Expertmaker Accelerator

Quick Install using the "Project Skeleton" Repository + Examples

v2, 2018-05-30, Added performance example. See page 3.

Introduction and System Requirements

The `accelerator-project_skeleton` project provides a simple and convenient way to install the Accelerator locally. This document lists the necessary steps to set up the Accelerator using it. The last part of the document is dedicated to examples showing important Accelerator concepts.

The Accelerator will run on almost any hardware, from small laptops to large multi-CPU rack servers. It is assumed in this manual that the computer is running Ubuntu 16.04 LTS or Debian 9. The Accelerator team is actively testing on Ubuntu, Debian, and FreeBSD, but the Accelerator will most likely run on many other Linux distributions as well.

Installation

There are three steps in the installation: *resolve dependencies*, *clone repository*, and *run the initiation script*. These steps will be described next.

1. Dependencies

The first step is to make sure that all software package dependencies are met. This command will install all required packages

```
sudo apt-get install build-essential python-dev python3-dev zlib1g-dev git virtualenv
```

The installer requires only `git`, `virtualenv`, and some `dev` packages in order to compile C-code.

2. Clone Repository

Clone the `accelerator-project_skeleton` like this

```
git clone https://github.com/eBay/accelerator-project_skeleton.git
```

3. Setup

The Accelerator will now be installed *locally without any administrator privileges*. To continue, `cd` into the cloned directory

```
cd accelerator-project_skeleton
```

In this directory there is a file `init.py` that performs all the installation steps. It will work out-of-the-box, but for a customized install it is recommended to read and modify this file before continuing. The next step is to run the script

```
./init.py
```

This will do a complete setup, and the next section provides more information about the process. After the script is finished, the Accelerator installation is complete. It could be run by issuing

```
cd accelerator
./daemon.py
```

The first time the Accelerator is run, it will compile some functions written in the C programming language. On some systems, this process may generate a few warning messages, but that is okay. Setup is now complete.

Accessing Libraries in the Virtual Environment

Since the installer uses a virtual environment for installing packages, it must be initiated in all shells running for example `automatarunner` or `dsinfo`. This is done by issuing

```
source ../venv/py3/bin/activate
```

from the `accelerator` directory.

The Installation in more Detail

The `accelerator-project_skeleton` script `init.py` will setup virtual environments for Python2 and Python3 using `virtualenv`. In these virtual environments, it will download and install some depending packages, and `git clone` and install the `accelerator-gzutil` library. The Accelerator itself is `git cloned` into a git submodule in the `accelerator` directory.

The default configuration file is located in `conf/framework.conf`. This file is used to specify workdirs, method directories, and more. For more information, see the Accelerator User's Reference Manual.

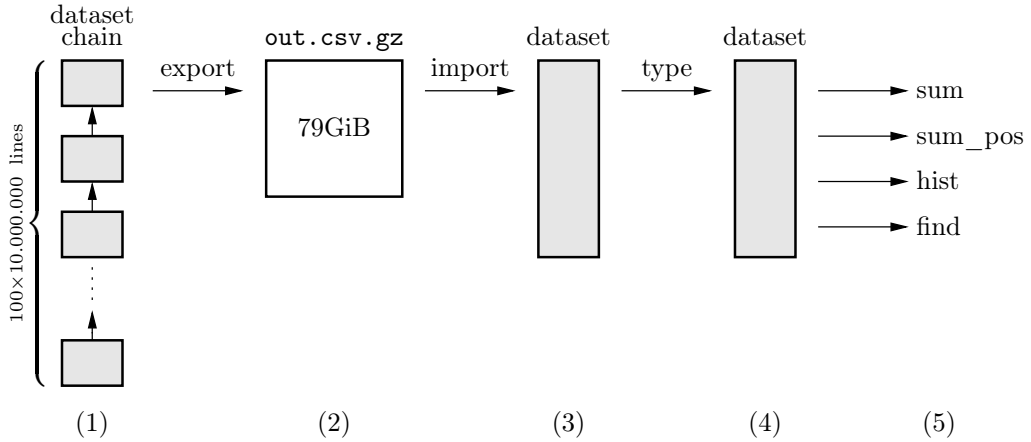
Acknowledgments

Thanks to Stefan Håkonsson for suggestions, testing, and proof reading.

References

`https://berkeman.github.io/pdf/acc_manual.pdf`

Example: Dataset Performance Measurement



This example will create one billion lines of random data and use it in a set of computations. By looking at the execution times, stored by default by the Accelerator, the example can be used as a simple performance measurement of a machine. The size of the dataset is an input parameter, and by default, a much smaller set of data is created in order to save space and execution time. At the end of the chapter, we present some quite impressive results.

Example Data Flow

The figure above shows the data flow in the example. The following happens

(1) The example starts by creating 100 chained datasets, each one containing 10 millon lines. (Default setting is actually only 1% of this, see section on changing dataset size.) The datasets will have the following columns

name	type	description
`a string`	`ascii`	a short random string, 10 characters long
`large number`	`number`	an integer evenly spread in the range $[-10^{20}, 10^{20}]$
`small number`	`number`	an integer evenly spread in the range $[-99, 99]$
`small integer`	`int32`	the same integer as in the previous column
`gauss number`	`number`	a normalised gaussian distribution
`gauss float`	`float64`	the same float as the previous column

Stored as a CSV (Comma Separated Values) file, this corresponds to 79 bytes per line on average.

(2) The complete dataset chain will be exported to a compressed text file in CSV-format. A one billion line CVS file is about 36GiB compressed and 79GiB uncompressed.

(3) The previously stored CSV file is imported again into a single dataset. Note that the import job will find and name the columns, and type all data to the `bytes` type.

(4) The imported dataset is typed, i.e. the data is written to disk in a more efficient format together with meta information such as `min` and `max` values.

(5) Finally, a number of operations are performed on the individual columns. The operation are

A. Sum all values of the column.

B. Sum all *positive* values of the column. Implemented using the iterator's `range`-argument.

C. Create histograms of the column's data.

D. Find a substring "`eBay`" in the string column and count number of matching lines.

Running the Example

The example is located in the `example_perf` directory. Here is a list of steps showing how to run it:

1. Clone and setup the `project_skeleton` as described earlier in this document

   ```
   git clone https://github.com/eBay/accelerator-project_skeleton.git
   cd accelerator-project_skeleton
   ./init.py
   ```

2. Now we can start the Accelerator. To start the server type

   ```
   cd accelerator
   ./daemon.py
   ```

   This terminal is now our *server* terminal, displaying the Accelerator's `stdout` and `stderr`.

3. Next, open a second terminal emulator for the *client* side. The Accelerator team prefers using GNU `screen`, but any terminal emulator like `xterm` is fine.

   In this second terminal (green), `cd` to the `accelerator` directory

   ```
   cd accelerator
   ```

   Since the installation is local (no administration privileges used), using virtual environments, we need to set up the virtual environment in this terminal like this

   ```
   source ../venv/py3/bin/activate
   ```

   Now we can run the the build script.

   ```
   ./automatarunner -P example_perf
   ```

   While running, the build script will print its progress to the terminal. The output is quite verbose so it might be helpful to read it. The Accelerator *server* will also print information during execution. Just switch to the terminal running `daemon.py` to see it.

   Please read the section below on the dataset size. Also note that all generated data will by default be stored in the home directory, which is probably not a good idea in most cases. Change this by editing the `workdir` assignment in `conf/framework.conf`.

4. Try running the build script again. The Accelerator will respond by printing the resulting statistics immediately. This is because all jobs have been build previously, and all `build()` calls will return links to existing jobs without executing any code. Thus, there is full tracking of each job's result as well as input source data. Changing one of the jobs or adding new jobs will not cause earlier jobs to be rebuilt. It is easy to see how this speeds up development while minimising the risk of mixing up results, code, and data. Try change something in, for example, `perf/a_example_histogram.py`, run again and see what happens!

Changing the Dataset Size

By default, the size of the example is 10 million lines, partitioned into 10 datasets of one million lines each. Changing the `if`-statement early in `example_perf/automata.py` from `False` to `True` will expand the example 100 times, to 1.000 million lines, by creating 100 datasets of 10 million lines each. Please note that this may take significant time (hours, depending on hardware) and about 140GB disk space.

Results and Discussion

The results from running on a fast[1] computer is shown below

```
operation                          exec time         rows/s

csvexport                            140.405      7,122,267

reimport total                       865.740      1,155,081
   csvimport                         793.695      1,259,930
   type                               72.045     13,880,208

sum
  small number                         0.900  1,110,709,359
  small integer                        0.776  1,288,260,784
  large number                         2.158    463,384,710
  gauss number                         1.203    830,964,312
  gauss float                          1.100    909,130,792

sum positive
  small number                         2.677    373,517,282
  small integer                        2.839    352,187,637
  large number                         3.658    273,371,843
  gauss number                         3.551    281,607,174
  gauss float                          3.425    291,951,520

histogram
  number                               9.399    106,398,284
  float                                9.444    105,882,626

find string                            2.662    375,653,405

Total test time                     1337.193

Example size is 1,000,000,000 lines.
Number of slices is 139.
```

On a high level, it seems we can

- Sum almost one billion (10^9) 64-bit floats per second. (Or almost 1.3 billion 32-bit ints per second.)
- Create a binned histogram of 100 million floats or integers per second.
- Find the number of strings (out of one billion) containing a specified substring in less than 3 seconds.

All this assuming that the data is imported by the Accelerator first. Importing the 79GiB CSV file into a six column dataset takes less than 15 minutes (866 seconds). Furthermore, this import is only carried out once. New or modified analysis jobs will use the existing import job. So, for example, if we change the binning of a histogram, it will take less than 10 seconds to get the new result.

[1] Quad "Intel(R) Xeon(R) CPU E7-8867 v4 @ 2.40GHz", 1TB RAM, and SSD storage

Example: Dataset Operations

This example shows how to create datasets and dataset chains, export datasets to CSV (Comma Separated Values) files, import datasets from CSV-files, append columns to datasets, and more.

Running the Example

The example is located in the `example1` directory. Here is a list of steps showing how to run it:

1. Clone and setup the `project_skeleton` as described earlier in this document

   ```
   git clone https://github.com/eBay/accelerator-project_skeleton.git
   cd accelerator-project_skeleton
   ./init.py
   ```

2. Now we can start the Accelerator. To start the server type

   ```
   cd accelerator
   ./daemon.py
   ```

   This terminal is now our *server* terminal, displaying the Accelerator's `stdout` and `stderr`.

3. Next, open a second terminal emulator for the *client* side. The Accelerator team prefers using GNU `screen`, but any terminal emulator like `xterm` is fine.

   In this second terminal (green), `cd` to the `accelerator` directory

   ```
   cd accelerator
   ```

   Since the installation is local (no administration privileges used), using virtual environments, we need to set up the virtual environment in this terminal like this

   ```
   source ../venv/py3/bin/activate
   ```

   Now we can run the the build script.

   ```
   ./automatarunner example1
   ```

Looking at the Output

The build script for this session is `example1/automata_example1.py`. Please have a look at this code with one eye while looking at the output with the other. Here is the output:

```
example1.automata_example1
        -  example1_create_dataset                          DONE  TEST-0  0.4 seconds
        -  example1_create_dataset                          DONE  TEST-1  0.4 seconds
        -  example1_create_dataset                          DONE  TEST-2  0.5 seconds
        -  example1_create_dataset                          DONE  TEST-3  0.4 seconds
        -  example1_create_dataset                          DONE  TEST-4  0.4 seconds
        -  csvexport                                        DONE  TEST-5  0.2 seconds
Exported file stored in "/home/ab/accelerator/workdirs/TEST/TEST-5/random.tsv"
```

This shows that five `example1_create_dataset` jobs have been run, and their jobids are `TEST-0` to `TEST-4`. By looking at the code, we see that the `csvexport` job is exporting the *last* (`TEST-4`) dataset to a CSV file on disk. (The whole dataset chain could be exported to CSV too by simply changing an input parameter to `csvimport`.) The location of this file is printed to `stdout` as well. We move on to

```
        -  csvimport                                        DONE  TEST-6  0.2 seconds
        -  dataset_type                                     DONE  TEST-7  0.2 seconds
```

Here, we've *imported* the CSV-file we just created. Note that an import types the data to `bytes`, so we issue an `dataset_type` job that does proper typing of the data. Next,

```
        -  example1_calc_average                          DONE  TEST-8  0.0 seconds
Column rint:  sum=-220338.000000, length=100000, average=-2.203380
        -  example1_calc_average                          DONE  TEST-9  0.0 seconds
Column rflt:  sum=50109.285978, length=100000, average=0.501093
```

there is a loop iterating over the columns of the dataset in `TEST-7`. In each iteration, it will compute the average of the values of the column and print it to `stdout`. Finally,

```
        -  example1_add_column                            DONE  TEST-10  0.2 seconds
        -  csvexport                                      DONE  TEST-11  0.3 seconds
```

appends a new column to the `TEST-7` dataset. This dataset is then exported to a CSV file. Which labels to export is explicitly set in this case. The build script then prints out a pretty-printed version of what is in the current Urd list

```
JobList(
   [  0]      Created_number_0 : TEST-0
   [  1]      Created_number_1 : TEST-1
   [  2]      Created_number_2 : TEST-2
   [  3]      Created_number_3 : TEST-3
   [  4]      Created_number_4 : TEST-4
   [  5]             csvexport : TEST-5
   [  6]             csvimport : TEST-6
   [  7]          dataset_type : TEST-7
   [  8] example1_calc_average : TEST-8
   [  9] example1_calc_average : TEST-9
   [ 10]   example1_add_column : TEST-10
   [ 11]             csvexport : TEST-11
)
```

Here we can see that the first five jobs have been given explicit names that makes it possible to uniquely identify them.

A Look at the Datasets

The `dsinfo` command is a simple tool to list the most important aspects of a dataset. Let's examine `TEST-4`, which is the last dataset in a chain

```
./dsinfo.py TEST-4
```

this results in

```
Parent: None
Hashlabel: None
Columns:
    rflt  float64
    rint  int64
2 columns
100,000 lines
Chain length 5, from TEST-0 to TEST-4
500,000 total lines
```

which shows that we have 100.000 lines in `TEST-4`, and 500.000 lines in the chain starting at `TEST-0`. Furthermore, it has two columns, one floating point and one integer, and the dataset is not hashed.

If we look at `TEST-10`, which is the dataset with a column appended to `TEST-7`, it looks like this

```
Parent: TEST-7/default
Hashlabel: None
Columns:
    prod  number
    rflt  number
    rint  number
3 columns
100,000 lines
```

Indeed, this dataset has a parent, `TEST-7/default`, and there are three columns, all typed to `number`. Similarly, it is worth looking at the imported dataset `TEST-6` too.

Conclusion

This example shows how to do some important dataset operations, such as importing data from a CSV file, exporting a dataset to a CSV file, typing a dataset, creating a chain of datasets, iterating over a dataset and compute something, and appending a new column to an existing dataset. It also shows how to find the absolute path to a job result, how to access a dataset's column names, and more. The idea is that this example could provide a way to start playing with the Accelerator and maybe use it in future projects.