (i.e. a subdirectory) which only registers components to the SensorBee server. There is usually one file named "plugin.go" in the plugin package and it only contains a series of registration function calls in `init` function. For instance, if the repository only provides one UDF, the contents of `plugin.go` would look like:

```
// in github.com/user/myudf/plugin/plugin.go
package plugin

import (
    "gopkg.in/sensorbee/sensorbee.v0/bql/udf"
    "github.com/user/myudf"
)

func init() {
    udf.MustRegisterGlobalUDF("my_udf", &myudf.MyUDF{})
}
```

There are two reasons to have a plugin subpackage separated from the implementation of components. First, by separating them, other Go packages can import the components to use the package as a library without registering them to SensorBee. Second, having a separated plugin package allows a user to register a component with a different name. This is especially useful when names of components conflict each other.

To use the example plugin above, the `github.com/user/myudf/plugin` package needs to be added to the plugin path list of SensorBee.

Repository Organization

The typical organization of the repository is

- `github.com/user/repo`
  - `README`: description and the usage of components in the repository
  - `.go` files: implementation of components
  - `plugin/`: a subpackage for the plugin registration
    - `plugin.go`
  - `othersubpackages/`: there can be optional subpackages

User-Defined Functions

This section describes how to write a UDF in Go. It first shows the basic interface of defining UDFs, and then describes utilities around it, how to develop a UDF in a Go-ish manner, and a complete example.

Implementing a UDF

Note: This is a very low-level way to implement a UDF in Go. To learn about an easier way, see *Generic UDFs*.

Any struct implementing the following interface can be used as a UDF:

```
type UDF interface {
    // Call calls the UDF.
    Call(*core.Context, ...data.Value) (data.Value, error)

    // Accept checks if the function accepts the given number of arguments
    // excluding core.Context.
    Accept(arity int) bool

    // IsAggregationParameter returns true if the k-th parameter expects
    // aggregated values. A UDF with Accept(n) == true is an aggregate
    // function if and only if this function returns true for one or more
    // values of k in the range 0, ..., n-1.
    IsAggregationParameter(k int) bool
}
```

This interface is defined in the `gopkg.in/sensorbee/sensorbee.v0/bql/udf` package.

A UDF can be *registered* via the `RegisterGlobalUDF` or `MustRegisterGlobalUDF` functions from the same package. `MustRegisterGlobalUDF` is the same as `RegisterGlobalUDF` but panics on failure instead of returning an error. These functions are usually called from the `init` function in the UDF package's `plugin` sub-package. A typical implementation of a UDF looks as follows:

```
type MyUDF struct {
    ...
}

func (m *MyUDF) Call(ctx *core.Context, args ...data.Value) (data.Value, error) {
    ...
}

func (m *MyUDF) Accept(arity int) bool {
    ...
}

func (m *MyUDF) IsAggregationParameter(k int) bool {
    ...
}

func init() {
    // MyUDF can be used as my_udf in BQL statements.
    udf.MustRegisterGlobalUDF("my_udf", &MyUDF{})
}
```

As it can be inferred from this example, a UDF itself should be stateless since it only registers one instance of a struct as a UDF and it will be shared globally. Stateful data processing can be achieved by the combination of UDFs and UDSs, which is described in *User-Defined States*.

A UDF needs to implement three methods to satisfy `udf.UDF` interface: `Call`, `Accept`, and `IsAggregationParameter`.

The `Call` method receives a `*core.Context` and multiple `data.Value` as its arguments. `*core.Context` contains the information of the current processing context. `Call`'s `...data.Value` argument holds the values passed to the UDF. `data.Value` represents a value used in BQL and can be any of *built-in types*.

```
SELECT RSTREAM my_udf(arg1, arg2) FROM stream [RANGE 1 TUPLES];
```

In this example, `arg1` and `arg2` are passed to the `Call` method:

```
func (m *MyUDF) Call(ctx *core.Context, args ...data.Value) (data.Value, error) {
    // When my_udf(arg1, arg2) is called, len(args) is 2.
    // args[0] is arg1 and args[1] is arg2.
    // It is guaranteed that m.Accept(len(args)) is always true.
}
```

Because `data.Value` is a semi-variant type, the `Call` method needs to check the type of each `data.Value` and convert it to a desired type.

The `Accept` method verifies if the UDF accepts the specific number of arguments. It can return true for multiple arities as long as it can receive the given number of arguments. If a UDF only accepts two arguments, the method is implemented as follows:

```
func (m *MyUDF) Accept(arity int) bool {
    return arity == 2
}
```

When a UDF aims to support variadic parameters (a.k.a. variable-length arguments) with two required arguments (e.g. `my_udf(arg1, arg2, optional1, optional2, ...)`), the implementation would be:

```
func (m *MyUDF) Accept(arity int) bool {
    return arity >= 2
}
```

Finally, `IsAggregationParameter` returns whether the *k*-th argument (starting from 0) is an aggregation parameter. Aggregation parameters are passed as a `data.Array` containing all values of a field in each group.

All of these methods can be called concurrently from multiple goroutines and they must be thread-safe.

The registered UDF is looked up based on its name and the number of argument passed to it.

```
SELECT RSTREAM my_udf(arg1, arg2) FROM stream [RANGE 1 TUPLES];
```

In this `SELECT`, a UDF having the name `my_udf` is looked up first. After that, its `Accept` method is called with 2 and `my_udf` is actually selected if `Accept(2)` returned true. `IsAggregationParameter` method is additionally called on each argument to see if the argument needs to be an aggregation parameter. Then, if there is no mismatch, `my_udf` is finally called.

Note: A UDF does not have a schema at the moment, so any error regarding types of arguments will not be reported until the statement calling the UDF actually processes a tuple.

Generic UDFs

SensorBee provides a helper function to register a regular Go function as a UDF without implementing the `UDF` interface explicitly.

```
func Inc(v int) int {
    return v + 1
}
```

This function `Inc` can be transformed into a UDF by `ConvertGeneric` or `MustConvertGeneric` function defined in the `gopkg.in/sensorbee/sensorbee.v0/bql/udf` package. By combining it with `RegisterGlobalUDF`, the `Inc` function can easily be registered as a UDF:

```
func init() {
    udf.MustRegisterGlobalUDF("inc", udf.MustConvertGeneric(Inc))
}
```

So, a complete example of the UDF implementation and registration is as follows:

```
package inc

import (
    "gopkg.in/sensorbee/sensorbee.v0/bql/udf"
)

func Inc(v int) int {
    return v + 1
}

func init() {
    udf.MustRegisterGlobalUDF("inc", udf.MustConvertGeneric(Inc))
}
```

Note: A UDF implementation and registration should actually be separated to different packages. See *Development Flow of Components in Go* for details.

Although this approach is handy, there is some small overhead compared to a UDF implemented in the regular way. Most of such overhead comes from type checking and conversions.

Functions passed to `ConvertGeneric` need to satisfy some restrictions on the form of their argument and return value types. Each restriction is described in the following subsections.

Form of Arguments

In terms of valid argument forms, there are some rules to follow:

1. A Go function can receive `*core.Context` as the first argument, or can omit it.
2. A function can have any number of arguments including 0 arguments as long as Go accepts them.
3. A function can be variadic with or without non-variadic parameters.

There are basically eight (four times two, whether a function has `*core.Context` or not) forms of arguments (return values are intentionally omitted for clarity):

- Functions receiving no argument in BQL (e.g. `my_udf()`)
    1. `func(*core.Context)`: A function only receiving `*core.Context`
    2. `func()`: A function having no argument and not receiving `*core.Context`, either
- Functions having non-variadic arguments but no variadic arguments
    3. `func(*core.Context, T1, T2, ..., Tn)`
    4. `func(T1, T2, ..., Tn)`
- Functions having variadic arguments but no non-variadic arguments
    5. `func(*core.Context, ...T)`
    6. `func(...T)`

- Functions having both variadic and non-variadic arguments
  7. `func(*core.Context, T1, T2, ..., Tn, ...Tn+1)`
  8. `func(T1, T2, ..., Tn, ...Tn+1)`

Here are some examples of invalid function signatures:

- `func(T, *core.Context)`: `*core.Context` must be the first argument.
- `func(NonSupportedType)`: Only supported types, which will be explained later, can be used.

Although return values are omitted from all the examples above, they are actually required. The next subsection explains how to define valid return values.

Form of Return Values

All functions need to have return values. There are two forms of return values:

- `func(...) R`
- `func(...) (R, error)`

All other forms are invalid:

- `func(...)`
- `func(...) error`
- `func(...) NonSupportedType`

Valid types of return values are same as the valid types of arguments, and they are listed in the following subsection.

Valid Value Types

The list of Go types that can be used for arguments and the return value is as follows:

- `bool`
- signed integers: `int`, `int8`, `int16`, `int32`, `int64`
- unsigned integers: `uint`, `uint8`, `uint16`, `uint32`, `uint64`
- `float32`, `float64`
- `string`
- `time.Time`
- data: `data.Bool`, `data.Int`, `data.Float`, `data.String`, `data.Blob`, `data.Timestamp`, `data.Array`, `data.Map`, `data.Value`
- A slice of any type above, including `data.Value`

`data.Value` can be used as a semi-variant type, which will receive all types above.

When the argument type and the actual value type are different, weak type conversion are applied to values. Conversions are basically done by `data.ToXXX` functions (see godoc comments of each function in `data/type_conversions.go`). For example, `func inc(i int) int` can be called by `inc("3")` in a BQL statement and it will return 4. If a strict type checking or custom type conversion is required, receive values as `data.Value` and manually check or convert types, or define the UDF in the regular way.

Examples of Valid Go Functions

The following functions can be converted to UDFs by `ConvertGeneric` or `MustConvertGeneric` function:

- `func rand() int`
- `func pow(*core.Context, float32, float32) (float32, error)`
- `func join(*core.Context, ...string) string`
- `func format(string, ...data.Value) (string, error)`
- `func keys(data.Map) []string`

Complete Examples

This subsection shows three example UDFs:

- `my_inc`
- `my_join`
- `my_join2`

Assume that these are in the repository `github.com/sensorbee/examples/udfs` (which actually does not exist). The repository has three files:

- `inc.go`
- `join.go`
- `plugin/plugin.go`

inc.go

In `inc.go`, the `Inc` function is defined as a pure Go function with a standard value type:

```
package udfs

func Inc(v int) int {
    return v + 1
}
```

join.go

In `join.go`, the `Join` UDF is defined in a strict way. It also performs strict type checking. It is designed to be called in one of two forms: `my_join("a", "b", "c", "separator")` or `my_join(["a", "b", "c"], "separator")`. Each argument and value in the array must be a string. The UDF receives an arbitrary number of arguments.

```
package udfs

import (
    "errors"
    "strings"

    "pfi/sensorbee/sensorbee/core"
    "pfi/sensorbee/sensorbee/data"
```

```
)

type Join struct {
}

func (j *Join) Call(ctx *core.Context, args ...data.Value) (data.Value, error) {
    empty := data.String("")
    if len(args) == 1 {
        return empty, nil
    }

    switch args[0].Type() {
    case data.TypeString: // my_join("a", "b", "c", "sep") form
        var ss []string
        for _, v := range args {
            s, err := data.AsString(v)
            if err != nil {
                return empty, err
            }
            ss = append(ss, s)
        }
        return data.String(strings.Join(ss[:len(ss)-1], ss[len(ss)-1])), nil

    case data.TypeArray: // my_join(["a", "b", "c"], "sep") form
        if len(args) != 2 {
            return empty, errors.New("wrong number of arguments for my_join(array,␣
↪sep)")
        }
        sep, err := data.AsString(args[1])
        if err != nil {
            return empty, err
        }

        a, _ := data.AsArray(args[0])
        var ss []string
        for _, v := range a {
            s, err := data.AsString(v)
            if err != nil {
                return empty, err
            }
            ss = append(ss, s)
        }
        return data.String(strings.Join(ss, sep)), nil

    default:
        return empty, errors.New("the first argument must be a string or an array")
    }
}

func (j *Join) Accept(arity int) bool {
    return arity >= 1
}

func (j *Join) IsAggregationParameter(k int) bool {
    return false
}
```

plugin/plugin.go

In addition to `Inc` and `Join`, this file registers the standard Go function `strings.Join` as `my_join2`. Because it's converted to a UDF by `udf.MustConvertGeneric`, arguments are weakly converted to given types. For example, `my_join2([1, 2.3, "4"], "-")` is valid although `strings.Join` itself is `func([]string, string) string`.

```
package plugin

import (
    "strings"

    "pfi/sensorbee/sensorbee/bql/udf"

    "pfi/nobu/docexamples/udfs"
)

func init() {
    udf.MustRegisterGlobalUDF("my_inc", udf.MustConvertGeneric(udfs.Inc))
    udf.MustRegisterGlobalUDF("my_join", &udfs.Join{})
    udf.MustRegisterGlobalUDF("my_join2", udf.MustConvertGeneric(strings.Join))
}
```

Evaluating Examples

Once the `sensorbee` command is built with those UDFs and a topology is created on the server, the `EVAL` statement can be used to test them:

```
EVAL my_inc(1); -- => 2
EVAL my_inc(1.5); -- => 2
EVAL my_inc("10"); -- => 11

EVAL my_join("a", "b", "c", "-"); -- => "a-b-c"
EVAL my_join(["a", "b", "c"], ",") -- => "a,b,c"
EVAL my_join(1, "b", "c", "-") -- => error
EVAL my_join([1, "b", "c"], ",") -- => error

EVAL my_join2(["a", "b", "c"], ",") -- => "a,b,c"
EVAL my_join2([1, "b", "c"], ",") -- => "1,b,c"
```

Dynamic Loading

Dynamic loading of UDFs written in Go is not supported at the moment because Go does not support loading packages dynamically.

User-Defined Stream-Generating Functions

This section describes how to write a UDSF in Go.

Implementing a UDSF

To provide a UDSF, two interfaces need to be implemented: `UDSF` and `UDSFCreator`.

The interface `UDSF` is defined as follows in the `gopkg.in/sensorbee/sensorbee.v0/bql/udf` package.

```
type UDSF interface {
    Process(ctx *core.Context, t *core.Tuple, w core.Writer) error
    Terminate(ctx *core.Context) error
}
```

The `Process` method processes an input tuple and emits computed tuples for subsequent streams. `ctx` contains the processing context information. `t` is the tuple to be processed in the UDSF. `w` is the destination to where computed tuples are emitted. The `Terminate` method is called when the UDFS becomes unnecessary. The method has to release all the resources the UDSF has.

How the `Process` method is called depends on the type of a UDSF. When a UDFS is a stream-like UDSF (i.e. it has input from other streams), the `Process` method is called every time a new tuple arrives. The argument `t` contains the tuple emitted from another stream. Stream-like UDSFs have to return from `Process` immediately after processing the input tuple. They must not block in the method.

A stream-like UDSF is used mostly when multiple tuples need to be computed and emitted based on one input tuple:

```
type WordSplitter struct {
    field string
}

func (w *WordSplitter) Process(ctx *core.Context, t *core.Tuple, writer core.Writer)␣
↪error {
    var kwd []string
    if v, ok := t.Data[w.field]; !ok {
        return fmt.Errorf("the tuple doesn't have the required field: %v", w.field)
    } else if s, err := data.AsString(v); err != nil {
        return fmt.Errorf("'%v' field must be string: %v", w.field, err)
    } else {
        kwd = strings.Split(s, " ")
    }

    for _, k := range kwd {
        out := t.Copy()
        out.Data[w.field] = data.String(k)
        if err := writer.Write(ctx, out); err != nil {
            return err
        }
    }
    return nil
}

func (w *WordSplitter) Terminate(ctx *core.Context) error {
    return nil
}
```

`WordSplitter` splits text in a specific field by space. For example, when an input tuple is `{"word": "a b c"}` and `WordSplitter.Field` is `word`, following three tuples will be emitted: `{"word": "a"}`, `{"word": "b"}`, and `{"word": "c"}`.

When a UDSF is a source-like UDSF, the `Process` method is only called once with a tuple that does not mean anything. Unlike a stream-like UDSF, the `Process` method of a source-like UDSF does not have to return until it has emitted all tuples, the `Terminate` method is called, or a fatal error occurs.

```
type Ticker struct {
    interval time.Duration
    stopped  int32
```

```
}

func (t *Ticker) Process(ctx *core.Context, tuple *core.Tuple, w core.Writer) error {
    var i int64
    for ; atomic.LoadInt32(&t.stopped) == 0; i++ {
        newTuple := core.NewTuple(data.Map{"tick": data.Int(i)})
        if err := w.Write(ctx, newTuple); err != nil {
            return err
        }
        time.Sleep(t.interval)
    }
    return nil
}

func (t *Ticker) Terminate(ctx *core.Context) error {
    atomic.StoreInt32(&t.stopped, 1)
    return nil
}
```

In this example, `Ticker` emits tuples having `tick` field containing a counter until the `Terminate` method is called.

Whether a UDSF is stream-like or source-like can be configured when it is created by `UDSFCreator`. The interface `UDSFCreator` is defined as follows in `gopkg.in/sensorbee/sensorbee.v0/bql/udf` package:

```
type UDSFCreator interface {
    CreateUDSF(ctx *core.Context, decl UDSFDeclarer, args ...data.Value) (UDSF, error)
    Accept(arity int) bool
}
```

The `CreateUDSF` method creates a new instance of a UDSF. The method is called when evaluating a UDSF in the `FROM` clause of a `SELECT` statement. `ctx` contains the processing context information. `decl` is used to customize the behavior of the UDSF, which is explained later. `args` has arguments passed in the `SELECT` statement. The `Accept` method verifies if the UDSF accept the specific number of arguments. This is the same as `UDF.Arity` method (see *User-Defined Functions*).

`UDSFDeclarer` is used in the `CreateUDSF` method to customize the behavior of a UDSF:

```
type UDSFDeclarer interface {
    Input(name string, config *UDSFInputConfig) error
    ListInputs() map[string]*UDSFInputConfig
}
```

By calling its `Input` method, a UDSF will be able to receive tuples from another stream with the name `name`. Because the `name` is given outside the UDSF, it's uncontrollable from the UDSF. However, there are cases that a UDSF wants to know from which stream a tuple has come. For example, when providing a UDSF performing a JOIN or two streams, a UDSF needs to distinguish which stream emitted the tuple. If the UDSF was defined as `my_join(left_stream, right_stream)`, `decl` can be used as follows in `UDSFCreator.CreateUDSF`:

```
decl.Input(args[0], &UDSFInputConfig{InputName: "left"})
decl.Input(args[1], &UDSFInputConfig{InputName: "right"})
```

By configuring the input stream in this way, a tuple passed to `UDSF.Process` has the given name in its `Tuple.InputName` field:

```
func (m *MyJoin) Process(ctx *core.Context, t *core.Tuple, w core.Writer) error {
    switch t.InputName {
    case "left":
        ... process tuples from left_stream ...
```

```
    case "right":
        ... process tuples from right_stream ...
    }
    ...
}
```

If a UDSF is configured to have one or more input streams by `decl.Input` in the `UDSFCreator.CreateUDSF` method, the UDSF is processed as a stream-like UDSF. Otherwise, if a UDSF doesn't have any input (i.e. `decl.Input` is not called), the UDSF becomes a source-like UDSF.

As an example, the `UDSFCreator` of `WordSplitter` is shown below:

```
type WordSplitterCreator struct {
}

func (w *WordSplitterCreator) CreateUDSF(ctx *core.Context,
    decl udf.UDSFDeclarer, args ...data.Value) (udf.UDSF, error) {
    input, err := data.AsString(args[0])
    if err != nil {
        return nil, fmt.Errorf("input stream name must be a string: %v", args[0])
    }
    field, err := data.AsString(args[1])
    if err != nil {
        return nil, fmt.Errorf("target field name must be a string: %v", args[1])
    }
    // This Input call makes the UDSF a stream-like UDSF.
    if err := decl.Input(input, nil); err != nil {
        return nil, err
    }
    return &WordSplitter{
        field: field,
    }, nil
}

func (w *WordSplitterCreator) Accept(arity int) bool {
    return arity == 2
}
```

Although the UDSF has not been registered to the SensorBee server yet, it could appear like `word_splitter(input_stream_name, target_field_name)` if it was registered with the name `word_splitter`.

For another example, the `UDSFCreator` of `Ticker` is shown below:

```
type TickerCreator struct {
}

func (t *TickerCreator) CreateUDSF(ctx *core.Context,
    decl udf.UDSFDeclarer, args ...data.Value) (udf.UDSF, error) {
    interval, err := data.ToDuration(args[0])
    if err != nil {
        return nil, err
    }
    // Since this is a source-like UDSF, there's no input.
    return &Ticker{
        interval: interval,
    }, nil
}
```

```
func (t *TickerCreator) Accept(arity int) bool {
    return arity == 1
}
```

Like `word_splitter`, its signature could be `ticker(interval)` if the UDSF is registered as `ticker`.

The implementation of this UDSF is completed and the next step is to register it to the SensorBee server.

Registering a UDSF

A UDSF can be used in BQL by registering its `UDSFCreator` interface to the SensorBee server using the `RegisterGlobalUDSFCreator` or `MustRegisterGlobalUDSFCreator` functions, which are defined in `gopkg.in/sensorbee/sensorbee.v0/bql/udf`.

The following example registers `WordSplitter` and `Ticker`:

```
func init() {
    udf.RegisterGlobalUDSFCreator("word_splitter", &WordSplitterCreator{})
    udf.RegisterGlobalUDSFCreator("ticker", &TickerCreator{})
}
```

Generic UDSFs

Just like UDFs have a `ConvertGeneric` function, UDSFs also have `ConvertToUDSFCreator` and `MustConvertToUDSFCreator` function. They convert a regular function satisfying some restrictions to the `UDSFCreator` interface.

The restrictions are the same as for *generic UDFs* except that a function converted to the `UDSFCreator` interface has an additional argument `UDSFDeclarer`. `UDSFDeclarer` is located after `*core.Context` and before other arguments. Examples of valid function signatures are show below:

- `func(*core.Context, UDSFDeclarer, int)`
- `func(UDSFDeclarer, string)`
- `func(UDSFDeclarer)`
- `func(*core.Context, UDSFDeclarer, ...data.Value)`
- `func(UDSFDeclarer, ...float64)`
- `func(*core.Context, UDSFDeclarer, int, ...string)`
- `func(UDSFDeclarer, int, float64, ...time.Time)`

Unlike `*core.Context`, `UDSFDeclarer` cannot be omitted. The same set of types can be used for arguments as types that `ConvertGeneric` function accepts.

`WordSplitterCreator` can be rewritten with the `ConvertToUDSFCreator` function as follows:

```
func CreateWordSplitter(decl udf.UDSFDeclarer,
    inputStream, field string) (udf.UDSF, error) {
    if err := decl.Input(inputStream, nil); err != nil {
        return nil, err
    }
    return &WordSplitter{
        field: field,
    }, nil
```

```
}

func init() {
    udf.RegisterGlobalUDSFCreator("word_splitter",
        udf.MustConvertToUDSFCreator(WordSplitterCreator))
}
```

`TickerCreator` can be replaced with `ConvertToUDSFCreator`, too:

```
func CreateTicker(decl udf.UDSFDeclarer, i data.Value) (udf.UDSF, error) {
    interval, err := data.ToDuration(i)
    if err != nil {
        return nil, err
    }
    return &Ticker{
        interval: interval,
    }, nil
}

func init() {
    udf.MustRegisterGlobalUDSFCreator("ticker",
       udf.MustConvertToUDSFCreator(udsfs.CreateTicker))
}
```

A Complete Example

This subsection provides a complete example of UDSFs described in this section. In addition to `word_splitter` and `ticker`, the example also includes the `lorem` source, which periodically emits random texts as `{"text": "lorem ipsum dolor sit amet"}`.

Assume that the import path of the example repository is `github.com/sensorbee/examples/udsfs`, which doesn't actually exist. The repository has four files:

- `lorem.go`
- `splitter.go`
- `ticker.go`
- `plugin/plugin.go`

lorem.go

To learn how to implement a source plugin, see *Source Plugins*.

```
package udsfs

import (
    "math/rand"
    "strings"
    "time"

    "gopkg.in/sensorbee/sensorbee.v0/bql"
    "gopkg.in/sensorbee/sensorbee.v0/core"
    "gopkg.in/sensorbee/sensorbee.v0/data"
)
```

```
var (
    Lorem = strings.Split(strings.Replace(`lorem ipsum dolor sit amet
consectetur adipiscing elit sed do eiusmod tempor incididunt ut labore et dolore
magna aliqua Ut enim ad minim veniam quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat Duis aute irure dolor in reprehenderit
in voluptate velit esse cillum dolore eu fugiat nulla pariatur Excepteur sint
occaecat cupidatat non proident sunt in culpa qui officia deserunt mollit anim
id est laborum`, "\n", " ", -1), " ")
)

type LoremSource struct {
    interval time.Duration
}

func (l *LoremSource) GenerateStream(ctx *core.Context, w core.Writer) error {
    for {
        var text []string
        for l := rand.Intn(5) + 5; l > 0; l-- {
            text = append(text, Lorem[rand.Intn(len(Lorem))])
        }

        t := core.NewTuple(data.Map{
            "text": data.String(strings.Join(text, " ")),
        })
        if err := w.Write(ctx, t); err != nil {
            return err
        }

        time.Sleep(l.interval)
    }
}

func (l *LoremSource) Stop(ctx *core.Context) error {
    return nil
}

func CreateLoremSource(ctx *core.Context,
    ioParams *bql.IOParams, params data.Map) (core.Source, error) {
    interval := 1 * time.Second
    if v, ok := params["interval"]; ok {
        i, err := data.ToDuration(v)
        if err != nil {
            return nil, err
        }
        interval = i
    }
    return core.ImplementSourceStop(&LoremSource{
        interval: interval,
    }), nil
}
```

splitter.go

```
package udsfs

import (
```

```
    "fmt"
    "strings"

    "gopkg.in/sensorbee/sensorbee.v0/bql/udf"
    "gopkg.in/sensorbee/sensorbee.v0/core"
    "gopkg.in/sensorbee/sensorbee.v0/data"
)

type WordSplitter struct {
    field string
}

func (w *WordSplitter) Process(ctx *core.Context,
    t *core.Tuple, writer core.Writer) error {
    var kwd []string
    if v, ok := t.Data[w.field]; !ok {
        return fmt.Errorf("the tuple doesn't have the required field: %v", w.field)
    } else if s, err := data.AsString(v); err != nil {
        return fmt.Errorf("'%v' field must be string: %v", w.field, err)
    } else {
        kwd = strings.Split(s, " ")
    }

    for _, k := range kwd {
        out := t.Copy()
        out.Data[w.field] = data.String(k)
        if err := writer.Write(ctx, out); err != nil {
            return err
        }
    }
    return nil
}

func (w *WordSplitter) Terminate(ctx *core.Context) error {
    return nil
}

func CreateWordSplitter(decl udf.UDSFDeclarer,
    inputStream, field string) (udf.UDSF, error) {
    if err := decl.Input(inputStream, nil); err != nil {
        return nil, err
    }
    return &WordSplitter{
        field: field,
    }, nil
}
```

ticker.go

```
package udsfs

import (
    "sync/atomic"
    "time"

    "gopkg.in/sensorbee/sensorbee.v0/bql/udf"
```

```
    "gopkg.in/sensorbee/sensorbee.v0/core"
    "gopkg.in/sensorbee/sensorbee.v0/data"
)

type Ticker struct {
    interval time.Duration
    stopped  int32
}

func (t *Ticker) Process(ctx *core.Context, tuple *core.Tuple, w core.Writer) error {
    var i int64
    for ; atomic.LoadInt32(&t.stopped) == 0; i++ {
        newTuple := core.NewTuple(data.Map{"tick": data.Int(i)})
        if err := w.Write(ctx, newTuple); err != nil {
            return err
        }
        time.Sleep(t.interval)
    }
    return nil
}

func (t *Ticker) Terminate(ctx *core.Context) error {
    atomic.StoreInt32(&t.stopped, 1)
    return nil
}

func CreateTicker(decl udf.UDSFDeclarer, i data.Value) (udf.UDSF, error) {
    interval, err := data.ToDuration(i)
    if err != nil {
        return nil, err
    }
    return &Ticker{
        interval: interval,
    }, nil
}
```

plugin/plugin.go

```
package plugin

import (
    "gopkg.in/sensorbee/sensorbee.v0/bql"
    "gopkg.in/sensorbee/sensorbee.v0/bql/udf"

    "github.com/sensorbee/examples/udsfs"
)

func init() {
    bql.MustRegisterGlobalSourceCreator("lorem",
        bql.SourceCreatorFunc(udsfs.CreateLoremSource))
    udf.MustRegisterGlobalUDSFCreator("word_splitter",
        udf.MustConvertToUDSFCreator(udsfs.CreateWordSplitter))
    udf.MustRegisterGlobalUDSFCreator("ticker",
        udf.MustConvertToUDSFCreator(udsfs.CreateTicker))
}
```

Example BQL Statements

```
CREATE SOURCE lorem TYPE lorem;
CREATE STREAM lorem_words AS
    SELECT RSTREAM * FROM word_splitter("lorem", "text") [RANGE 1 TUPLES];
```

Results of `word_splitter` can be received by the following `SELECT`:

```
SELECT RSTREAM * FROM lorem_words [RANGE 1 TUPLES];
```

User-Defined States

This section describes how to write a UDS in Go.

Implementing a UDS

A struct implementing the following interface can be used as a UDS:

```
type SharedState interface {
    Terminate(ctx *Context) error
}
```

This interface is defined in `gopkg.in/sensorbee/sensorbee.v0/core` package.

`Terminate` method is called when the UDS becomes no longer in use. It should release any resource that the UDS has allocated so far.

As an example, a UDS having a monotonically increasing counter can be implemented as follows:

```
type Counter struct {
    c int64
}

func (c *Counter) Terminate(ctx *core.Context) error {
    return nil
}

func (c *Counter) Next() int64 {
    return atomic.AddInt64(&c.c, 1)
}
```

At the moment, there's no way to manipulate the UDS from BQL statements. UDSs are usually provided with a set of UDFs that read or update the UDS. It'll be described later in *Manipulating a UDS via a UDF*. Before looking into the UDS manipulation, registering and creating a UDS needs to be explained.

Registering a UDS

To register a UDS to the SensorBee server , the UDS needs to provide its `UDSCreator`. `UDSCreator` is an interface defined in `gopkg.in/sensorbee/sensorbee.v0/bql/udf` package as follows:

```
type UDSCreator interface {
    CreateState(ctx *core.Context, params data.Map) (core.SharedState, error)
}
```

`UDSCreator.CreateState` method is called when executing a `CREATE STATE` statement. The method creates a new instance of the UDS and initializes it with the given parameters. The argument `ctx` has the processing context information and `params` has parameters specified in the `WITH` clause of the `CREATE STATE`.

The creator can be registered by `RegisterGlobalUDSCreator` or `MustRegisterGlobalUDSCreator` function defined in `gopkg.in/sensorbee/sensorbee.v0/bql/udf` package.

The following is the implementation and the registration of the creator for `Counter` UDS above:

```
type CounterCreator struct {
}

func (c *CounterCreator) CreateState(ctx *core.Context,
    params data.Map) (core.SharedState, error) {
    cnt := &Counter{}
    if v, ok := params["start"]; ok {
        i, err := data.ToInt(v)
        if err != nil {
            return nil, err
        }
        cnt.c = i - 1
    }
    return cnt, nil
}

func init() {
    udf.MustRegisterGlobalUDSCreator("my_counter", &CounterCreator{})
}
```

The creator in this example is registered with the UDS type name `my_counter`. The creator supports `start` parameter which is used as the first value that `Counter.Next` returns. The parameter can be specified in the `CREATE STATE` as follows:

```
CREATE STATE my_counter_instance TYPE my_counter WITH start = 100;
```

Because the creator creates a new instance every time the `CREATE STATE` is executed, there can be multiple instances of a specific UDS type:

```
CREATE STATE my_counter_instance1 TYPE my_counter;
CREATE STATE my_counter_instance2 TYPE my_counter;
CREATE STATE my_counter_instance3 TYPE my_counter;
...
```

Once an instance of the UDS is created by the `CREATE STATE`, UDFs can refer them and manipulate their state.

udf.UDSCreatorFunc

A function having the same signature as `UDSCreator.CreateState` can be converted into `UDSCreator` by by `udf.UDSCreatorFunc` utility function:

```
func UDSCreatorFunc(f func(*core.Context, data.Map) (core.SharedState, error))␣
↪UDSCreator
```

For example, `CounterCreator` can be defined as a function and registered as follows with this utility:

```
func CreateCounter(ctx *core.Context,
    params data.Map) (core.SharedState, error) {
```

```
    cnt := &Counter{}
    if v, ok := params["start"]; ok {
        i, err := data.ToInt(v)
        if err != nil {
            return nil, err
        }
        cnt.c = i - 1
    }
    return cnt, nil
}

func init() {
    udf.MustRegisterGlobalUDSCreator("my_counter",
        &udf.UDSCreatorFunc(CreateCounter))
}
```

To support `SAVE STATE` and `LOAD STATE` statements, however, this utility function cannot be used because the creator needs to have the `LoadState` method. How to support saving and loading is described later.

Manipulating a UDS via a UDF

To manipulate a UDS from BQL statements, a set of UDFs that read or update the UDS has to be provided with it:

```
func Next(ctx *core.Context, uds string) (int64, error) {
    s, err := ctx.SharedStates.Get(uds)
    if err != nil {
        return 0, err
    }

    c, ok := s.(*Counter)
    if !ok {
        return 0, fmt.Errorf("the state isn't a counter: %v", uds)
    }
    return c.Next(), nil
}

func init() {
    udf.MustRegisterGlobalUDF("my_next_count", udf.MustConvertGeneric(Next))
}
```

In this example, a UDF `my_next_count` is registered to the SensorBee server. The UDF calls `Counter.Next` method to obtain the next count and returns it. The UDF receives one argument `uds` that is the name of the UDS to be updated.

```
CREATE STATE my_counter_instance TYPE my_counter;
CREATE STREAM events_with_id AS
    SELECT RSTREAM my_next_count("my_counter_instance") AS id, *
    FROM events [RANGE 1 TUPLES];
```

The BQL statements above add IDs to tuples emitted from a stream `events`. The state `my_counter_instance` is created with the type `my_counter`. Then, `my_next_count` UDF is called with the name. Every time the UDF is called, the state of `my_counter_instance` is updated by its `Next` method.

`my_next_count` (i.e. `Next` function in Go) can look up the instance of the UDS by its name through `core.Context.SharedStates`. `SharedStates` manages all the UDSs created in a topology. `SharedState.Get` returns the instance of the UDS having the given name. It returns an error if it couldn't find the instance. In the

example above, `my_next_count("my_counter_instance")` will look up an instance of the UDS having the name `my_counter_instance`, which was previously created by the `CREATE STATE`. The UDS returned from `Get` method has the type `core.SharedState` and cannot directly be used as `Counter`. Therefore, it has to be cast to `*Counter`.

Since the state can be any type satisfying `core.SharedState`, a UDS can potentially have any information such as machine learning models, dictionaries for natural language processing, or even an in-memory database.

Note: As UDFs are concurrently called from multiple goroutines, UDSs also needs to be thread-safe.

Saving and Loading a UDS

`Counter` implemented so far doesn't support saving and loading its state. Thus, its count will be reset every time the server restarts. To save the state and load it later on, the UDS and its creator need to provide some methods. After providing those method, the state can be saved by the `SAVE STATE` statement and loaded by `LOAD STATE` statement.

Supporting `SAVE STATE`

By adding `Save` method having the following signature to a UDS, the UDS can be saved by the `SAVE STATE` statement:

```
Save(ctx *core.Context, w io.Writer, params data.Map) error
```

`Save` method writes all the data that the state has to `w io.Writer`. The data can be written in any format as long as corresponding loading methods can reconstruct the state from it. It can be in JSON, msgpack, Protocol Buffer, and so on.

Warning: Providing forward/backward compatibility or version controlling of the saved data is the responsibility of the author of the UDS.

`*core.Context` has the processing context information. `params` argument is not used at the moment and reserved for the future use.

Once Save method is provided, the UDS can be saved by `SAVE STATE` statement:

```
SAVE STATE my_counter_instance;
```

The `SAVE STATE` doesn't take any parameters now. The location and the physical format of the saved UDS data depend on the configuration of the SensorBee server or program running BQL statements. However, it is guaranteed that the saved data can be loaded by the same program via the `LOAD STATE` statement, which is described later.

`Save` method of previously implemented `Counter` can be as follows:

```
func (c *Counter) Save(ctx *core.Context, w io.Writer, params data.Map) error {
    return binary.Write(w, binary.LittleEndian, atomic.LoadInt64(&c.c))
}
```

Note: Because this counter is very simple, there's no version controlling logic in the method. As the minimum solution, having a version number at the beginning of the data is sufficient for most cases.

Supporting `LOAD STATE`

To support the `LOAD STATE` statement, a `UDSCreator` needs to have `LoadState` method having the following signature:

```
LoadState(ctx *core.Context, r io.Reader, params data.Map) (core.SharedState, error)
```

Note: `LoadState` method needs to be defined in a `UDSCreator`, not in the UDS itself.

`LoadState` method reads data from `r io.Reader`. The data has exactly the same format as the one previously written by `Save` method of a UDS. `params` has parameters specified in the `SET` clause in the `LOAD STATE` statement.

Note: Parameters specified in the `SET` clause doesn't have to be same as ones given in the `WITH` clause of the `CREATE STATE` statement. See *LOAD STATE* for details.

When `LoadState` method returns an error, the `LOAD STATE` statement with `CREATE IF NOT STATE` doesn't fallback to `CREATE STATE`, but it just fails.

Once `LoadState` method is added to the `UDSCreator`, the saved state can be loaded by `LOAD STATE` statement.

`LoadState` method of previously implemented `CounterCreator` can be as follows:

```
func (c *CounterCreator) LoadState(ctx *core.Context, r io.Reader,
    params data.Map) (core.SharedState, error) {
    cnt := &Counter{}
    if err := binary.Read(r, binary.LittleEndian, &cnt.c); err != nil {
        return nil, err
    }
    return cnt, nil
}
```

Providing `Load` method in a UDS

In addition to implementing `LoadState` method in a UDS's creator, a UDS itself can provide `Load` method. While `LoadState` method creates a new state instance and replace it with the previous instance, `Load` method dynamically modifies the existing instance. Therefore, `Load` method can potentially be more efficient than `LoadState` method although it has to provide appropriate failure handling and concurrency control so that (1) the UDS doesn't become invalid on failure (i.e. `Load` methods is "exception safe") or by concurrent calls, and (2) other operations on the UDS don't block for a long time.

The signature of `Load` method is almost the same as `LoadState` method except that `Load` method doesn't return a new `core.SharedState` but updates the UDS itself instead:

```
Load(ctx *Context, r io.Reader, params data.Map) error
```

`Load` method of previously implemented `Counter` can be as follows:

```
func (c *Counter) Load(ctx *core.Context, r io.Reader, params data.Map) error {
    var cnt int64
    if err := binary.Read(r, binary.LittleEndian, &cnt); err != nil {
        return err
    }
    atomic.StoreInt64(&c.c, cnt)
```

```
    return nil
}
```

How Loading is Processed

SensorBee tries to use these two loading methods `LoadState` and `Load` in the following rule:

1. When a UDS's creator doesn't provide `LoadState` method, the `LOAD STATE` statement fails.
    - The `LOAD STATE` statement fails even if the UDS implements its `Load` method. To support the statement, `LoadState` method is always required in its creator. This is because `Load` method only works when an instance of the UDS is already created or loaded, and it cannot be used for a nonexistent instance.
    - The `LOAD STATE CREATE IF NOT SAVED` statement also fails if `LoadState` method isn't provided. The statement calls `CreateState` method when the state hasn't previously been saved. Otherwise, it'll try to load the saved data. Therefore, if the data is previously saved and an instance of the UDS hasn't been created yet, the statement cannot create a new instance without `LoadState` method in the creator. To be consistent on various conditions, the `LOAD STATE CREATE IF NOT SAVED` statement fails if `LoadState` method isn't provided regardless of whether the state has been saved before or not.
2. When a UDS's creator provides `LoadState` method and the UDS doesn't provide `Load` method, the `LOAD STATE` statement tries to load a model through `LoadState` method.
    - It will create a new instance so that it consumes twice as much memory.
3. When a UDS's creator provides `LoadState` method and the UDS also provides `Load` method,
    - `Load` method will be used when the instance has already been created or loaded.
        - `LoadState` method wouldn't be used even if `Load` method failed.
    - `LoadState` method will be used otherwise.

Note: This is already mentioned in the list above, but `LoadState` method always needs to be provided even if a UDS implements `Load` method.

A Complete Example

A complete example of the state is shown in this subsection. Assume that the import path of the example repository is `github.com/sensorbee/examples/counter`, which doesn't actually exist. The repository has two files:

- counter.go
- plugin/plugin.go

counter.go

```
package counter

import (
    "encoding/binary"
    "fmt"
    "io"
    "sync/atomic"
```

```
    "gopkg.in/sensorbee/sensorbee.v0/core"
    "gopkg.in/sensorbee/sensorbee.v0/data"
)

type Counter struct {
    c int64
}

func (c *Counter) Terminate(ctx *core.Context) error {
    return nil
}

func (c *Counter) Next() int64 {
    return atomic.AddInt64(&c.c, 1)
}

func (c *Counter) Save(ctx *core.Context, w io.Writer, params data.Map) error {
    return binary.Write(w, binary.LittleEndian, atomic.LoadInt64(&c.c))
}

func (c *Counter) Load(ctx *core.Context, r io.Reader, params data.Map) error {
    var cnt int64
    if err := binary.Read(r, binary.LittleEndian, &cnt); err != nil {
        return err
    }
    atomic.StoreInt64(&c.c, cnt)
    return nil
}

type CounterCreator struct {
}

func (c *CounterCreator) CreateState(ctx *core.Context,
    params data.Map) (core.SharedState, error) {
    cnt := &Counter{}
    if v, ok := params["start"]; ok {
        i, err := data.ToInt(v)
        if err != nil {
            return nil, err
        }
        cnt.c = i - 1
    }
    return cnt, nil
}

func (c *CounterCreator) LoadState(ctx *core.Context, r io.Reader,
    params data.Map) (core.SharedState, error) {
    cnt := &Counter{}
    if err := binary.Read(r, binary.LittleEndian, &cnt.c); err != nil {
        return nil, err
    }
    return cnt, nil
}

func Next(ctx *core.Context, uds string) (int64, error) {
    s, err := ctx.SharedStates.Get(uds)
    if err != nil {
        return 0, err
```

```
    }

    c, ok := s.(*Counter)
    if !ok {
        return 0, fmt.Errorf("the state isn't a counter: %v", uds)
    }
    return c.Next(), nil
}
```

plugin/plugin.go

```
package plugin

import (
    "gopkg.in/sensorbee/sensorbee.v0/bql/udf"

    "github.com/sensorbee/examples/counter"
)

func init() {
    udf.MustRegisterGlobalUDSCreator("my_counter",
        &counter.CounterCreator{})
    udf.MustRegisterGlobalUDF("my_next_count",
        udf.MustConvertGeneric(counter.Next))
}
```

Writing Tuples to a UDS

When a UDS implements `core.Writer`, the `INSERT INTO` statement can insert tuples into the UDS via the `uds` sink:

```
type Writer interface {
    Write(*Context, *Tuple) error
}
```

The following is the example of using the `uds` sink:

```
CREATE STATE my_state TYPE my_state_type;
CREATE SINK my_state_sink TYPE uds WITH name = "my_state";
INSERT INTO my_state_sink FROM some_stream;
```

If `my_state_type` doesn't implement `core.Writer`, the `CREATE SINK` statement fails. Every time `some_stream` emits a tuple, the `Write` method of `my_state` is called.

Example

Models provided by Jubatus machine learning plugin for SensorBee implement the `Write` method. When tuples are inserted into a UDS, it trains the model it has.

Source Plugins

This section describes how to implement a source as a plugin of SensorBee.

Implementing a Source

A struct implementing the following interface can be a source:

```
type Source interface {
    GenerateStream(ctx *Context, w Writer) error
    Stop(ctx *Context) error
}
```

This interface is defined in `gopkg.in/sensorbee/sensorbee.v0/core` package.

The `GenerateStream` methods actually generate tuples for subsequent streams. The argument `ctx` contains the information of the current processing context. `w` is the destination to where generated tuples are emitted. The `Stop` method stops `GenerateStream`. It should wait until the `GenerateStream` method call returns, but it isn't mandatory.

Once the `GenerateStream` method is called, a source can emit as many tuples as many tuples as it requires. A source basically needs to return from its `GenerateStream` method when:

1. it emitted all the tuples it has
2. the `Stop` method was called
3. a fatal error occurred

The `Stop` method can be called concurrently while the `GenerateStream` method is working and it must be thread-safe. As long as a source is used by components defined in SensorBee, it's guaranteed that its `Stop` method is called only once and it doesn't have to be idempotent. However, it is recommended that a source provide a termination check in its `Stop` method to avoid a double free problem.

A typical implementation of a source is shown below:

```
func (s *MySource) GenerateStream(ctx *core.Context, w core.Writer) error {
    <initialization>
    defer func() {
        <clean up>
    }()

    for <check stop> {
        t := <create a new tuple>
        if err := w.Write(ctx, t); err != nil {
            return err
        }
    }
    return nil
}

func (s *MySource) Stop(ctx *core.Context) error {
    <turn on a stop flag>
    <wait until GenerateStream stops>
    return nil
}
```

The following example source emits tuple periodically:

```
type Ticker struct {
    interval time.Duration
    stopped  int32
}

func (t *Ticker) GenerateStream(ctx *core.Context, w core.Writer) error {
    var cnt int64
    for ; ; cnt++ {
        if atomic.LoadInt32(&t.stopped) != 0 {
            break
        }

        tuple := core.NewTuple(data.Map{"tick": data.Int(cnt)})
        if err := w.Write(ctx, tuple); err != nil {
            return err
        }
        time.Sleep(t.interval)
    }
    return nil
}

func (t *Ticker) Stop(ctx *core.Context) error {
    atomic.StoreInt32(&t.stopped, 1)
    return nil
}
```

The `interval` field is initialized in `SourceCreator`, which is described later. This is the source version of the example in UDSF's section. This implementation is a little wrong since the `Stop` method doesn't wait until the `GenerateStream` method actually returns. Because implementing a thread-safe source which stops correctly is a difficult task, `core` package provides a utility function that implements a source's `Stop` method on behalf of the source itself. See *Utilities* for details.

Registering a Source

To register a source to the SensorBee server, the source needs to provide its `SourceCreator`. The `SourceCreator` interface is defined in `gopkg.in/sensorbee/sensorbee.v0/bql` pacakge as follows:

```
type SourceCreator interface {
    CreateSource(ctx *core.Context, ioParams *IOParams, params data.Map) (core.Source,
↪ error)
}
```

It only has one method: `CreateSource`. The `CreateSource` method is called when the *CREATE SOURCE* statement is executed. The `ctx` argument contains the information of the current processing context. `ioParams` has the name and the type name of the source, which are given in the `CREATE SOURCE` statement. `params` has parameters specified in the `WITH` clause of the `CREATE SOURCE` statement.

The creator can be registered by `RegisterGlobalSourceCreator` or `MustRegisterGlobalSourceCreator` function. As an example, the creator of `Ticker` above can be implemented and registered as follows:

```
type TickerCreator struct {
}

func (t *TickerCreator) CreateSource(ctx *core.Context,
    ioParams *bql.IOParams, params data.Map) (core.Source, error) {
```

```
    interval := 1 * time.Second
    if v, ok := params["interval"]; ok {
        i, err := data.ToDuration(v)
        if err != nil {
            return nil, err
        }
        interval = i
    }
    return &Ticker{
        interval: interval,
    }, nil
}

func init() {
    bql.MustRegisterGlobalSourceCreator("ticker", &TickerCreator{})
}
```

In this example, the source has a parameter `interval` which can be specified in the `WITH` clause of the `CREATE SOURCE` statement:

```
CREATE SOURCE my_ticker TYPE ticker WITH interval = 0.1;
```

`my_ticker` emits tuples that look like `{"tick": 123}` in every 100ms. Without the `interval` parameter, `my_ticker` will emit tuples in every one second by default.

Types of a Source

In addition to a regular source, there're two more types of sources: a resumable source and a rewindable source. This subsection describes those sources in detail.

Resumable Sources

A source that supports *PAUSE SOURCE* and the *RESUME SOURCE* statements are called a resumable source.

Although all sources support them by default, which is done by the `core` package, a source can explicitly implement `core.Resumable` interface so that it can provide more efficient pause and resume capability:

```
type Resumable interface {
    Pause(ctx *Context) error
    Resume(ctx *Context) error
}
```

The `Pause` method is called when `PAUSE SOURCE` statement is executed and the `Resume` method is called by `RESUME SOURCE`. The `Pause` method may be called even when the source is already paused, so is the `Resume` method.

A source can be non-resumable by implementing these method to return an error:

```
type MyNonResumableSource struct {
    ...
}

...

func (m *MyNonResumableSource) Pause(ctx *core.Context) error {
    return errors.New("my_non_resumable_source doesn't support pause")
```

```
}

func (m *MyNonResumableSource) Resume(ctx *core.Context) error {
    return errors.New("my_non_resumable_source doesn't support resume")
}
```

Rewindable Sources

A rewindable source can re-generate the same tuples again from the beginning after it emits all tuples or while it's emitting tuples. A rewindable source supports the *REWIND SOURCE* statement.

A source can become rewindable by implementing the `core.RewindableSource` interface:

```
type RewindableSource interface {
    Source
    Resumable

    Rewind(ctx *Context) error
}
```

A rewindable source also needs to implement `core.Resumable` to be rewindable.

Note: The reason that a rewindable source also needs to be resumable is due to the internal implementation of the default pause/resume support. While a source is paused, it blocks `core.Writer.Write` called in the `GenerateStream` method. The `Rewind` method could also be blocked while the `Write` call is being blocked until the `Resume` method is called. It, of course, depends on the implementation of a source, but it's very error-prone. Therefore, implementing the `Resumable` interface is required to be rewindable at the moment.

Unlike a regular source, the `GenerateStream` method of a rewindable source must not return after it emits all tuples. Instead, it needs to wait until the `Rewind` method or the `Stop` method is called. Once it returns, the source is considered stopped and no further operation including the `REWIND SOURCE` statement wouldn't work on the source.

Due to its nature, a stream isn't often resumable. A resumable source is mostly used for relatively static data sources such as relations or files. Also, because implementing the `RewindableSource` interface is even harder than implementing the `Resumable` interface, utilities are usually used.

Utilities

There're some utilities to support implementing sources and its creators. This subsection describes each utility.

core.ImplementSourceStop

`core.ImplementSourceStop` is a function that implements the `Stop` method of a source in a thread-safe manner:

```
func ImplementSourceStop(s Source) Source
```

A source returned from this function is resumable, but not rewindable even if the original source implements the `core.RewindableSource` interface. In addition, although a source passed to `core.ImplementSourceStop` can explicitly implement the `core.Resumable` interface, its `Pause` and `Resume` method will never be called because the source returned from `core.ImplementSourceStop` also implements those methods and controls pause and resume.

To apply this function, a source must satisfy following restrictions:

1. The `GenerateStream` method must be implemented in a way that it can safely be called again after it returns.
2. The `GenerateStream` method must return when the `core.Writer.Write` returned `core.ErrSourceStopped`. The method must return exactly the same error variable that the writer returned.
3. The `Stop` method just returns nil.
    - This means all resource allocation and deallocation must be done within the `GenerateStream` method.

A typical implementation of a source passed to `core.ImplementSourceStop` is shown below:

```
func (s *MySource) GenerateStream(ctx *core.Context, w core.Writer) error {
    <initialization>
    defer func() {
        <clean up>
    }()

    for {
        t := <create a new tuple>
        if err := w.Write(ctx, t); err != nil {
            return err
        }
    }
    return nil
}

func (s *MySource) Stop(ctx *core.Context) error {
    return nil
}
```

If a source wants to ignore errors returned from `core.Writer.Write` other than `core.ErrSourceStopped`, the `GenerateStream` method can be modified as:

```
if err := w.Write(ctx, t); err != nil {
    if err == core.ErrSourceStopped {
        return err
    }
}
```

By applying `core.ImplementSourceStop`, the `Ticker` above can be implemented as follows:

```
type Ticker struct {
    interval time.Duration
}

func (t *Ticker) GenerateStream(ctx *core.Context, w core.Writer) error {
    var cnt int64
    for ; ; cnt++ {
        tuple := core.NewTuple(data.Map{"tick": data.Int(cnt)})
        if err := w.Write(ctx, tuple); err != nil {
            return err
        }
        time.Sleep(t.interval)
    }
    return nil
}

func (t *Ticker) Stop(ctx *core.Context) error {
```

```
    return nil
}

type TickerCreator struct {
}

func (t *TickerCreator) CreateSource(ctx *core.Context,
    ioParams *bql.IOParams, params data.Map) (core.Source, error) {
    interval := 1 * time.Second
    if v, ok := params["interval"]; ok {
        i, err := data.ToDuration(v)
        if err != nil {
            return nil, err
        }
        interval = i
    }
    return core.ImplementSourceStop(&Ticker{
        interval: interval,
    }), nil
}
```

There's no `stopped` flag now. In this version, the `Stop` method of the source returned by `core.ImplementSourceStop` waits until the `GenerateStream` method returns.

core.NewRewindableSource

`core.NewRewindableSource` is a function that converts a regular source into a rewindable source:

```
func NewRewindableSource(s Source) RewindableSource
```

A source returned from this function is resumable and rewindable. A source passed to the function needs to satisfy the same restrictions as `core.ImplementSourceStop`. In addition to that, there's one more restriction for `core.NewRewindableSource`:

4. The `GenerateStream` method must return when the `core.Writer.Write` returned `core.ErrSourceRewound`. The method must return exactly the same error variable that the writer returned.

Although the `GenerateStream` method of a rewindable source must not return after it emits all tuples, a source passed to the `core.NewRewindableSource` function needs to return in that situation. For example, let's assume there's a source that generate tuples from each line in a file. To implement the source without a help of the utility function, its `GenerateStream` must wait for the `Rewind` method to be called after it processes all lines in the file. However, with the utility, its `GenerateStream` can just return once it emits all tuples. Therefore, a typical implementation of a source passed to the utility can be same as a source for `core.ImplementSourceStop`.

As it will be shown later, a source that infinitely emits tuples can also be rewindable in some sense.

The following is an example of `TickerCreator` modified from the example for `core.ImplementSourceStop`:

```
func (t *TickerCreator) CreateSource(ctx *core.Context,
    ioParams *bql.IOParams, params data.Map) (core.Source, error) {
    interval := 1 * time.Second
    if v, ok := params["interval"]; ok {
        i, err := data.ToDuration(v)
        if err != nil {
            return nil, err
        }
```

```
        interval = i
    }

    rewindable := false
    if v, ok := params["rewindable"]; ok {
        b, err := data.AsBool(v)
        if err != nil {
            return nil, err
        }
        rewindable = b
    }

    src := &Ticker{
        interval: interval,
    }
    if rewindable {
        return core.NewRewindableSource(src), nil
    }
    return core.ImplementSourceStop(src), nil
}
```

In this example, `Ticker` has the `rewindable` parameter. If it is true, the source becomes rewindable:

```
CREATE SOURCE my_rewindable_ticker TYPE ticker WITH rewindable = true;
```

By issuing the `REWIND SOURCE` statement, `my_rewindable_ticker` resets the value of `tick` field:

```
REWIND SOURCE my_rewindable_ticker;

-- output examples of SELECT RSTREAM * FROM my_rewindable_ticker [RANGE 1 TUPLES];
{"tick":0}
{"tick":1}
{"tick":2}
...
{"tick":123}
-- REWIND SOURCE is executed here
{"tick":0}
{"tick":1}
...
```

bql.SourceCreatorFunc

`bql.SourceCreatorFunc` is a function that converts a function having the same signature as `SourceCreator.CreateSource` to a `SourceCreator`:

```
func SourceCreatorFunc(f func(*core.Context,
    *IOParams, data.Map) (core.Source, error)) SourceCreator
```

For example, `TickerCreator` above and its registration can be modified to as follows with this utility:

```
func CreateTicker(ctx *core.Context,
    ioParams *bql.IOParams, params data.Map) (core.Source, error) {
    interval := 1 * time.Second
    if v, ok := params["interval"]; ok {
        i, err := data.ToDuration(v)
        if err != nil {
```

```
            return nil, err
        }
        interval = i
    }
    return core.ImplementSourceStop(&Ticker{
        interval: interval,
    }), nil
}

func init() {
    bql.MustRegisterGlobalSourceCreator("ticker",
        bql.SourceCreatorFunc(CreateTicker))
}
```

A Complete Example

A complete example of `Ticker` is shown in this subsection. Assume that the import path of the example is `github.com/sensorbee/examples/ticker`, which doesn't actually exist. There're two files in the repository:

- ticker.go
- plugin/plugin.go

The example uses `core.NewRewindableSource` utility function.

ticker.go

```
package ticker

import (
    "time"

    "gopkg.in/sensorbee/sensorbee.v0/bql"
    "gopkg.in/sensorbee/sensorbee.v0/core"
    "gopkg.in/sensorbee/sensorbee.v0/data"
)

type Ticker struct {
    interval time.Duration
}

func (t *Ticker) GenerateStream(ctx *core.Context, w core.Writer) error {
    var cnt int64
    for ; ; cnt++ {
        tuple := core.NewTuple(data.Map{"tick": data.Int(cnt)})
        if err := w.Write(ctx, tuple); err != nil {
            return err
        }
        time.Sleep(t.interval)
    }
    return nil
}

func (t *Ticker) Stop(ctx *core.Context) error {
    // This method will be implemented by utility functions.
    return nil
```

```
}

type TickerCreator struct {
}

func (t *TickerCreator) CreateSource(ctx *core.Context,
    ioParams *bql.IOParams, params data.Map) (core.Source, error) {
    interval := 1 * time.Second
    if v, ok := params["interval"]; ok {
        i, err := data.ToDuration(v)
        if err != nil {
            return nil, err
        }
        interval = i
    }

    rewindable := false
    if v, ok := params["rewindable"]; ok {
        b, err := data.AsBool(v)
        if err != nil {
            return nil, err
        }
        rewindable = b
    }

    src := &Ticker{
        interval: interval,
    }
    if rewindable {
        return core.NewRewindableSource(src), nil
    }
    return core.ImplementSourceStop(src), nil
}
```

plugin/plugin.go

```
package plugin

import (
    "gopkg.in/sensorbee/sensorbee.v0/bql"

    "github.com/sensorbee/examples/ticker"
)

func init() {
    bql.MustRegisterGlobalSourceCreator("ticker", &ticker.TickerCreator{})
}
```

Sink Plugins

This section describes how to implement a sink as a plugin of SensorBee.

Implementing a Sink

A struct implementing the following interface can be a sink:

```
type Sink interface {
    Write(ctx *Context, t *Tuple) error
    Close(ctx *Context) error
}
```

This interface is defined in `gopkg.in/sensorbee/sensorbee.v0/core` package.

The `Write` method write a tuple to a destination of the sink. The argument `ctx` contains the information of the current processing context. `t` is the tuple to be written. The `Close` method is called when the sink becomes unnecessary. It must release all resources allocated for the sink.

The following example sink write a tuple as a JSON to stdout:

```
type StdoutSink struct {
}

func (s *StdoutSink) Write(ctx *core.Context, t *core.Tuple) error {
    _, err := fmt.Fprintln(os.Stdout, t.Data)
    return err
}

func (s *StdoutSink) Close(ctx *core.Context) error {
    // nothing to release
    return nil
}
```

A sink is initialized by its `SinkCreator`, which is described later.

Note: SensorBee doesn't provide buffering or retry capability for sinks.

Registering a Sink

To register a sink to the SensorBee server, the sink needs to provide its `SinkCreator`. The `SinkCreator` interface is defined in `gopkg.in/sensorbee/sensorbee.v0/bql` pacakge as follows:

```
// SinkCreator is an interface which creates instances of a Sink.
type SinkCreator interface {
    // CreateSink creates a new Sink instance using given parameters.
    CreateSink(ctx *core.Context, ioParams *IOParams, params data.Map) (core.Sink,␣
↪error)
}
```

It only has one method: `CreateSink`. The `CreateSink` method is called when the *CREATE SINK* statement is executed. The `ctx` argument contains the information of the current processing context. `ioParams` has the name and the type name of the sink, which are given in the `CREATE SINK` statement. `params` has parameters specified in the `WITH` clause of the `CREATE SINK` statement.

The creator can be registered by `RegisterGlobalSinkCreator` or `MustRegisterGlobalSinkCreator` function. As an example, the creator of `StdoutSink` above can be implemented and registered as follows:

```
type StdoutSinkCreator struct {
}

func (s *StdoutSinkCreator) CreateSink(ctx *core.Context,
    ioParams *bql.IOParams, params data.Map) (core.Sink, error) {
    return &StdoutSink{}, nil
}

func init() {
    bql.MustRegisterGlobalSinkCreator("my_stdout", &StdoutSinkCreator{})
}
```

This sink doesn't have parameters specified in the `WITH` clause of the `CREATE SINK` statement. How to handle parameters for sink is same as how source does. See *Source Plugins* for more details.

Utilities

There's one utility function for sink plugins: `SinkCreatorFunc`:

```
func SinkCreatorFunc(f func(*core.Context,
    *IOParams, data.Map) (core.Sink, error)) SinkCreator
```

This utility function is defined in `gopkg.in/sensorbee/sensorbee.v0/bql` pacakge. It converts a function having the same signature as `SinkCreator.CreateSink` to a `SinkCreator`. With this utility, for example, `StdoutSinkCreator` can be modified to:

```
func CreateStdoutSink(ctx *core.Context,
    ioParams *bql.IOParams, params data.Map) (core.Sink, error) {
    return &StdoutSink{}, nil
}

fucn init() {
    bql.MustRegisterGlobalSinkCreator("stdout",
        bql.SinkCreatorFunc(CreateStdoutSink))
}
```

A Complete Example

A complete example of the sink is shown in this subsection. The package name for the sink is `stdout` and `StdoutSink` is renamed to `Sink`. Also, this example uses `SinkCreatorFunc` utility for `SinkCreator`.

Assume that the import path of the example is `github.com/sensorbee/examples/stdout`, which doesn't actually exist. The repository has to files:

- stdout.go
- plugin/plugin.go

stdout.go

```
package stdout

import (
    "fmt"
```

```
    "os"

    "gopkg.in/sensorbee/sensorbee.v0/bql"
    "gopkg.in/sensorbee/sensorbee.v0/core"
    "gopkg.in/sensorbee/sensorbee.v0/data"
)

type Sink struct {
}

func (s *Sink) Write(ctx *core.Context, t *core.Tuple) error {
    _, err := fmt.Fprintln(os.Stdout, t.Data)
    return err
}

func (s *Sink) Close(ctx *core.Context) error {
    return nil
}

func Create(ctx *core.Context, ioParams *bql.IOParams, params data.Map) (core.Sink,␣
↪error) {
    return &Sink{}, nil
}
```

plugin/plugin.go

```
package plugin

import (
    "gopkg.in/sensorbee/sensorbee.v0/bql"

    "github.com/sensorbee/examples/stdout"
)

func init() {
    bql.MustRegisterGlobalSinkCreator("my_stdout",
        bql.SinkCreatorFunc(stdout.Create))
}
```

Part V

Reference

CHAPTER 14

BQL Statements

CREATE SINK

Synopsis

```
CREATE SINK name TYPE type_name [WITH parameter_name = parameter_value [, ...]]
```

Description

`CREATE SINK` creates a new sink in a topology.

Parameters

name The name of the sink to be created.

type_name The type name of the sink.

parameter_name The name of a sink-specific parameter.

parameter_value The value for a sink-specific parameter.

Sink Parameters

The optional `WITH` clause specifies parameters specific to the sink. See each sink's documentation to find out parameters it provides.

Examples

To create a sink having the name "snk" with no sink-specific parameter:

```
CREATE SINK snk TYPE fluentd;
```

To create a sink with sink-specific parameters:

```
CREATE SINK fluentd TYPE fluentd WITH tag_field = "fluentd_tag";
```

As you can see, the name of a sink can be same as the type name of a sink.

CREATE SOURCE

Synopsis

```
CREATE [PAUSED] SOURCE name TYPE type_name [WITH parameter_name = parameter_value [, .
↪..]]
```

Description

`CREATE SOURCE` creates a new source in a topology.

Parameters

`PAUSED` The source is paused when it's created if this option is used.

name The name of the source to be created.

type_name The type name of the source.

parameter_name The name of a source-specific parameter.

parameter_value The value for a source-specific parameter.

Source Parameters

The optional `WITH` clause specifies parameters specific to the source. See each source's documentation to find out parameters it provides.

Notes

Some sources stop after emitting all tuples. They can stop even before any subsequent statement is executed. For such sources, specify `PAUSED` parameter and run *RESUME SOURCE* after completely setting up a topology so that no tuple emitted from the source will be lost.

Examples

To create a source having the name "src" with no source-specific parameter:

```
CREATE SOURCE src TYPE dropped_tuples;
```

To create a source with source-specific parameters:

```
CREATE SOURCE fluentd TYPE fluentd WITH bind = "0.0.0.0:12345",
  tag_field = "my_tag";
```

As you can see, the name of a source can be same as the type name of a source.

CREATE STATE

Synopsis

```
CREATE STATE name TYPE type_name [WITH parameter_name = parameter_value [, ...]]
```

Description

`CREATE STATE` creates a new UDS (User Defined State) in a topology.

Parameters

name The name of the UDS to be created.

type_name The type name of the UDS.

parameter_name The name of a UDS-specific parameter.

parameter_value The value for a UDS-specific parameter.

UDS Parameters

The optional `WITH` clause specifies parameters specific to the UDS. See each UDS's documentation to find out parameters it provides.

Examples

To create a UDS named "my_uds" with no UDS-specific parameter:

```
CREATE STATE my_uds TYPE my_uds_type;
```

To create a UDS with UDS-specific parameters:

```
CREATE STATE my_ids TYPE snowflake_id WITH machine_id = 1;
```

CREATE STREAM

Synopsis

```
CREATE STREAM name AS select
```

Description

`CREATE STREAM` creates a new stream (a.k.a a continuous view) in a topology.

Parameters

name The name of the stream to be created.

select The `SELECT` statement to generate a stream. **select** can be any `SELECT` statement including a statement using `UNION ALL`.

Examples

To create a stream named "strm":

```
CREATE STREAM strm AS SELECT RSTREAM * FROM src [RANGE 1 TUPLES];
```

To create a stream which merges all tuples from multiple streams:

```
CREATE STREAM strm AS
    SELECT RSTREAM * FROM src1 [RANGE 1 TUPLES]
    UNION ALL
    SELECT RSTREAM * FROM src2 [RANGE 1 TUPLES];
```

DROP SINK

Synopsis

```
DROP SINK name
```

Description

`DROP SINK` drops a sink that is already created in a topology. The sink can no longer be used after executing the statement.

Parameters

name The name of the sink to be dropped.

Examples

To drop a sink having the name "snk":

```
DROP SINK snk;
```

DROP SOURCE

Synopsis

```
DROP SOURCE name
```

Description

`DROP SOURCE` drops a source that is already created in a topology. The source is stopped and removed from a topology. After executing the statement, the source cannot be used.

Parameters

name The name of the source to be dropped.

Examples

To drop a source having the name "src":

```
DROP SOURCE src;
```

DROP STATE

Synopsis

```
DROP STATE name
```

Description

`DROP STATE` drops a UDS that is already created in a topology. The UDS can no longer be used after executing the statement.

Note: Even if a `uds` sink exist for the UDS, the sink will not be dropped when dropping the UDS. The `uds` sink must be dropped manually.

Parameters

name The name of the UDS to be dropped.

Examples

To drop a UDS named "my_uds":

```
DROP STATE my_ds;
```

DROP STREAM

Synopsis

```
DROP STREAM name
```

Description

`DROP STREAM` drops a stream that is already created in a topology. The stream can no longer be used after executing the statement.

Parameters

name The name of the stream to be dropped.

Examples

To drop a stream having the name "strm":

```
DROP STREAM strm;
```

INSERT INTO

Synopsis

```
INSERT INTO sink FROM stream
```

Description

`INSERT INTO` inserts tuples from a stream or a source to a sink.

Parameters

sink The name of the sink to which tuples are inserted.

stream The name of a stream or a source.

Examples

To insert tuples into a sink from a source having the name "src":

```
INSERT INTO snk FROM src;
```

LOAD STATE

Synopsis

```
LOAD STATE name TYPE type_name [TAG tag]
    [SET set_parameter_name = set_parameter_key]
    [create_if_not_saved]

where create_if_not_saved is:

    OR CREATE IF NOT SAVED
        [WITH create_parameter_name = create_parameter_value]
```

Description

`LOAD STATE` loads a UDS that is previously saved by *SAVE STATE*.

`LOAD STATE` fails if the UDS hasn't been saved yet. When `OR CREATE IF NOT SAVED` is specified, `LOAD STATE` creates a new UDS with the given optional parameters if the UDS hasn't been saved yet.

`LOAD STATE`, even with `OR CREATE IF NOT SAVED`, fails if the UDS doesn't support the statement.

Parameters

name The name of the UDS to be loaded.

type_name The type name of the UDS.

tag The name of the user defined tag for versioning of the saved UDS data. When **tag** is omitted, "default" is used as the default tag name.

set_parameter_name The name of a UDS-specific parameter defied for `LOAD STATE`.

set_parameter_value The value for a UDS-specific parameter defined for `LOAD STATE`.

create_parameter_name The name of a UDS-specific parameter defined for `CREATE STATE`.

create_parameter_value The value for a UDS-specific parameter defined for `CREATE STATE`.

`LOAD STATE` can have two sets of parameters: **set_parameters** and **create_parameters**. **set_parameters** are used when there's a saved UDS data having the given tag. On the other hand, **create_parameters** are used when the UDS hasn't been saved yet. **create_parameters** are exactly same as parameters that the UDS defines for *CREATE STATE*. However, **set_parameters** are often completely different from **create_parameters**. Because **create_parameters** are often saved as a part of the UDS's information by `SAVE STATE`, **set_parameters** doesn't have to have the same set of parameters defined in **create_parameters**.

There're some use-cases that a UDS uses **set_parameters**:

- Customize loading behavior

  - When a UDS doesn't provide a proper versioning of saved data, `LOAD STATE` may fail to load it due to the format incompatibility. In such a case, it's difficult to modify saved binary data to have a format version number. Thus, providing a **set_parameter** specifying the format version number could be the only solution.

- Overwrite some saved values of **create_parameters**

Like **create_parameters**, **set_parameters** are specific to each UDS. See the documentation of each UDS to find out parameters it provides.

Examples

To load a UDS named "my_uds" and having the type "my_uds_type":

```
LOAD STATE my_uds TYPE my_uds_type;
```

Note that "my_uds" needs to be saved before executing this statement. Otherwise, it fails.

To load a UDS named "my_uds" and having the type "my_uds_type" and assigned a tag "trained":

```
LOAD STATE my_uds TYPE my_uds_type TAG trained;
```

To load a UDS with **set_parameters**:

```
LOAD STATE my_uds TYPE my_uds_type TAG trained
    SET force_format_version = "v1";
```

To load a UDS that hasn't been saved yet with `OR CREATE IF NOT SAVED`:

```
LOAD STATE my_uds TYPE my_uds_type OR CREATE IF NOT SAVED;
```

To load a UDS that hasn't been saved yet with `OR CREATE IF NOT SAVED` with **create_parameters**:

```
LOAD STATE my_uds TYPE my_uds_type OR CREATE IF NOT SAVED
    WITH id = 1;
```

When the UDS hasn't been saved previously, the statement above falls back into `CREATE STATE` as follows:

```
CREATE STATE my_uds TYPE my_uds_type WITH id = 1;
```

`OR CREATE IF NOT SAVED` can be used with a tag and **set_parameters**:

```
LOAD STATE my_uds TYPE my_uds_type TAG trained SET force_format_version = "v1"
    OR CREATE IF NOT SAVED WITH id = 1;
```

PAUSE SOURCE

Synopsis

```
PAUSE SOURCE name
```

Description

`PAUSE SOURCE` pauses a running source so that the source stops emitting tuples until executing *RESUME SOURCE* on it again. Executing `PAUSE SOURCE` on a paused source doesn't affect anything.

`PAUSE SOURCE` fails if the source doesn't support the statement.

Parameters

name The name of the source to be paused.

Examples

To pause a source named "src":

```
PAUSE SOURCE src;
```

RESUME SOURCE

Synopsis

```
RESUME SOURCE name
```

Description

`RESUME SOURCE` resumes a paused source so that the source can start to emit tuples again. Executing `RESUME SOURCE` on a running source doesn't affect anything.

`RESUME SOURCE` fails if the source doesn't support the statement.

Parameters

name The name of the source to be resumed.

Examples

A common use case of `RESUME SOURCE` to resume a source which is created by `CREATE PAUSED SOURCE`.

```
CREATE PAUSED SOURCE src TYPE some_source_type WITH ...parameters...;

-- ... construct a topology connected to src ...

RESUME SOURCE src;
```

By doing this, no tuple emitted from *src* will be lost.

REWIND SOURCE

Synopsis

```
REWIND SOURCE name
```

Description

`REWIND SOURCE` rewinds a source so that the source emits tuples from the beginning again.

`REWIND SOURCE` fails if the source doesn't support the statement.

Parameters

name The name of the source to be rewound.

Examples

To rewind a source named "src":

```
REWIND SOURCE src;
```

SAVE STATE

Synopsis

```
SAVE STATE name [TAG tag]
```

Description

`SAVE STATE` saves a UDS to SensorBee's storage. The location or the format of the saved UDS depends on a storage that SensorBee uses and is not controllable from BQL.

`SAVE STATE` fails if the UDS doesn't support the statement.

Parameters

name The name of the UDS to be saved.

tag The name of the user defined tag for versioning of the saved UDS data. When **tag** is omitted, "default" is used as the default tag name.

Examples

To save a UDS named "my_uds" without a tag:

```
SAVE STATE my_uds;
```

To save a UDS with a tag "trained":

```
SAVE STATE my_uds TAG trained;
```

SELECT

Synopsis

```
SELECT emitter {* | expression [AS output_name]} [, ...]
    FROM from_item stream_to_relation_operator
        [AS stream_alias] [, ...]
    [WHERE condition [, ...]]
    [GROUP BY expression [, ...]]
    [HAVING condition [, ...]]
    [UNION ALL select]

where emitter is:

    {RSTREAM | ISTREAM | DSTREAM}

where stream_to_relation_operator is:

    '[' RANGE range_number {TUPLES | SECONDS | MILLISECONDS}
        [, BUFFER SIZE buffer_size]
        [, drop_mode IF FULL]
    ']'
```

Description

`SELECT` retrieves tuples from one or more streams. The general processing of `SELECT` is as follows:

1. Each **from_item** is converted into a relation (i.e. a window) from a stream. Then, each tuple emitted from a **from_item** is computed within the window. If more than one element is specified in the `FROM` clause, they are cross-joined.
2. When the `WHERE` clause is specified, **condition** is evaluated for each set of tuples cross-joined in the `FROM` clause. Tuples which do not satisfy the condition are eliminated from the output.
3. If the `GROUP BY` clause is specified, tuples satisfied the condition in the `WHERE` clause are combined into groups based on the result of expressions. Then, aggregate functions are performed on each group. When the `HAVING` clause is given, it eliminates groups that do not satisfy the given condition.
4. The output tuples are computed using the `SELECT` output expressions for each tuple or group.
5. Computed output tuples are converted into a stream using *Relation-to-Stream Operators* and emitted from the `SELECT`.
6. `UNION ALL`, if present, combines outputs from multiple `SELECT`. It simply emits all tuples from all `SELECT` without considering duplicates.

Parameters

Emitter

emitter controls how a `SELECT` emits resulting tuples.

`RSTREAM` When `RSTREAM` is specified, all tuples in a relation as a result of processing a newly coming tuple are output. See *Relation-to-Stream Operators* for more details.

`ISTREAM` When `ISTREAM` is specified, tuples contained in the current relation but not in the previously computed relation are emitted. In other words, tuples that are newly inserted or updated since the previous computation are output. See *Relation-to-Stream Operators* for more details.

`DSTREAM` When `DSTREAM` is specified, tuples contained in the previously computed relation but not in the current relation are emitted. In other words, tuples in the previous relation that are deleted or updated in the current relation are output. Note that output tuples are from the previous relation so that they have old values. See *Relation-to-Stream Operators* for more details.

`FROM` Clause

The `FROM` clause specifies one or more source streams for the `SELECT` and converts those streams into relations using stream to relation operators.

The `FROM` clause contains following parameters:

from_item **from_item** is a source stream which can be either a source, a stream, or a UDSF.

range_number **range_number** is a numeric value which specifies how many tuples are in the window. **range_number** is followed by one of interval types: `TUPLES`, `SECONDS`, or `MILLISECONDS`.

When `TUPLES` is given, **range_number** must be an positive integer and the window can contain at most **range_number** tuples. If a new tuple is inserted into the window having **range_number** tuples, the oldest tuple is removed. "The oldest tuple" is the tuple that was inserted into the window before any other tuples, not the tuple having the oldest timestamp. The maximum **range_number** is 1048575 with `TUPLES` keywords.

When `SECONDS` or `MILLISECONDS` is specified, **range_number** can be a positive number and the difference of the minimum and maximum timestamps of tuples in the window can be at most **range_number** seconds or milliseconds. If a new tuple is inserted into the window, tuples whose timestamp is **range_number** seconds or milliseconds earlier than the new tuple's timestamp are removed. The maximum **range_number** is 86400 with `SECONDS` and 86400000 with `MILLISECONDS`.

buffer_size **buffer_size** specifies the size of buffer, or a queue, located between **from_item** and the `SELECT`. **buffer_size** must be an integer and greater than 0. The meximum **buffer_size** is 131071.

drop_mode **drop_mode** controls how a new tuple is inserted into the buffer located between **from_item** and the `SELECT` when the buffer is full. **drop_mode** can be one of the followings:

- `WAIT`
  - A new tuple emitted from **from_item** is blocked until the `SELECT` consumes at least one tuple.
- `DROP OLDEST`
  - The oldest tuple in the buffer is removed and a new tuple is inserted into the buffer. "The oldest tuple" is the tuple that was inserted into the buffer before any other tuples, not the tuple having the oldest timestamp.
- `DROP NEWEST`

- The oldest tuple in the buffer is removed and a new tuple is inserted into the buffer. "The newest tuple" is the tuple that was inserted into the buffer after any other tuples, not the tuple having the newest timestamp.

Note: A buffer is different from a window. A buffer is placed in front of a window. A window gets a new tuple from a buffer and computes a new relation. A buffer is used not to block emitting tuples so that multiple `SELECT` statements can work concurrently without waiting for their receivers to consume tuples.

---

stream_alias **stream_alias** provides an alias of **from_item** and it can be referred by the alias in other parts of the `SELECT`. If the alias is given, the original name is hidden and cannot be used to refer **from_item**.

Fields of tuples can be referred by `<field_name>` or `<stream>:<field_name>` in other clauses and the `SELECT` list. For example, when the `SELECT` has `FROM strm [RANGE 1 TUPLES]` and `strm` emits `{"a":<some value>}`, the field `a` can be referred by `a` or `strm:a`. These two forms cannot be mixed in a `SELECT` statement.

The form `<stream>:<field_name>` is required when the `FROM` clause has multiple input streams.

`WHERE` Clause

The `SELECT` can optionally have a `WHERE` clause. The `WHERE` clause have a condition. The condition can be any expression that evaluates to a result of type `bool`. Any tuple that does not satisfy the condition (i.e. the result of the expression is `false`) will be eliminated from the output.

Operators describes operators that can be used in the condition.

`GROUP BY` Clause

The `GROUP BY` clause is an optional clause and condenses into a single tuple all selected tuples whose expressions specified in `GROUP BY` clause result in the same value.

expression can be any expression using fields of an input tuple. When there're multiple expressions in the clause, tuples having the same set of values computed from those expressions are grouped into a single tuple.

When the `GROUP BY` clause is present, any ungrouped field cannot be used as an output field without aggregate functions. For example, when tuples have 4 fields `a`, `b`, `c`, and `d`, and the `GROUP BY` clause has following expressions:

```
GROUP BY a, b + c
```

`a` can only be used as an output field:

```
SELECT a FROM stream [RANGE 1 TUPLES]
GROUP BY a, b + c;
```

Other fields need to be specified in aggregate functions:

```
SELECT a, max(b), min(b + c), avg(c * d) FROM stream [RANGE 1 TUPLES]
GROUP BY a, b + c;
```

Aggregate functions are evaluated for each group using all tuples in the group.

Note: The `GROUP BY` clause performs grouping within a window:

```
SELECT a FROM stream [RANGE 10 TUPLES]
GROUP BY a;
```

This SELECT computes at most 10 groups of tuples because there're only 10 tuples in the window.

HAVING Clause

The HAVING clause is an optional clause and placed after the GROUP BY clause. The HAVING clause has an condition and evaluate it for each group, instead of each tuple. When ungrouped fields are used in the condition, they need to be in aggregate functions:

```
SELECT a, max(b), min(b + c), avg(c * d) FROM stream [RANGE 1 TUPLES]
GROUP BY a, b + c HAVING min(b + c) > 1 AND avg(c * d) < 10;
```

In this example, b, c, and d are ungrouped fields and cannot directly specified in the condition.

SELECT List

The SELECT list, placed between the **emitter** and the FROM clause, defines the form of the output tuples emitted from the SELECT statement.

Each item in the list can be any expression. Each item (i.e. output field) will have a name. When an expression only consists of a field name, the output name of the expression will be the field name. For example, the output name of strm:price in SELECT RSTREAM strm:price FROM ... will be price, not strm:price. When the expression is a UDF call, the name of the UDF will be used as the name of the output field. For example, the result of count(*) is named as count. If an expression is other than a field name or a UDF call, the output name will be col_n where n is replaced with the number corresponding to n-th expression (counting from 0). The output field name can manually be specified by AS output_name.

When the expression is *, all fields which have not been specified in the SELECT list yet will be included. Output names of those fields will be identical to the original field names.

If an expression results in a map, its output name can be AS *. In such case, all fields of the map is extended to the top level fields. For example, in SELECT RSTREAM a, b AS *, c FROM strm ..., when strm emits tuples having

```
{
    "a": v1,
    "b": {
        "d": v3,
        "e": v4
    },
    "c": v2,
}
```

to the SELECT, its output will be

```
{
    "a": v1,
    "c": v2,
    "d": v3,
    "e": v4
}
```

When some fields have the same name, only one of them will be included in the result. It is undefined which field will be chosen as a result.

Notes

An Emitter and Its Performance

There're some use case specific optimizations of the evaluation of the `SELECT` and this subsection describes each optimization and its limitation.

Simple Transformation and Filtering

Performing a simple per-tuple transformation or filtering over an input stream is a very common task. Therefore, BQL optimizes statements having the following form:

```
SELECT RSTREAM projection FROM input [RANGE 1 TUPLES] WHERE condition;
```

Limitations of this optimization are:

- There can only be one input stream and its range is `[RANGE 1 TUPLES]`.
- The emitter must be `RSTREAM`.

Evaluation in `WHERE` Clause

Each set of tuples cross-joined in the `FROM` clause is evaluated exactly once in the `WHERE` clause. Therefore, all functions in the `WHERE` clause are only called once for each set:

```
SELECT RSTREAM * FROM stream1 [RANGE 100 TUPLES], stream2 [RANGE 100 TUPLES]
    WHERE random() < 0.2;
```

In this example, 80% of sets of cross-joined tuples are filtered out and only 20% of sets (around 20 tuples for each input from either stream) are emitted.

CHAPTER 15

Commands

build_sensorbee

Because SensorBee is written in Go, all its dependencies including plugins generally need to be statically linked, or at least cannot be dynamically loeaded at runtime. The `build_sensorbee` command is provided to support building a custom `sensorbee` command.

Basic Usage

Prepare `build.yaml` configuration file and run `build_sensorbee` in the same directory as `build.yaml` is located at:

```
$ ls
build.yaml
$ build_sensorbee
sensorbee_main.go
$ ls
build.yaml
sensorbee
sensorbee_main.go
```

`sensorbee` is the result executable file and `sensorbee_main.go` is a Go file generated by `build_sensorbee` and passed for `go build` command to build `sensorbee`.

Configuration

`build_sensorbee` requires a configuration file named `build.yaml`. The file is written in YAML and has following optional sections:

- `plugins`
- `commands`

plugins

The `plugins` section is optional and may have a list of plugins as follows:

```
plugins:
  - github.com/sensorbee/twitter/plugin
  - github.com/sensorbee/fluentd/plugin
  - github.com/sensorbee/nlp/plugin
  - github.com/sensorbee/tutorial/ml/plugin
  - github.com/sensorbee/jubatus/classifier/plugin
```

A plugin must be provided as a valid import path of Go. A path depends on each plugin.

commands

The `commands` section is optional and is used to customize subcommands that the `sensorbee` command will have. By default, or when the section is empty, subcommands include all standard commands:

- `run`
- `runfile`
- `shell`
- `topology`

`commands` section is a map whose key is the name of subcommand. Standard subcommands like `run` can be added by providing empty entries:

```
commands:
  run:
  shell:
```

With this configuration, the `sensorbee` command will only have `run` and `shell` commands.

To add a custom command, an entry must have `path` parameter that is a Go import path of the command:

```
commands:
  run:
  shell:
  mytest:
    path: "path/to/sb-mytest"
```

With this configuration, the `sensorbee` command will also have the `mytest` subcommand. The subcommand is implemented at `path/to/sb-mytest`.

Names of commands must be unique and cannot be any of:

- `cli`
- `os`
- `version`
- `time`

Prohibited names might be added in the future version.

Custom Subcommand Development

A custom subcommand for the `sensorbee` command can be developed as a Go package. Only thing the package has to do is to provide a function `func SetUp() cli.Command`. `cli` is `gopkg.in/urfave/cli.v1`. A minimum example is provided in the SensorBee tutorial repository:

```
package hello

import (
    "fmt"

    cli "gopkg.in/urfave/cli.v1"
)

func SetUp() cli.Command {
    return cli.Command{
        Name:  "hello",
        Usage: "say hello",
        Action: func(c *cli.Context) error {
            fmt.Println("hello")
            return nil
        },
    }
}
```

This command prints "hello" when *sensorbee hello* is executed. See https://github.com/urfave/cli to learn how to create a command using the `cli` library.

A Complete Example

```
plugins:
  - github.com/sensorbee/twitter/plugin
  - github.com/sensorbee/fluentd/plugin
  - github.com/sensorbee/nlp/plugin
  - github.com/sensorbee/tutorial/ml/plugin
  - github.com/sensorbee/jubatus/classifier/plugin

commands:
  run:
  runfile:
  shell:
  topology:
  hello:
    path: "github.com/sensorbee/tutorial/custom_command/hello"
```

Flags and Options

`--config path` or `-c path`

This option specifies the path to the configuration file to be used. Its default value is `build.yaml`. With this option, a configuration file in another directory can be used as follows:

```
$ build_sensorbee -c /path/to/dir/special_build.yaml
```

`--download-plugins={true|false}`

This option have to be `true` or `false`. When the value is `true`, `build_sensorbee` downloads (i.e. `go get`) all plugins listed in `build.yaml`. When it's `false`, `build_sensorbee` doesn't download plugins and tries to used plugins as installed in the environment. The default value is `true`.

Specifying `false` is useful when the custom `sensorbee` command needs to depend on a plugin that is in a special git branch or locally modified.

`--help` or `-h`

When this flag is given, the command shows the usage of itself and exits without doing anything.

`--only-generate-source`

When this flag is given, `build_sensorbee` doesn't build a `sensorbee` command but only generate a source code that can be built by `go build` command. For example:

```
$ build_sensorbee --only-generate-source
sensorbee_main.go
$ go build -o sensorbee sensorbee_main.go
```

`--out executable_name` or `-o executable_name`

This option customizes the name of the output executable file. The default is `sensorbee.exe` in Windows and `sensorbee` in all other environment. The following command generates an executable named `my_sensorbee` instead of `sensorbee`:

```
$ build_sensorbee -o my_sensorbee
```

`--source-filename filename`

The filename of the Go source code file automatically generated by `build_sensorbee` can be speicfied by this option. The default value is `sensorbee_main.go`.

```
$ build_sensorbee --source-filename custom_main.go
```

By executing this command, `custom_main.go` is generated instead of `sensorbee_main.go`.

`--version` or `-v`

When this flag is given, the command prints the version of the `build_sensorbee` command.

sensorbee

`sensorbee` is the main command to manipulate SensorBee. `sensorbee` consists of a set of following subcommands:

- *sensorbee run*
- *sensorbee runfile*
- *sensorbee shell or bql*
- *sensorbee topology*

`sensorbee` command can needs to be created by *build_sensorbee* command and all the example commands are written as `./sensorbee` to emphasize that there's no default `sensorbee` command.

See each command's reference for details.

Flags and Options

`--help` or `-h`

When this flag is given, the command prints the usage of itself.

`--version` or `-v`

The command prints the version of SensorBee.

sensorbee run

`sensorbee run` runs the SensorBee server that manages multiple topologies that can dynamically modified at runtime.

Basic Usage

```
$ ./sensorbee run -c sensorbee.yaml
```

Configuration

A configuration file can optionally be provided for `sensorbee run` command. The file is written in YAML and has following optional sections:

- `logging`
- `network`
- `storage`
- `topologies`

logging

The `logging` section customizes behavior of the logger. It has following optional parameters:

`target`

The `target` parameter changes the destination of log messages. Its value can be of followings:

- `stdout`: write log messages to stdout
- `stderr`: write log messages to stderr
- file path: a path to the log file

When the value is neither `stdout` nor `stderr`, it's considered to be a file path. The default value of this parameter is `stderr`.

`min_log_level`

This option specifies the minimum level (severity) of log messages to be written. Valid values are one of `debug`, `info`, `warn`, `warning`, `error`, or `fatal`. `warn` can also be `warning`. When `debug` is given, all levels of messages will be written into the log. When the value is `error`, only log messages with `error` or `fatal` level will be written. The default value of this parameter is `info`.

`log_dropped_tuples`

The SensorBee server can prints a log message and contents of tuples when they're dropped from a topology. When this option is `true`, the server writes log messages reporting dropped tuples. When it's `false`, the server doesn't. The default value of this option is `false`.

`log_destinationless_tuples`

A destinationless tuple is one kind of dropped tuples that is caused when a source or a stream doesn't have any destination and it drops a tuple. By setting `true` to this option, the server reports all destinationless tuples. The default value of this option is `false`. Note that, to log replication tuples, the `log_dropped_tuples` option also needs to be `true`.

`summarize_dropped_tuples`

This option turns on or off summarization of dropped tuple logging. Valid values for this option is `true` or `false`. When its value is the `true`, dropped tuples are summarized in log messages.

Note: At the current version, only `blob` fields are summarized to `"(blob)"`. Other configuration parameters will be supported in the future version such as the maximum number of fields, the maximum depths of maps, the maximum length of arrays, and so on.

When `false` is specified, each log message shows a complete JSON that are compatible to the original tuple. Although this is useful for debugging, tuples containing large binary data like images may result in disk.

The default value of this option is `false`.

Example:

```
logging:
  target: /path/to/sensorbee.log
  min_log_level: info
  log_dropped_tuples: true
  summarize_dropped_tuples: true
```

network

The `network` section has parameters related to server's network configuration. It has following optional parameters:

`listen_on`

This parameter controls how the server expose its listening port. The syntax of the value is like `host:port`. `host` can be IP addresses such as `0.0.0.0` or `127.0.0.1`. When `host` is given, the server only listens on the interface with the given host address. If the `host` is omitted, the server listens on all available interfaces, that is, the server accepts connections from any host. The default value of this parameter is `:15601`.

Example:

```
network:
  listen_on: ":15601"
```

storage

The `storage` section contains the configuration of storages used for saving UDSs or other information. It has following optional subsections:

- `uds`

uds

The `uds` subsection configures the storage for saving and loading UDSs. It provides following optional parameters:

`type`

The type name of the storage. `in_memory` is used as the default value.

`params`

`params` has subparameter specific to the given `type`.

Currently, following types are available:

- `in_memory`
- `fs`

Descriptions of types and parameters are provided below:

`in_memory`

`in_memory` saves UDSs in memory. It loses all saved data when the server restarts. This type doesn't have any parameter.

Example:

```
storage:
  uds:
    type: in_memory
```

`fs`

`fs` saves UDSs in the local file system. It has following required parameters:

`dir`

`dir` has the path to the directory that saved data will be stored.

`fs` also has following optional parameters:

`temp_dir`

`temp_dir` has the path to the temporary directory that is used when the UDS writes data. After the UDS has written all the data, the file is move to the directory specified by `dir` parameter. The same value as `dir` is used by default.

The file name of each saved UDS is formatted as `<topology>-<name>-<tag>.state`.

Example:

```
storage:
  uds:
    type: fs
    params:
      dir: /path/to/uds_dir
      temp_dir: /tmp
```

topologies

The `topologies` section contains the configuration of topologies in the following format:

```
topologies:
  name_of_topology1:
    ... configuration for name_of_topology1 ...
  name_of_topology2:
    ... configuration for name_of_topology2 ...
  name_of_topology3:
    ... configuration for name_of_topology3 ...
  ... other topologies ...
```

Topologies listed in this section will be created at the startup of the server based on the sub-configuration of each topology. Following optional configuration parameters are provided for each topology:

`bql_file`

> This parameter has the path to the file containing BQL statements for the topology. All statements are executed before the server gets ready. If the execution fails, the server would exit with an error.

Example:

```
$ ls
my_topology.bql
sensorbee.yaml
$ cat my_topology.bql
CREATE SOURCE fluentd TYPE fluentd;
CREATE STREAM users AS
    SELECT RSTREAM users FROM fluentd [RANGE 1 TUPLES];
CREATE SINK user_file TYPE file WITH path = "users.jsonl";
$ cat sensorbee.yaml
topologies:
  my_topology:
    bql_file: my_topology.bql
$ ./sensorbee run -c sensorbee.yaml
```

As a result of these commands above, the server started with `sensorbee.yaml` has a topology named `my_topology`. The topology has three nodes: `fluentd`, `users`, and `user_file`.

Note: This is the only way to persist the configuration of topologies at the moment. Any updates applied at runtime will not be reflected into the bql file. For example, if the server restarts after creating a new stream in `my_topology`, the new stream will be lost unless it's explicitly added to `my_topology.bql` manually.

The configuration of a topology can be empty:

```
topologies:
  my_empty_topology:
```

In this case, an empty topology `my_empty_topology` will be created so that the `sensorbee topology create` command doesn't have to be executed every time the server restarts.

A Complete Example

```
logging:
  target: /path/to/sensorbee.log
  min_log_level: info
  log_dropped_tuples: true
  summarize_dropped_tuples: true
```

```
network:
  listen_on: ":15601"

storage:
  uds:
    type: fs
    params:
      dir: /path/to/uds_dir
      temp_dir: /tmp

topologies:
  empty_topology:
  my_topology:
    bql_file: /path/to/my_topology.bql
```

Flags and Options

`--config path` or `-c path`

This option receives the path of the configuration file. By default, the value is empty and no configuration file is used. This value can also be passed through `SENSORBEE_CONFIG` environment variable.

`--help` or `-h`

When this flag is given, the command prints the usage of itself.

sensorbee runfile

`sensorbee runfile` runs a single BQL file. This command is mainly designed for offline data processing but can be used as a standalone SensorBee process that doesn't expose any interface to manipulate the topology.

`sensorbee runfile` stops after all the nodes created by the given BQL file stops. The command doesn't stop if it contains a source that generates infinite tuples or is rewindable. Other non-rewindable sources such as `file` stopping when it emits all tuples written in a file can work well with the command.

Sources generally need to be created with `PAUSED` keyword in the *CREATE SOURCE* statement. Without `PAUSED`, a source can start emitting tuples before all nodes in a topology can correctly be set up. Therefore, a BQL file passed to the command should look like:

```
CREATE PAUSED SOURCE source_1 TYPE ...;
CREATE PAUSED SOURCE source_2 TYPE ...;
...
CREATE PAUSED SOURCE source_n TYPE ...;

... CREATE STREAM, CREATE SINK, or other statements

RESUME SOURCE source_1;
RESUME SOURCE source_2;
...
RESUME SOURCE source_n;
```

With the `--save-uds` option described later, it saves UDSs at the end of its execution.

Basic Usage

```
$ ./sensorbee runfile my_topology.bql
```

With options:

```
$ ./sensorbee runfile -c sensorbee.yaml -s '' my_topology.bql
```

Configuration

`sensorbee runfile` accepts the configuration file for `sensorbee run`. It only uses `logging` and `storage` sections. The configuration file may contain other sections as well and the same file for `sensorbee run` can also be used for `sensorbee runfile`. See *its configuration* for details.

Flags and Options

`--config path` or `-c path`

This option receives the path of the configuration file. By default, the value is empty and no configuration file is used. This value can also be passed through `SENSORBEE_CONFIG` environment variable.

`--help` or `-h`

When this flag is given, the command prints the usage of itself.

`--save-uds udss` or `-s udss`

This option receives a list of names of UDSs separated by commas. UDSs listed in it will be saved at the end of execution. For example, when the option is `-s "a,b,c"`, UDSs named `a`, `b`, and `c` will be saved. To save all UDSs in a topology, pass an empty string: `-s ""`.

By default, all UDSs will **not** be saved at the end of execution.

`--topology name` or `-t name`

This option changes the name of the topology to be run with the given BQL file. The default name is taken from the file name of the BQL file. The name specified to this option will be used in log messages or saved UDS data. Especially, names of files containing saved UDS data has contains the name of the topology. Therefore, providing the same name as the topology that will be run by `sensorbee run` later on allows users to prepare UDSs including pre-trained machine learning models in advance.

sensorbee shell or bql

`sensorbee shell` or `bql` starts a new shell to manipulate the SensorBee server. The shell can be terminated by writing `exit` or typing `C-d`.

Both `sensorbee shell` and `bql` have the same interface, but `bql` is installed by default while the `sensorbee` command needs to be built manually to run `sensorbee shell`.

Basic Usage

To run `sensorbee shell`,

```
$ ./sensorbee shell -t my_topology
my_topology>
```

To run `bql`,

```
$ bql -t my_topology
my_topology>
```

Flags and options

`--api-version version`

This option changes the API version of the SensorBee server. The default value of this option is `v1`.

`--help` or `-h`

When this flag is given, the command prints the usage of itself.

`--topology name` or `-t name`

The name of a topology to be manipulated can be specified through this option so that `USE topology_name` doesn't have to be used in the shell. The default value is an empty name, that is, no topology is specified.

`--uri`

This option is used when the SensorBee server is running at non-localhost or using non-default port number (15601). The value should have a format like `http://host:port/`. The default value of this option is `http://localhost:15601/`.

sensorbee topology

`sensorbee topology`, or `sensorbee t`, is used to manipulate topologies on the SensorBee server.

Note: This command is provided because the syntax of BQL statements that controls topologies has not been discussed enough yet.

The command consists of following subcommands:

`sensorbee topology create <name>` or `sensorbee t c <name>`

This command creates a new topology on the SensorBee server. The `<name>` argument is the name of the topology to be created. `$?` will be 0 if the command is successful. Otherwise, it'll be non-zero. The command fails if the topology already exists on the server.

`sensorbee topology drop <name>` or `sensorbee t drop <name>`

This command drops an existing topology on the SensorBee server. The `<name>` argument is the name of the topology to be dropped. `$?` will be 0 if the command is successful. Otherwise, it'll be non-zero. The command doesn't fail even if the topology doesn't exist on the server.

`sensorbee topology list` or `sensorbee t l`

This commands prints names of all topologies that the SensorBee server has, one name per line.

All commands share the same flags and options. Flags and options need to be given after the subcommand name:

```
$ ./sensorbee topology create --flag --option value my_topology
```

In this example, a flag `--flag` and an option `--option value` are provided. The argument of the command, i.e. the name of topology, is `my_topology`.

Flags and Options

`--api-version version`

This option changes the API version of the SensorBee server. The default value of this option is `v1`.

`--help` or `-h`

When this flag is given, the command prints the usage of itself.

`--uri`

This option is used when the SensorBee server is running at non-localhost or using non-default port number (15601). The value should have a format like `http://host:port/`. The default value of this option is `http://localhost:15601/`.

CHAPTER 16

Function Reference

Common Mathematical Functions

For the functions below, if a given parameter is outside the mathematically valid range for that function (e.g., `sqrt(-2)`, `log(0)`, `div(2.0, 0.0)`) and the return type is `float`, then `NaN` is returned. However, if the return type is `int` (e.g., `div(2, 0)`), there is no `NaN` option and an error will occur instead.

abs

```
abs(x)
```

Description

`abs` computes the absolute value of a number.

Parameter Types

x `int` or `float`

Return Type

same as input

Examples

Function Call	Result
`abs(-17.4)`	`17.4`

cbrt

```
cbrt(x)
```

Description

`cbrt` computes the cube root of a number.

Parameter Types

x `int` or `float`

Return Type

`float`

Examples

Function Call	Result
`cbrt(27.0)`	`3.0`
`cbrt(-3)`	`-1.4422495703074083`

ceil

```
ceil(x)
```

Description

`ceil` computes the smallest integer not less than its argument.

Parameter Types

x `int` or `float`

Return Type

same as input

The return type is `float` for `float` input in order to avoid problems with input values that are too large for the `int` data type.

Examples

Function Call	Result
`ceil(1.3)`	`2.0`
`ceil(-1.7)`	`-1.0`

degrees

```
degrees(x)
```

Description

`degrees` converts radians to degrees.

Parameter Types

x `int` or `float`

Return Type

`float`

Examples

Function Call	Result
`degrees(3.141592653589793)`	`180.0`

div

```
div(y, x)
```

Description

`div` computes the integer quotient `y`/`x` of two numbers `y` and `x`. If `x` is `0.0` (float) then `NaN` will be returned; it it is `0` (integer) then a runtime error will occur.

Parameter Types

y `int` or `float`

x same as `y`

Return Type

same as input

Examples

Function Call	Result
`div(9, 4)`	`2`
`div(9.3, 4.5)`	`2.0`

exp

```
exp(x)
```

Description

`exp` computes the exponential of a number.

Parameter Types

x `int` or `float`

Return Type

`float`

Examples

Function Call	Result
`exp(1.0)`	`2.718281828459045`

floor

```
floor(x)
```

Description

`floor` computes the largest integer not greater than its argument.

Parameter Types

x `int` or `float`

Return Type

same as input

The return type is `float` for `float` input in order to avoid problems with input values that are too large for the `int` data type.

Examples

Function Call	Result
`floor(1.3)`	`1.0`
`floor(-1.7)`	`-2.0`

ln

```
ln(x)
```

Description

`ln` computes the natural logarithm of a number. If the parameter is not strictly positive, `NaN` is returned.

Parameter Types

x `int` or `float`

Return Type

`float`

Examples

Function Call	Result
`ln(2)`	`0.6931471805599453`

log

```
log(x)
log(b, x)
```

Description

`log` computes the logarithm of a number `x` to base `b` (default: 10).

Parameter Types

x `int` or `float`

b (optional) same as `x`

Return Type

`float`

Examples

Function Call	Result
`log(100)`	`2.0`
`log(2.5, 6.25)`	`2.0`
`log(2, 8)`	`3.0`

mod

```
mod(y, x)
```

Description

`mod` computes the remainder of integer division `y`/`x` of two numbers `y` and `x`. If `x` is `0.0` (float) then `NaN` will be returned; it it is `0` (integer) then a runtime error will occur.

Parameter Types

y `int` or `float`

x same as `y`

Return Type

same as input

Examples

Function Call	Result
`mod(9, 4)`	`1`
`mod(9.3, 4.5)`	`0.3`

pi

```
pi()
```

Description

`pi` returns the π constant (more or less 3.14).

Return Type

`float`

Examples

Function Call	Result
`pi()`	`3.141592653589793`

power

```
power(a, b)
```

Description

`power` computes `a` raised to the power of `b`.

Parameter Types

a `int` or `float`

b same as `a`

Return Type

`float`

The return type is `float` even for integer input in order to have a uniform behavior for cases such as `power(2, -2)`.

Examples

Function Call	Result
`power(9.0, 3.0)`	`729.0`
`power(2, -1)`	`0.5`

radians

```
radians(x)
```

Description

`radians` converts degrees to radians.

Parameter Types

x `int` or `float`

Return Type

`float`

Examples

Function Call	Result
`radians(180)`	`3.141592653589793`

round

```
round(x)
```

Description

`round` computes the nearest integer of a number.

Parameter Types

x `int` or `float`

Return Type

same as input

The return type is `float` for `float` input in order to avoid problems with input values that are too large for the `int` data type.

Examples

Function Call	Result
`round(1.3)`	`1.0`
`round(0.5)`	`1.0`
`round(-1.7)`	`-2.0`

sign

```
sign(x)
```

Description

`sign` returns the sign of a number: 1 for positive numbers, -1 for negative numbers and 0 for zero.

Parameter Types

x `int` or `float`

Return Type

`int`

Examples

Function Call	Result
`sign(2)`	`1`

sqrt

```
sqrt(x)
```

Description

`sqrt` computes the square root of a number. If the parameter is negative, `NaN` is returned.

Parameter Types

x `int` or `float`

Return Type

`float`

Examples

Function Call	Result
`sqrt(2)`	`1.4142135623730951`

trunc

```
trunc(x)
```

Description

`trunc` computes the truncated integer (towards zero) of a number.

Parameter Types

x `int` or `float`

Return Type

same as input

The return type is `float` for `float` input in order to avoid problems with input values that are too large for the `int` data type.

Examples

Function Call	Result
`trunc(1.3)`	`1.0`
`trunc(-1.7)`	`-1.0`

width_bucket

```
width_bucket(x, left, right, count)
```

Description

`width_bucket` computes the bucket to which `x` would be assigned in an equidepth histogram with `count` buckets in the range $[\text{left}, \text{right}[$. Points on a bucket border belong to the right bucket. Points outside of the $[\text{left}, \text{right}[$ range have bucket number 0 and count $+ 1$, respectively.

Parameter Types

x `int` or `float`

left `int` or `float`

right `int` or `float`

count `int`

Return Type

`int`

Examples

Function Call	Result
`width_bucket(5, 0, 10, 5)`	`3`

Pseudo-Random Functions

The characteristics of the functions below are equal to those from the Go rand module. They are not suitable for cryptographic applications.

random

```
random()
```

Description

`random` returns a pseudo-random number in the range $0.0 <= x < 1.0$.

This function is not safe for use in cryptographic applications. See the Go math/rand package for details.

Return Type

`float`

Examples

Function Call	Result
`random()`	`0.6046602879796196`

setseed

```
setseed(x)
```

Description

`setseed` initializes the seed for subsequent `random()` calls. The parameter must be in the range $-1.0 <= x <= 1.0$.

This function is not safe for use in cryptographic applications. See the Go math/rand package for details.

Parameter Types

x `float`

Trigonometric Functions

All trigonometric functions take arguments and return values of type `float`. Trigonometric functions arguments are expressed in radians. Inverse functions return values are expressed in radians.

acos

```
acos(x)
```

Description

`acos` computes the inverse cosine of a number.

asin

```
asin(x)
```

Description

`asin` computes the inverse sine of a number.

atan

```
atan(x)
```

Description

`atan` computes the inverse tangent of a number.

cos

```
cos(x)
```

Description

`cos` computes the cosine of a number.

cot

```
cot(x)
```

Description

`cot` computes the cotangent of a number.

sin

```
sin(x)
```

Description

`sin` computes the sine of a number.

tan

```
tan(x)
```

Description

`tan` computes the tangent of a number.

String Functions

bit_length

```
bit_length(s)
```

Description

`bit_length` computes the number of bits in a string `s`. Note that due to UTF-8 encoding, this is equal to `octet_length(s) * 8`, not necessarily `char_length(s) * 8`.

Parameter Types

`s` `string`

Return Type

`int`

Examples

Function Call	Result
`bit_length("über")`	`40`

btrim

```
btrim(s)
btrim(s, chars)
```

Description

`btrim` removes the longest string consisting only of characters in `chars` (default: whitespace) from the start and end of `s`.

Parameter Types

`s` `string`

`chars` (optional) `string`

Return Type

`string`

Examples

Function Call	Result
`btrim(" trim ")`	`"trim"`
`btrim("xyxtrimyyx", "xy")`	`"trim"`

char_length

```
char_length(s)
```

Description

`char_length` computes the number of characters in a string.

Parameter Types

s `string`

Return Type

`int`

Examples

Function Call	Result
`char_length("über")`	`4`

concat

```
concat(s [, ...])
```

Description

`concat` concatenates all strings given as input arguments. `NULL` values are ignored, i.e., treated like an empty string.

Parameter Types

s and all subsequent parameters `string`

Return Type

`string`

Examples

Function Call	Result
`concat("abc", NULL, "22")`	`"abc22"`

concat_ws

```
concat_ws(sep, s [, ...])
```

Description

`concat_ws` concatenates all strings given as input arguments `s` using the separator `sep`. `NULL` values are ignored.

Parameter Types

sep `string`

s and all subsequent parameters `string`

Return Type

`string`

Examples

Function Call	Result
`concat_ws(":", "abc", NULL, "22")`	`"abc:22"`

format

```
format(s, [x, ...])
```

Description

`format` formats a variable number of arguments `x` according to a format string `s`.

See the Go package fmt for details of what formatting codes are allowed.

Parameter Types

s `string`

x and all subsequent parameters (optional) any

Return Type

`string`

Examples

Function Call	Result
`format("%s-%d", "abc", 22)`	`"abc-22"`

lower

```
lower(s)
```

Description

`lower` converts a string `s` to lower case. Non-ASCII Unicode characters are mapped to their lower case, too.

Parameter Types

s `string`

Return Type

`string`

Examples

Function Call	Result
`lower("ÜBer")`	`"über"`

ltrim

```
ltrim(s)
ltrim(s, chars)
```

Description

`ltrim` removes the longest string consisting only of characters in `chars` (default: whitespace) from the start of `s`.

Parameter Types

s `string`

chars (optional) `string`

Return Type

`string`

Examples

Function Call	Result
`ltrim(" trim ")`	`"trim "`
`ltrim("xyxtrimyyx", "xy")`	`"trimyyx"`

md5

```
md5(s)
```

Description

`md5` computes the MD5 checksum of a string `s` and returns it in hexadecimal format.

Parameter Types

s `string`

Return Type

`string`

Examples

Function Call	Result
`md5("abc")`	`"900150983cd24fb0d6963f7d28e17f72"`

octet_length

```
octet_length(s)
```

Description

`octet_length` computes the number of bytes in a string `s`. Note that due to UTF-8 encoding, this may differ from the number returned by `char_length`.

Parameter Types

s `string`

Return Type

`int`

Examples

Function Call	Result
`octet_length("über")`	`5`

overlay

```
overlay(s, repl, from)
overlay(s, repl, from, for)
```

Description

`overlay` replaces `for` characters in a string `s` with the string `repl`, starting at `from`. (Index counting starts at 0.) If `for` is not given, the length of `repl` is used as a default.

Parameter Types

`s` `string`

`repl` `string`

`from` `int`

`for` (optional) `int`

Return Type

`string`

Examples

Function Call	Result
`overlay("Txxxxas", "hom", 1)`	`"Thomxas"`
`overlay("Txxxxas", "hom", 1, 4)`	`"Thomas"`

rtrim

```
rtrim(s)
rtrim(s, chars)
```

Description

`rtrim` removes the longest string consisting only of characters in `chars` (default: whitespace) from the end of `s`.

Parameter Types

`s` `string`

`chars` (optional) `string`

Return Type

`string`

Examples

Function Call	Result
`rtrim(" trim ")`	`" trim"`
`rtrim("xyxtrimyyx", "xy")`	`"xyxtrim"`

sha1

```
sha1(s)
```

Description

`sha1` computes the SHA1 checksum of a string `s` and returns it in hexadecimal format.

Parameter Types

s `string`

Return Type

`string`

Examples

Function Call	Result
`sha1("abc")`	`"a9993e364706816aba3e25717850c26c9cd0d89d"`

sha256

```
sha256(s)
```

Description

`sha256` computes the SHA256 checksum of a string `s` and returns it in hexadecimal format.

Parameter Types

s `string`

Return Type

`string`

Examples

Function Call	Result
`sha256("abc")`	`"ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad"`

strpos

```
strpos(s, t)
```

Description

`strpos` returns the index of the first occurrence of `t` in `s` (index counting starts at 0) or -1 if it is not found.

Parameter Types

s `string`

t `string`

Return Type

`int`

Examples

Function Call	Result
`strpos("high", "ig")`	`1`

substring

```
substring(s, r)
substring(s, from)
substring(s, from, for)
```

Description

`substring(s, r)` extracts the substring matching regular expression `r` from `s`. See the Go regexp package for details of matching.

`substring(s, from, for)` returns the `for` characters of `str` starting from the `from` index. (Index counting starts at 0.) If `for` is not given, everything until the end of `str` is returned.

Which of those behaviors is used depends on the type of the second parameter (`int` or `string`).

Parameter Types

s `string`

r `string`

from `int`

for (optional) `int`

Return Type

`string`

Examples

Function Call	Result
`substring("Thomas", "...$")`	`"mas"`
`substring("Thomas", 1)`	`"homas"`
`substring("Thomas", 1, 3)`	`"hom"`

upper

```
upper(s)
```

Description

`upper` converts a string `s` to upper case. Non-ASCII Unicode characters are mapped to their upper case, too.

Parameter Types

s `string`

Return Type

`string`

Examples

Function Call	Result
`upper("ÜBer")`	`"ÜBER"`

Time Functions

distance_us

```
distance_us(u, v)
```

Description

distance_us computes the signed temporal distance from u to v in microseconds.

Parameter Types

u timestamp

v timestamp

Return Type

int

Examples

Function Call	Result
distance_us("2016-02-09T05:40:25.123Z"::timestamp, "2016-02-09T05:41:25.456Z"::timestamp)	60333000
distance_us(clock_timestamp(), clock_timestamp())	2

clock_timestamp

```
clock_timestamp()
```

Description

clock_timestamp returns the current date and time in UTC.

Return Type

timestamp

now

```
now()
```

Description

now returns the date and time in UTC of the point in time when processing of the current tuple started. In particular and as opposed to clock_timestamp, the timestamp returned by now() does not change during a processing run triggered by the arrival of a tuple. For example, in

```
SELECT RSTREAM clock_timestamp() AS a, clock_timestamp() AS b,
    now() AS c, now() AS d FROM ...
```

the values of a and b are most probably different by a very short timespan, but c and d are equal by definition of now().

now cannot be used in an EVAL statement outside of a stream processing context.

Return Type

timestamp

Array Functions

array_length

```
array_length(a)
```

Description

array_length computes the number of elements in an array a. Elements with a NULL value are also counted.

Parameter Types

a array

Return Type

int

Examples

Function Call	Result
array_length([3, NULL, "foo"])	3

Other Scalar Functions

coalesce

```
coalesce(x [, ...])
```

Description

`coalesce` returns the first non-null input parameter or `NULL` if there is no such parameter.

Parameter Types

x and all subsequent any

Return Type

same as input

Examples

Function Call	Result
`coalesce(NULL, 17, "foo")`	`17`

Aggregate Functions

Aggregate functions compute a single result from a set of input values. It should be noted that except for `count`, these functions return a `NULL` value when no rows are selected. In particular, `sum` of no rows returns `NULL`, not zero as one might expect, and `array_agg` returns `NULL` rather than an empty array when there are no input rows. The `coalesce` function can be used to substitute zero or an empty array for `NULL` when necessary.

Also note that most aggregate functions ignore singular `NULL` values in their input, i.e., processing is done as if this row had not been in the input. (One notable exception is the `array_agg` function that includes input `NULL` values in its output.)

array_agg

```
array_agg(x)
```

Description

`array_agg` returns an array containing all input values, including `NULL` values. There is no guarantee on the order of items in the result. Use the `ORDER BY` clause to achieve a certain ordering.

Parameter Types

x any

Return Type

`array`

avg

```
avg(x)
```

Description

`avg` computes the average (arithmetic mean) of all input values.

Parameter Types

x `int` or `float` (mixed types are allowed)

Return Type

`float`

bool_and

```
bool_and(x)
```

Description

`bool_and` returns `true` if all input values are true, otherwise `false`.

Parameter Types

x `bool`

Return Type

`bool`

bool_or

```
bool_or(x)
```

Description

`bool_or` returns `true` if at least one input value is true, otherwise `false`.

Parameter Types

x `bool`

Return Type

`bool`

count

```
count(x)
count(*)
```

Description

`count` returns the number of input rows for which `x` is not `NULL`, or the number of total rows if `*` is passed.

Parameter Types

x any

Return Type

`int`

json_object_agg

```
json_object_agg(k, v)
```

Description

`json_object_agg` aggregates pairs of key `k` and value `v` as a map. If both key and value are `NULL`, the pair is ignored. If only the value is `NULL`, it is still added with the corresponding key. It is an error if only the key is `NULL`. It is an error if a key appears multiple times.

A map does not have an ordering, therefore there is no guarantee on the result map ordering, whether or not `ORDER BY` is used.

Parameter Types

k `string`

v any

Return Type

`map`

max

```
max(x)
```

Description

`max` computes the maximum value of all input values.

Parameter Types

x `int` or `float` (mixed types are allowed)

Return Type

same as largest input value

median

```
median(x)
```

Description

`median` computes the median of all input values.

Parameter Types

x `int` or `float` (mixed types are allowed)

Return Type

`float`

min

```
min(x)
```

Description

`min` computes the minimum value of all input values.

Parameter Types

x `int` or `float` (mixed types are allowed)

Return Type

same as smallest input value

string_agg

```
string_agg(x, sep)
```

Description

`string_agg` returns a string with all values of `x` concatenated, separated by the (non-aggregate) `sep` parameter.

Parameter Types

x `string`

sep `string` (scalar)

Return Type

`string`

sum

```
sum(x)
```

Description

`sum` computes the sum of all input values.

Parameter Types

x `int` or `float` (mixed types are allowed)

Return Type

`float` if the input contains a `float`, `int` otherwise

Part VI

Indices and Tables

- genindex
- modindex
- search

Bibliography

[Crammer09] Koby Crammer, Alex Kulesza and Mark Dredze, Adaptive Regularization Of Weight Vectors, Advances in Neural Information Processing Systems, 2009

[cql] Arasu et al., "The CQL Continuous Query Language: Semantic Foundations and Query Execution", http://ilpubs.stanford.edu:8090/758/1/2003-67.pdf

[streamsql] Jain et al., "Towards a Streaming SQL Standard", http://cs.brown.edu/~ugur/streamsql.pdf