The Accelerator

User's Reference

Version 0.981, 2019-06-24, **draft**

— Fast and reproducible data processing —

Anders Berkeman, Carl Drougge, and Sofia Hörberg

version: db47cc19

Document History

version	git hash	date	description
–	`772990f4`	2018-04-23	First open version.
0.90	`a6d6750b`	2018-05-28	Updated parts of Urd chapters.
0.98	`24b4b6c2`	2019-06-11	Major makeover.
0.981	`db47cc19`	2019-06-24	More on `depend_extra`, valid column names, appending columns in synthesis, and some formatting.

Contents

1 Introduction **7**
1.1 Main Design Goals . . . 8

2 Overview **9**
2.1 High Level View . . . 10
2.2 Jobs: Executing Code . . . 10
2.2.1 Basic Job Running: "Hello, World" . . . 10
2.2.2 Linking Jobs . . . 11
2.3 Datasets: Storing Data . . . 12
2.3.1 Importing Data . . . 12
2.3.2 Linking Datasets, Chaining . . . 13
2.3.3 Adding New Columns to a Dataset . . . 13
2.3.4 Multiple Datasets in a Job . . . 14
2.3.5 Parallel Dataset Access and Hashing . . . 14
2.3.6 Dataset Column Types . . . 15
2.3.7 Dataset Attributes . . . 15
2.4 Iterators: Working with Data . . . 15
2.4.1 Iterator Basics . . . 15
2.4.2 Parallel Execution . . . 15
2.4.3 Iterating over Several Columns . . . 16
2.4.4 Iterating over Dataset Chains . . . 16
2.4.5 Asserting the Hashlabel . . . 17
2.4.6 Dataset Translators and Filters . . . 17
2.4.7 Job Execution Flow and Result Passing . . . 17
2.4.8 Job Parameters . . . 18

3 Basic Build Scripting **20**
3.1 Build Scripts . . . 21
3.1.1 Building a Job: `urd.build()` . . . 21
3.1.2 Connecting Jobs . . . 21
3.1.3 Building Chained Jobs: `urd.build_chained()` . . . 22
3.1.4 Replaying Build Scripts . . . 22
3.2 Working with Build History: `urd.joblist` . . . 22
3.2.1 Printing a `JobList`: `urd.joblist.pretty` . . . 22
3.2.2 Finding jobs by name: `urd.joblist[name]` . . . 22
3.2.3 Finding the latest job: `urd.joblist.latest` . . . 23
3.2.4 Finding jobs: `urd.joblist.find(name)` . . . 23
3.2.5 Get a List of all Jobs: `urd.joblist.all` . . . 23
3.2.6 Indexing and Slicing a `JobList` . . . 23
3.3 Configuration Information: `urd.info` . . . 24
3.4 Summary . . . 24

4 Jobs **25**
4.1 Definitions . . . 26
4.1.1 Methods and jobs . . . 26
4.1.2 Jobids . . . 26
4.2 Python Packages . . . 26
4.2.1 Creating a new Package . . . 27

4.3 Method Source Files . . . 27
4.3.1 Creating a New Method . . . 27
4.3.2 Methods.conf . . . 27
4.4 Job Already Built Check . . . 28
4.5 Depend on More Files: `depend_extra` . . . 28
4.6 Avoiding Rebuild: `equivalent_hashes` . . . 28
4.7 Method Execution . . . 29
4.7.1 Execution Flow . . . 29
4.7.2 Function Arguments . . . 29
4.7.3 Parallel Processing: The `analysis` function, Slices, and Datasets . . . 30
4.7.4 Return Values . . . 30
4.7.5 Merging Results from `analysis` . . . 30
4.8 Method Input Parameters . . . 30
4.9 Reading all Method Parameters: `params` . . . 32
4.10 Accessing Another Job's Parameters: `job_params` . . . 33
4.11 Accessing Another Job's Datasets . . . 33
4.12 Job Directories . . . 34
4.13 Accessing Files in Another Job Directory . . . 34
4.14 Subjobs . . . 34
4.15 Formal Option Rules . . . 35
4.16 Jobs - a Summary . . . 38

5 Datasets **40**
5.1 Dataset Internals . . . 41
5.2 Chaining . . . 42
5.3 Slicing and Hashing . . . 42
5.4 Dataset as Input Parameter . . . 42
5.5 Datasets from Jobids . . . 43
5.6 Dataset Properties . . . 43
5.6.1 Column Names . . . 43
5.6.2 Column Properties . . . 43
5.6.3 Rows per Slice . . . 44
5.6.4 Dataset Shape . . . 44
5.6.5 Hashlabel . . . 44
5.6.6 Filename and Caption . . . 44
5.6.7 Chains . . . 44
5.7 Column Data Types . . . 45
5.7.1 Arbitrary precision numbers: `number` . . . 45
5.7.2 Standard Fixed Size Numbers . . . 45
5.7.3 Booleans . . . 45
5.7.4 Types Relating to Time . . . 45
5.7.5 String Types . . . 46
5.7.6 Raw Data . . . 46
5.7.7 Bitmasks . . . 46
5.7.8 JSON Type . . . 46
5.7.9 `parsed` Types . . . 46
5.8 Create a New Dataset . . . 46
5.8.1 Create in `prepare` + `analysis` . . . 46
5.8.2 Create in synthesis . . . 47
5.8.3 Completing Dataset Creation . . . 47
5.8.4 Creating Hashed Datasets . . . 48
5.8.5 Column Name Restrictions . . . 48
5.8.6 More Advanced Dataset Creation . . . 48
5.9 Appending New Columns to an Existing Dataset . . . 48
5.9.1 Appending New Columns in Analysis . . . 49
5.9.2 Appending New Columns in Synthesis . . . 49

6 Iterators **50**
6.1 The Three Iterators . . . 51
6.2 Basic Iteration . . . 52
6.3 Halting Iteration . . . 55
6.4 Iterating Over a Data Range . . . 55
6.5 Iterating in the Reverse Direction . . . 56
6.6 Hashed Datasets and on-the-fly Rehash . . . 56
6.7 Callbacks . . . 56
6.8 Translators . . . 57
6.9 Filters . . . 58

7 High Level Control: Urd **60**
7.1 Introduction to Urd . . . 61
7.2 A Simple Use case . . . 61
7.3 Urd Sessions and Lists . . . 61
7.4 A First Urd Query . . . 62
7.5 The Contents of the Stored Session . . . 62
7.6 Urd Sessions: `begin()` and `finish()` . . . 63
7.6.1 What if a Build Script is Run Again? . . . 64
7.7 Timestamp Resolution . . . 64
7.8 Finding Items in Urd . . . 64
7.8.1 Finding an Exact or Closest Match: `get()` . . . 65
7.8.2 Finding the Latest Session: `latest()` . . . 65
7.8.3 Finding the first item: `first()` . . . 65
7.9 Aborting an Urd Session: `abort()` . . . 65
7.10 Building Jobs: `build()` . . . 66
7.11 Changing workdir: `set_workdir()` . . . 67
7.12 Truncating and Updating . . . 67
7.12.1 Updating the last item . . . 67
7.12.2 Truncating a list . . . 67
7.12.3 Truncation Consequences: Ghosts . . . 67
7.13 Avoiding Recording Dependency . . . 68
7.14 More Search Functions . . . 68
7.14.1 Listing all urd lists: `list()` . . . 68
7.14.2 Listing all Items After a Specific Timestamp: `since()` . . . 68
7.15 The Urd HTTP-API . . . 68
7.15.1 The `list` endpoint . . . 69
7.15.2 The `since` endpoint . . . 69
7.15.3 The `first` and `latest` endpoints . . . 69
7.15.4 The `get` endpoint . . . 69
7.16 Profiling a Build Script: `print_profile()` . . . 70
7.17 Passing Flags from the Command Line . . . 70
7.18 Urd Internals . . . 70

8 Standard Methods **71**
8.1 `csvimport` – Importing Data Files . . . 72
8.1.1 Options . . . 72
8.1.2 Datasets . . . 73
8.1.3 Output . . . 73
8.1.4 Example Invocation . . . 73
8.2 `dataset_type` – Typing Datasets . . . 74
8.2.1 Datasets . . . 74
8.2.2 Options . . . 74
8.2.3 Example Invocation . . . 75
8.2.4 Typing . . . 75
8.3 `csvexport` – Exporting Text Files . . . 78
8.3.1 Example Invocation . . . 78
8.4 `dataset_rehash` – Hash Partition a Dataset . . . 80
8.4.1 Example Invocation . . . 80

8.4.2 Hashing Details . . . 80
8.4.3 Notes on Chains . . . 81
8.5 `dataset_filter_columns` – Removing Columns from a Dataset . . . 82
8.6 `dataset_sort` – Sorting a Dataset . . . 83
8.6.1 A Practical Limitation . . . 83
8.7 dataset_checksum, dataset_checksum_chain . . . 84

9 The Executables **85**
9.1 `daemon` Accelerator Server . . . 86
9.1.1 Invocation . . . 86
9.2 `urd` Job Database Server . . . 86
9.2.1 Invocation . . . 86
9.3 `runner` Build Script Runner . . . 87
9.3.1 Invocation . . . 87
9.3.2 Authorization to Urd . . . 88
9.4 `dsinfo` – Dataset Information . . . 88
9.4.1 Invocation . . . 88
9.5 `dsgrep` – Find Data in Dataset . . . 88
9.5.1 Invocation . . . 88
9.5.2 Abuse dsgrep to show datasets . . . 89

A Miscellaneous **90**
A.1 Daemon Configuration File . . . 91
A.2 Setting up Urd . . . 93
A.2.1 The Urd Database . . . 93
A.3 Workdirs . . . 95
A.3.1 Creating a Workdir . . . 95
A.4 Progress Indication: `C-t` . . . 95
A.5 Typical Installation . . . 95
A.6 Working with Relative Paths . . . 96
A.6.1 The SOURCE_DIRECTORY . . . 96
A.6.2 The RESULT_DIRECTORY . . . 96
A.6.3 Understanding Workdirs . . . 96
A.6.4 How to Create New Workdirs . . . 97

B Helper Functions **98**
B.1 Share Data Between Jobs: the `blob` Module . . . 99
B.1.1 Storing/Loading a Sliced File . . . 99
B.1.2 Default Value . . . 99
B.1.3 Save Files for Debugging . . . 99
B.2 Find the Full Path to a File in Another Job . . . 100
B.3 Symlinking . . . 100
B.4 job_params . . . 100
B.5 job_post . . . 101
B.6 json_encode . . . 101
B.7 json_decode . . . 101
B.8 json_save . . . 101
B.9 json_load . . . 101
B.10 DotDict . . . 102
B.11 OptionEnum . . . 102
B.12 OptionString . . . 102
B.13 RequiredOption . . . 102
B.14 OptionDefault . . . 102
B.15 gzutil . . . 103
B.16 profile_jobs . . . 103

Index **104**

Chapter 1

Introduction

The Accelerator is a tool for fast reproducible data processing, capable of working at high speed with terabytes of data with billions of rows on a single computer. Typical applications include data analysis work as well as live production systems for various data processing tasks, such as recommendation systems, and more. It has a small footprint and runs on laptops as well as rack servers.

The Accelerator was first used in 2012, and has been continuously developed and improved since. It has been in use in projects for companies like Safeway, Starbucks, eBay, Ericsson, and Vodafone. Most project have been related to data analysis, some to optimisation, and some projects have been recommendation systems running live for years. The Accelerator has evolved from being the core of these projects. In 2016, it was acquired by Ebay, who contributed it to the open source community early 2018.

Data set sizes in these projects range from a few hundred lines up to several tens of billions rows with multiple columns. The number of items in a dataset used in a live system was well above 10^{11}, and this was handled with ease on a *single* 32 core computer.

The authors are Anders Berkeman, Carl Drougge, and Sofia Hörberg. More than 1600 commits have been removed to clean up the open version of the code base. Extensive testing has been done by Stefan Håkonsson. The Accelerator is written in Python, with the exception of some critical parts that are written in the C programming language.

1.1 Main Design Goals

The Accelerator is designed to process log-files in "CSV"-like formats[1]. Log files bring determinism (i.e. reproducibility) and transparency, and most data can be represented in this format. The Accelerator is developed bottom up for high performance and simplicity, and the main design goals are:

Parallel processing should be made simple. Modern computers come with several cores, it should be straightforward to make use of them.

Data rates should be as fast as possible. It should be possible to process *large datasets*, even on commodity hardware.

Any processing step should be *reproducible*. The Accelerator maps any output to its corresponding input data and processing.

Never recompute old results, always "recycle" old jobs, when possible. Also, *sharing results* between multiple users should be effortless.

Organise and keep track of all jobs, files, and results in order to work with projects having 100.000s of input files and lots of programs and scripts processing them.

In addition, the Accelerator is designed to be used in all levels of a project, including data analysis, algorithm development, as well as production. Using the same tool for analysis and production closes the loop between input and output, making it really simple to analyse and get insights from the whole system.

[1] CSV is short for Comma Separated Values, but any separator character can be used. CSV files store data into rows and columns of text.

Chapter 2

Overview

This chapter presents an overview of the Accelerator's features in a rather non-formal way. It is based on an article published on the eBay Tech Blog website.

2.1 High Level View

The Accelerator is a client-server based application, and from a high level, it can be visualised like in figure 2.1. This manual will describe the setup in detail. For now, the most important

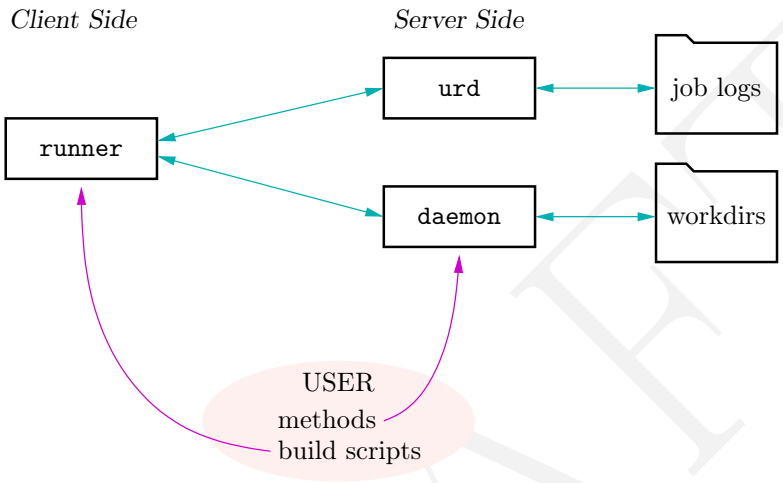


Figure 2.1: High level view of the Accelerator framework. See text for details.

features are as follows. On the left side there is a `runner` client. To the right, there are two servers, called `daemon` and `urd`. The `runner` program runs scripts, called `build scripts`, that execute jobs on the `daemon` server. This server will load and store information and results for all jobs executed using the *workdirs* file system based database.

In parallel, all jobs covered by a build script may be stored by the `urd` server into the *job logs* file system database. `urd` is also responsible for finding collections, or lists, of related previously executed jobs. The `urd` server is very capable and helps ensuring reproducibility and transparency, and its details are not covered in this chapter.

2.2 Jobs: Executing Code

The basic operation of the Accelerator is to execute small Python programs called *methods*. In a method, some reserved function names are used to execute code sequentially or in parallel and to pass parameters and results. A method that has completed execution is called a *job*.

The Accelerator keeps a record of all jobs that have been run. It turns out that this is very useful for avoiding unnecessary re-computing and instead rely on reusing previously computed results. This does not only speed up processing and encourage incremental design, but also makes it transparent which code and which data was used for any particular result, thus reducing uncertainty.

In this section, we'll look at basic job running and result sharing.

2.2.1 Basic Job Running: "Hello, World"

Let's begin with a simple "hello world" program. We create a method that we call `hello_world` with the following contents

```
def synthesis():
    return "hello world"
```

This program does not take any input parameters. It just returns a string and exits. Slightly simplified for clarity, we run the method like this. (Running methods on the Accelerator is further explained in chapter 3.)

```
jobid = build('hello_world')
```

When execution of the method is completed, a single link, called a `jobid` is the only thing that is returned to the user. The jobid points to a directory where the result from the execution is stored, together with all information that was needed to run the job plus some profiling information.

If we try to run the job again it will not execute, simply because the Accelerator remembers that the job has been run in the past. Instead of running the job again, it immediately returns the jobid pointing to the previous run. This means that from a user's perspective, there is no difference between job running and job result recalling! In order to have the job executing again, we have to change either the source code or input parameters.

Figure 2.2 illustrates the dispatch of the `hello_world` method. The created jobid is called `test-0`, and corresponding directory information is shown in green. The job directory contains several files, of which the most important to mention right now are

`setup.json`, containing job information;

`result.pickle`, containing the returned data; and

`method.tar.gz`, containing the method's source code.

Figure 2.2: A simple hello world program, represented as graph and work directory.

Jobids are named by their corresponding *workdir* plus an integer counter value. Workdirs are used to separate jobs into different physical locations, and jobs may be shared between users on the workdir level. A running Accelerator could have any number of workdirs associated, but only one at a time for writing.

2.2.2 Linking Jobs

The Accelerator makes it easy to split complex tasks into several smaller operations. Several jobs may be connected so that the next job will depend on the result of a previous job or set of jobs.

Assume that the job that we just run was computationally expensive, and that it returned a result that we'd like to use as input to further processing. To keep things simple, we demonstrate the principle by creating a method that just reads and prints the result from the previous job to `stdout`. We create a new method `print_result`, and it goes like this

```
import blob

jobids = {'hello_world_job',}

def synthesis():
    x = blob.load(jobid=jobids.hello_world_job)
    print(x)
```

This method expects the `hello_world_job` input parameter to be provided at execution time, and this is accomplished by the following *simplified* build script

```
jobid1 = build('hello_world')
jobid2 = build('print_result', hello_world_job=jobid1)
```

The `print_result` method then reads the result from the provided jobid and assigns it to the variable `x`, which is then printed to `stdout`. Note that this method does not return anything. Figure 2.3 illustrates the situation. (Note the direction of the arrow: the second job, `test-1` had `test-0` as input parameter, but `test-0` does not know of any jobs run in the future. Hence, arrows point to previous jobs.)

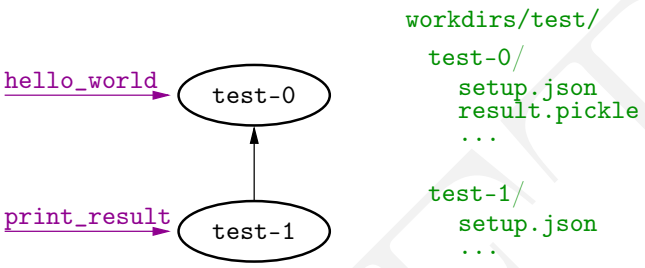


Figure 2.3: Jobid `test-0`, is used as input to the `print_result` job.

A complex task is split into several jobs, each reading intermediate results from previous jobs. The Accelerator will keep track of all job dependencies, so there is no doubt which jobs that are run when and on which data. Furthermore, since the Accelerator remembers if a job has been executed before, it will link and "recycle" previous jobs, which brings a great execution speed advantage. A recycled job is the proof of that the code, input- and output data is connected.

2.3 Datasets: Storing Data

The `dataset` is the Accelerator's default storage type for small or large quantities of data, designed for parallel processing and high performance. Datasets are built on top of jobs, so *datasets are created by methods and stored in job directories, just like any job result.*

Internally, data in a dataset is stored in a row-column format, and is typically *sliced* into a fixed number of slices to allow efficient parallel access, see figure 2.4. Columns are accessed independently, so there is no overhead in reading a set of or even a single column.

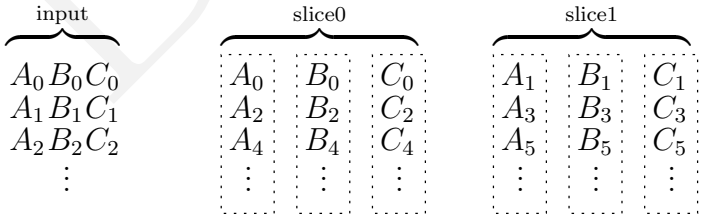


Figure 2.4: A dataset containing three columns, A, B, and C stored using two slices. Each dotted box corresponds to a file, so there are two files for each column, allowing for parallel read of the data using two processes.

Furthermore, datasets may be *hashed*, so that slicing is based on the hash value of a given column. Slicing on, for example, a column containing some ID string will partition all rows such that rows corresponding to any particular ID is stored in a single slice only. In many practical applications, hashing makes parallel processes independent, minimising the need for complicated merging operations. This is explained further in chapter **??**.

2.3.1 Importing Data

Let's have a look at the common operation of *importing*, data, i.e. creating a dataset from a file. See figure 2.5.

Figure 2.5: Importing `file0.txt`.

The bundled standard method `csvimport` is designed to parse a plethora of "comma separated values"-file formats and store the data as a dataset. The method takes several input options, where the file name is mandatory. Here is an example (non-simplified) invocation

```
def main(urd):
    jid = urd.build('csvimport', option=dict(filename='file0.txt'))
```

When executed, the created dataset will be stored in the resulting job, and the name of the dataset will by default be the jobid plus the string `default`. For example, if the `csvimport` jobid is `imp-0`, the dataset will be referenced by `imp-0/default`. In this case, and always when there is no ambiguity, the jobid alone (`imp-0`) could be used too. The reason for this is that a single job can contain any number of datasets, as we will see shortly.

2.3.2 Linking Datasets, Chaining

Just like jobs can be linked to each other, datasets can link to each other too. Since datasets are build on top of jobs, this is straightforward. Assume that we've just imported `file0.txt` into `imp-0/default` and that there is more data like it stored in `file1.txt`. We can import the latter file and supply a link to the previous dataset, see figure 2.6. The `imp-1` (or

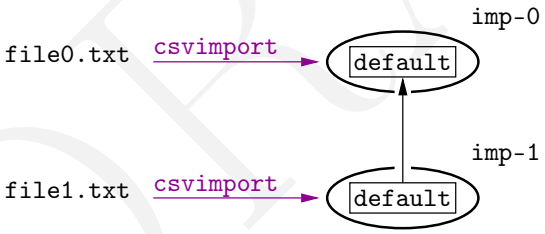


Figure 2.6: Chaining the import of `file1.txt` to the previous import of `file0.txt`.

`imp-1/default`) dataset reference can now be used to access all data imported from *both* files!

Linking datasets containing related content is called *chaining*, and this is particularly convenient when dealing with data that grows over time. Good example are all kind of *log* data, such as logs of transactions, user interactions, etc. Using chaining, we can extend datasets with more rows just by linking, which is a very lightweight constant time operation.

2.3.3 Adding New Columns to a Dataset

We have seen how easy it is to add more lines to data to a dataset using chaining. A dataset chain is created simply by linking one dataset to the other, so the overhead is minimal. Now we'll see that is it equally simple to add new columns to existing datasets. Adding columns is a common operation and the Accelerator handles this situation efficiently using links.

The idea is very simple. Assume that we have a "source" dataset to which we want to add one or more new columns. We create a new dataset containing *only* the new column(s), and while creating it we instruct the Accelerator to link all the source dataset's columns to the new dataset such that the new dataset appears to contain all columns from both datasets. (Note that this linking is similar to but different from chaining.)

Accessing the new dataset will transparently access all the columns in both the new and the source dataset in parallel, making it indistinguishable from a single dataset. See Figure 2.7.

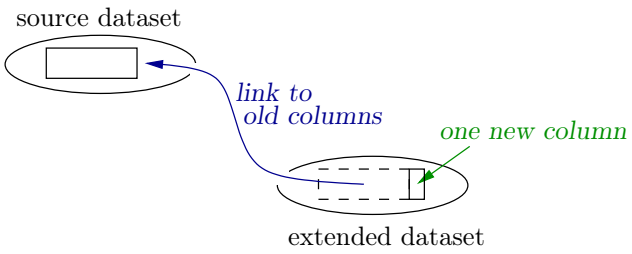


Figure 2.7: Adding one new column to the source dataset.

A common case is to compute new columns based on existing ones. In this case, we iterate over the rows in the source dataset, and for each iteration we write a new row of values to the new columns. This will be shown later in section 5.9

2.3.4 Multiple Datasets in a Job

Typically, a method creates a single dataset in the job directory, but there is no limit to how many datasets that could be created and stored in a single job directory. This leads to some interesting applications.

One application for keeping multiple datasets in a job is when data is split into subsets based on some condition. This could, for example, be when a dataset is split into a training set and a test set. One way to achieve this using the Accelerator is by creating a Boolean column that tells if the current row is train or test data, followed by a job that splits the dataset in two based on the value on that column. See Figure 2.8.

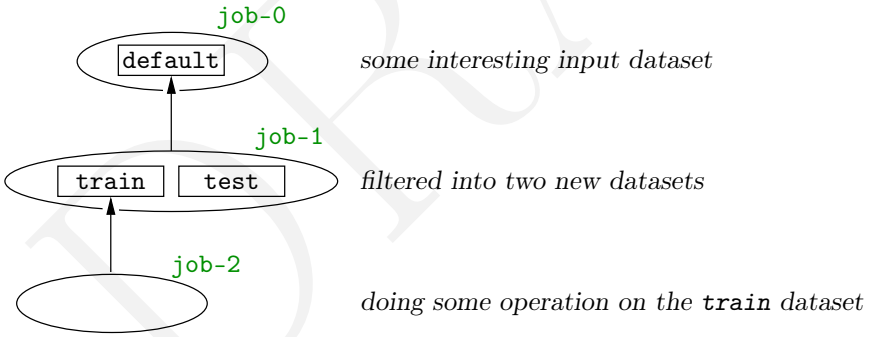


Figure 2.8: `job-1` separates the dataset `job-0/default` into two new datasets, named `job-1/train` and `job-1/test`.

In the setup of figure 2.8 we have full tracking from either `train` or `test` datasets. If we want to know the source of one of these sets, we just follow the links back to the previous jobs until we reach the source job. In the figure, `job-0` may for example be a `csvimport` job, and will therefore contain the name of the input file in its parameters. Thus, it is straightforward to link any data to its source.

Splitting a dataset into parts creates "physical" isolation while still keeping all the data at the same place. No data is lost in the process, and this is good for transparency reasons. For example, a following method may iterate over *both* datasets in `job-1` and by that read the complete dataset.

2.3.5 Parallel Dataset Access and Hashing

As shown earlier in this chapter, data in datasets is stored in multiple files for two reasons. One reason is that we can read only the columns that we need, without overhead, and the other is to allow fast parallel reads. The parameter `slices` determines how many slices that the dataset should be partitioned into, and it also sets the number of parallel process that may be used for processing the dataset. There is always one process for each slice of the dataset, and each process operates on a unique part of the dataset.

Datasets can be partitioned, sliced, in different ways. One obvious way is to use round robin, where each consecutive data row is written to the next slice, modulo the number of slices. This leads to "well balanced" datasets with approximately equal number of rows per slice. Another alternative to slicing is to slice based on the hash value of a particular column's values. Using this method, all rows with the same value in the hash column end up in the same slice. This is efficient for many parallel processing tasks, and we'll talk more about it later on.

2.3.6 Dataset Column Types

There are a number of useful types available for dataset columns. They include floating and integer point numbers, Booleans, timestamps, several string types, and json types for storing arbitrary data collections. Most of these types come with advanced parsers, making importing data from text files straightforward with deterministic handling of errors, overflows, and so on.

2.3.7 Dataset Attributes

The dataset has a number of attributes associated with it, such as shape, number of rows, column names and types, and more. An attribute is accessed like this

```
datasets = ('source',)
def synthesis():
    print(datasets.source.shape)
    print(datasets.source.columns)
```

and so on.

2.4 Iterators: Working with Data

Data in a dataset is typically accessed using an *iterator* that reads and streams one dataset slice at a time to a CPU core. The parallel processing capabilities of the Accelerator will dispatch a set of parallel iterators, one for each slice, in order to have efficient parallel processing of the dataset.

In this section, we'll have a look at iterators for reading data, how to take advantage of slicing to have parallel processing, and how to efficiently create datasets.

2.4.1 Iterator Basics

Assume that we have a dataset with a column containing movie titles named `movie`, and we want to know the ten most frequent movies. Consider the following example of a complete method

```
from collections import Counter
datasets = ('source',)

def synthesis():
    c = Counter(datasets.source.iterate(None, 'movie'))
    print(c.most_common(10))
```

This will print the ten most common movie titles and their corresponding counts in the `source` dataset. The code will run on a single CPU core, because we use the single-process `synthesis` function, which is called and executed only once. The `iterate` (class-)method therefore has to read through all slices, one at a time, in a serial fashion, and this is reflected by the first argument to the iterator being `None`.

2.4.2 Parallel Execution

The Accelerator is much about parallel processing, and since datasets are sliced, we can modify the above program to execute in parallel by doing the following modification

```
def analysis(sliceno):
    return Counter(datasets.source.iterate(sliceno, 'movie'))

def synthesis(analysis_res)
    c = analysis_res.merge_auto()
    print(c.most_common(10))
```

Here, we run `iterate` inside the `analysis()` function. This function is forked once for each slice, and the argument `sliceno` will contain an integer between zero and the number of slices minus one. The returned value from the analysis functions will be available as input to the synthesis function in the `analysis_res` Python iterable. It is possible to merge the results explicitly, but the iterator comes with a rather magic method `merge_auto()`, which merges the results from all slices into one based on the data type. It can for example merge `Counters`, `sets`, and composed types like `sets` of `Counters`, and so on. For larger datasets, this version will run much faster.

2.4.3 Iterating over Several Columns

Since each column is stored independently in a dataset, there is no overhead from reading a subset of a dataset's columns. In the previous section we've seen how to iterate over a single column using `iterate`. Iterating over more columns is straightforward by feeding a list of column names to `iterate`, like in this example

```
from collections import defaultdict
datasets = {'source',}

def analysis(sliceno):
    user2movieset = defaultdict(set)
    for user, movie in datasets.source.iterate(sliceno, ('user', 'movie')):
    user2movieset[user].add(movie)
    return user2movieset
```

This example creates a lookup dictionary from users to sets of movies. Note that in this case, we would like to have the dataset hashed on the `user` column, so that each user appears in exactly one slice. This will make later merging (if necessary) much easier.

It is also possible to iterate over all columns by specifying an empty list of columns or by using the value `None`.

```
...
def analysis(sliceno):
    for columns in datasets.source.iterate(sliceno, None):
    print(columns)
    break
```

This example will print the first row for each slice of a dataset and then exit.

2.4.4 Iterating over Dataset Chains

Previously, we've seen how to iterate over a single dataset using `iterate`. There is a corresponding function, `iterate_chain`, that is used for iterating over chains of datasets. This function takes a number of arguments, such as

`length`, i.e. the number of datasets to iterate over. By default, it will iterate over all datasets in the chain.

`callbacks`, functions that can be called before and/or after each dataset in a chain. Very useful for aggregating data between datasets.

`stop_id` which stops iterating at a certain dataset. This dataset could be from *another* job's parameters, so we can for example iterate exactly over all new datasets not covered by a previous job.

`range`, which allows for iterating over a range of data.

The `range` options is based on the max/min values stored for each column in the dataset. Assuming that the chain is sorted, one can for example set

```
range={timestamp, ('2016-01-01', '2016-01-31')}
```

in order to get rows within the specified range only. Using `range=` is quite costly, since it requires each row in the dataset chain with dates within the range to be checked against the range criterion. Therefore, there is a `sloppy` version that iterates over complete datasets in the chain that contains at least one row with a date within the range. This is useful, for example, to very quickly produce histograms or plots of subsets of the data.

2.4.5 Asserting the Hashlabel

Depending on how the parallel processing is implemented in a method, some methods will only work if the input datasets are hashed on a certain column. To make sure this is the case, there is an optional `hashlabel` parameter to the iterators that will cause a failure if the supplied column name does not correspond to the dataset's hashlabel.

On the other hand it is also possible to have the iterator re-hash on-the-fly. In general this is not recommended, since there is a `dataset_rehash` method that does the same and stores the result for immediate re-use. Using `dataset_rehash` will be much more efficient.

2.4.6 Dataset Translators and Filters

The iterator may perform data translation and filtering on-the-fly using the `translators` and `filters` options. Here is an example of how a dictionary can be fed into the iterator to map a column

```
mapper = {2: "HUMANLIKE", 4: "LABRADOR", 5: "STARFISH",}
for animal in datasets.source.iterate_chain(sliceno, \
  "NUM_LEGS", translator={"NUM_LEGS": mapper,}):
    ...
```

This will iterate over the `NUM_LEGS` column, and map numbers to strings according to the `mapper` dict.

Filters work similarly.

2.4.7 Job Execution Flow and Result Passing

Execution of code in a method is either parallel or serial depending on which function is used to encapsulate it. There are three functions in a method that are called from the Accelerator when a method is running, and they are `prepare()`, `analysis()`, and `synthesis()`. All three may exist in the same method, and at least one is required. When the method executes, they are called one after the other.

`prepare()` is executed first. The returned value is available in the variable `prepare_res`.

`analysis()` is run in parallel processes, one for each slice. It is called after completion of `prepare()`. Common input parameters are `sliceno`, holding the number of the current process instance, and `prepare_res`. The return value for each process becomes available in the `analysis_res` variable.

`synthesis()` is called after the last `analysis()`-process is completed. It is typically used to aggregate parallel results created by `analysis()` and takes both `prepare_res` and `analysis_res` as optional parameters. The latter is an iterator of the results from the parallel processes.

Figure 2.9 shows the execution order from top to bottom, and the data passed between functions in coloured branches. `prepare()` is executed first, and its return value is available to both the `analysis()` and `synthesis()` functions. There are `slices` (a configurable

parameter) number of parallel `analysis()` processes, and their output is available to the `synthesis()` function, which is executed last.

Return values from any of the three functions may be stored in the job's directory making them available to other jobs.

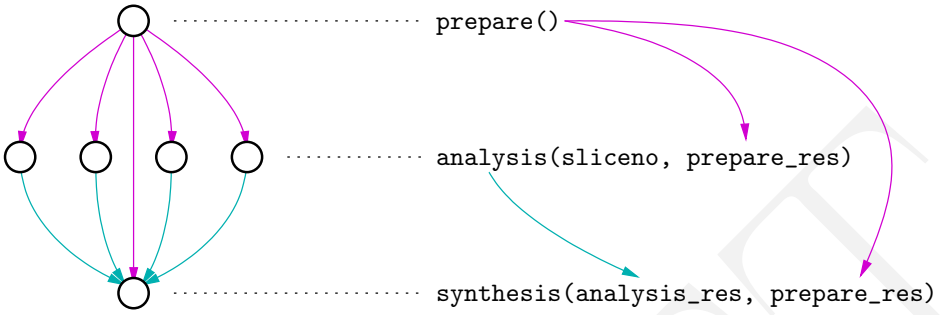


Figure 2.9: Execution flow and result propagation in a method.

2.4.8 Job Parameters

We've seen how jobids from completed jobs can be used as input to new jobs. Jobid parameters is one of three kinds of input parameters that a job can take. Here the input parameters are summarised:

`jobids`, a set of identifiers to previously executed jobs;

`options`, a dictionary of options; and

`datasets`, a set of input *datasets*.

See Figure 2.10. These will all be explained later in this chapter. Parameters are entered as global variables early in the method's source.

The Params Variable

A job's parameters are always available in the `params` variable. The `params` variable is made available by adding it as input to `prepare()`, `analysis()`, or `synthesis()`, like this

```
def synthesis(params):
    print(params)
```

Accessing the `params` variable for *another* job is done like this

```
jobids = {'thejob',}

def synthesis():
    print(jobids.thejob.params)
```

It turns out that it can be very useful to know for example which datasets another job has been processing, and so on.

Apart from the input parameters, the `params` variable also gives access to more information about the job and the current Accelerator setup, such as the total number of slices, the random seed, the hash of the source code, and method execution start time.

Input parameter `jobids`

The `jobids` parameter is a set containing any number of input jobids. For example

```
jobid = {'sales_stats', 'overhead_stats',}
```

When a method is to be executed, actual links to jobs, i.e. jobids, are passed using these parameters.

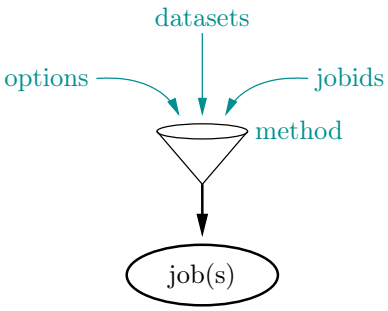


Figure 2.10: Execution flow of a method. The method takes optionally three kinds of parameters: `options`, `jobids`, and `datasets`.

Input parameter `options`

Options are supplied as a dictionary and is very flexible in typing. For example

```
options = dict(
    name = 'alice',
    defaultnames = set('alice', 'bob'),
    param1 = 1,
    param2 = float,
)
```

Here, values are defaults, so `param1` is default set to 1, while a floating point value must be supplied to `param2`.

Input parameter `datasets`

The `datasets` parameter is similar to the `jobids` parameter, each entry in the `datasets` parameter will be passed a value when the method is executed.

Chapter 3

Basic Build Scripting

Build scripts are used to instruct the Accelerator about which jobs to build. This chapter describes the basics of job building. More advanced features, using the `urd` server, is presented in chapter 7.

3.1 Build Scripts

Build scripts are executed by the `automatarunner` executable. Example of its use is provided in this section 9.3. A build script must contain the function `main`, since this is called by the runner, like this

```
def main(urd):
    ...
```

At run time, the `runner` inserts an object of the `Urd` class as argument to the `main` function. This `urd` object has a number of member functions and attributes useful for job building and tracking. For tracking purposes, it remembers all jobs that are built, together with their input parameters and some more meta information. This chapter will only cover the basic possibilities provided by Urd, and a more comprehensive view will be provided in the next chapter.

3.1.1 Building a Job: `urd.build()`

The `build` function is used to build a job from a method (i.e. source file). Here is an example of how to build the method `method1`:

```
def main(urd):
    urd.build('method1')
```

The full syntax for the `build` function is as follows

```
jobid = urd.build(method,
                  options={}, datasets={}, jobids={},
                  name='', caption='', workdir=None)
```

All parameters, except the name of the method, are optional, and the `options`, `datasets`, and `jobids` parameters must correspond to what is defined in the method to be executed. Urd will record the job based on the name of the method, unless the `name=` is specified. It is, for example, common to build several `csvimport` jobs, and using `name=` they can be told apart easily. It is also possible to assign a caption to a job, but this has no functional benefits.

When the job has been successfully built, the `build` function will return a reference, a *jobid* to the job. Similarly, if the job already existed in an available *workdir*, the `build` function immediately returns the jobid to that job without executing anything.

3.1.2 Connecting Jobs

It is straightforward to connect jobs by feeding the output jobid from the `build` function into a new job build. For example

```
jid_filter = urd.build('filter', datasets=dict(source=<some_input>))
jid_reduce = urd.build('reduce', datasets=dict(source=jid_filter))
```

In the example above, the first job, `filter`, creates a new dataset from its input dataset. This is then forwarded to the second job, `reduce`, using the jobid reference `jid_filter`.

What happens if the same build script is run a second time? If there has been no changes since the last run, the Accelerator will very quickly find the jobids to the existing jobs and exit, thus both finding the jobs as well as confirming the links and dependencies between them. On the other hand, if something has been modified, such as a method's source code or any input parameters, the affected job(s) will be re-executed. For example, assume that the input to the `filter` job is modified. This will cause this method to be executed again,

leading to a new job with a new jobid. This jobid is input to the `reduce` job, causing it too to be re-executed. Modifying the input to a sequence of jobs will cause all jobs to be re-executed. Modifying the parameters to, say, only the last job will cause just that job to be re-executed.

3.1.3 Building Chained Jobs: `urd.build_chained()`

There is a special version of `build()` that can be used for linking a set of dataset-creating jobs. This function was created for the purpose of having build scripts that imported a large set of files in a `for`-loop. The call looks like this

```
jobid = urd.build_chained('method1', name='myjob')
```

and it takes the same options as the standard `build` method, with the exception that name is mandatory. The method must also have a "`previous`" key in its `datasets` parameter. The way it works is that the Accelerator will look up the latest job having the same name and the new job to be built will have a link to this job inserted in the `dataset.previous` parameter.

3.1.4 Replaying Build Scripts

This was mentioned in the previous section, but it is so fundamental that it will be repeated here. Executing a build script a second time will not cause any new jobs to be executed, assuming nothing has been changed. Instead, the Accelerator will fill in the jobids of the existing jobs so that processing can continue immediately. A successful "replay" of a build script ensures the integrity and dependencies of the calculations. If nothing has changed, the same result remains. If, however, some of the code has been modified, the Accelerator will compute new jobs to reflect the new situation. The result may be different, and the user is notified.

3.2 Working with Build History: `urd.joblist`

Information about previously executed jobs is stored in the `urd.joblist` variable. This variable is of type `JobList`, which is basically a standard ordered Python `list` with some additional features for searching, profiling and pretty-printing.

3.2.1 Printing a `JobList`: `urd.joblist.pretty`

Create a `JobList` and pretty-print it

```
def main(urd):
    jid1 = urd.build('first')
    jid2 = urd.build('second', jobids=dict(first=jid1))
    print(urd.joblist.pretty)
```

which results in

```
JobList(
   [  0]  first : TEST-38
   [  1] second : TEST-39
)
```

The name is either the name of the method, or, if present, the name given explicitly by `name=` in `urd.build()`.

3.2.2 Finding jobs by name: `urd.joblist[name]`

Any job in the list can be accessed by its name, so for example the jobid to the `first` job could be retrieved by

```
urd.joblist['first'].jobid
```

Using the `JobList`, we can modify the previous example to

```
def main(urd):
    urd.build('first')
    urd.build('second', jobids=dict(first=urd.joblist['first'].jobid))
```

An exception will be raised if `first` does not exist in the joblist.

Note that if there are several jobs created by the same method, they will have the same key in the `JobList`. This situation could be resolved using unique names to the `name=` option to the `build()`-function as mentioned earlier. By default, the last (newest) matching instance will be returned.

The "`.jobid`" is a bit clumsy, but currently required to get the actual jobid for the job. There is also a `.method` that will return the name of the last method in the `JobList`.

3.2.3 Finding the latest job: `urd.joblist.latest`

An important special case is that the *latest* jobid is always accessible using `urd.joblist.jobid`, so the code example could also be written

```
def main(urd):
    urd.build('first')
    urd.build('second', jobids=dict(first=urd.joblist.jobid))
```

3.2.4 Finding jobs: `urd.joblist.find(name)`

The `urd.joblist.find()` function returns a new `JobList` containing all matching items, see this example

```
x = urd.joblist.find('csvimport')
```

The `x` variable will be a (perhaps) empty `JobList` containing all `csvimport` jobs present in `urd.joblist` in chronological order.

3.2.5 Get a List of all Jobs: `urd.joblist.all`

There is a `.all` method that will return the `JobList` as a comma separated list of jobids

```
print(urd.joblist.all)
```

producing something like

```
TEST-38,TEST-39
```

3.2.6 Indexing and Slicing a `JobList`

Since the `Joblist` is actually a `Python list`, it is possible to access individual items and slices like this

```
jl[3]
```

and like this

```
jl[-2:]
```

3.3 Configuration Information: `urd.info`

The dictionary `urd.info` contains configuration information from the Accelerator daemon. In particular, it contains these fields

name	description
`slices`	Configured number of slices.
`urd`	An URL to the Urd server, if configured, *None* otherwise.
`result_directory`	see section A.1.
`common_directory`	see section A.1.
`source_directory`	see section A.1.

3.4 Summary

The `urd` object has functionality for building and retrieving jobs. A job is built using `urd.build()`, and references to all built jobs will be stored in `urd.joblist`. These references could be fed as input parameters to new jobs so that the output from one job could be used as input by another. The `urd.joblist` variable is basically of type `list`, but with extra functionality to find previous jobs and their jobids.

Chapter 4

Jobs

This chapter covers most aspects of jobs and methods. It describes what methods are and how they are stored. When they are built and when not. What input parameters look like in full detail. How to access another job's parameters. Parallel processing. Return values and result merging. Storing and retrieving data. Building subjobs.

4.1 Definitions

4.1.1 Methods and jobs

In general, doing a computation on a computer follows the following equation

source code + input data and parameters + execution time → result

In the Accelerator context, the equation becomes

method + input data + input parameters + execution time → job

where the **method** is the source code, and the **job** is a directory containing

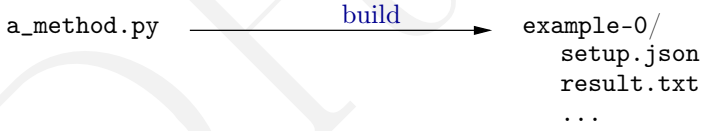
- any number of output files created by the running method, as well as
- a number of job meta information files.

The exact contents of the job directory will be discussed in section 4.12.

Computing a job is denoted job *building*. Jobs are **built** from methods. When a job has been built, it is *static*, and cannot be altered or removed by the Accelerator. Jobs are built either by

- a *build script*, see chapter 7, or
- by a method, using *subjobs*.

subjobs will be the topic of section 4.14. The following figure illustrates how a job `example-0` is built from the method `a_method.py`



The name is always unique so that it can be used as a reference to the job, and the reasons for the actual name (in this case `example-0`) will be apparent later in this chapter.

4.1.2 Jobids

A **jobid** is a reference to a job. Since all job directories have unique names, the name of the job is used as the jobid. In the example above, the job is uniquely identified by the string `example-0`.

4.2 Python Packages

Methods are stored in standard Python packages, i.e. in a directory that is

- reachable by the Python interpreter, and
- contains the (empty) file `__init__.py`.

In addition, for the Accelerator to accept a package, the following constraints need to be satisfied

- the package must contain a file named `methods.conf`, and
- the package must be added in the Accelerator's configuration file.

A package is reachable by Python if it is included in the `PYTHONPATH` variable. By default, the directory where the Accelerator daemon is started is included in this path. The next section guides through the steps required to create a new package.

4.2.1 Creating a new Package

The following shell commands illustrate how to create a new package directory

```
% mkdir <dirname>
% touch <dirname>/__init__.py
% touch <dirname>/methods.conf
```

The first two lines create a Python package, and the third line adds the file `methods.conf`, which is required by the Accelerator.

For security reasons, the Accelerator does not look for packages by itself. Instead, packages to be used need to be explicitly specified in the Accelerator's configuration file using the `method_directories=` assignment. Basically, the assignment should be a comma separated list of packages, see chapter A.1 for detailed information about the configuration file.

4.3 Method Source Files

Method source files are stored in Python packages as described in the previous section. The Accelerator searches all packages for methods to execute, and therefore *method names need to be globally unique*! In order to reduce risk of executing the wrong file, there are three limitations that apply to methods:

1. For a method file to be accepted by the Accelerator, the filename has to start with the prefix "`a_`";
2. the method name, without this prefix must be present on a separate line in the `methods.conf` file for the package, see section 4.3.2; and
3. the method name must be *globally* unique, i.e. there can not be a method with the same name in any other method directory visible to the Accelerator.

4.3.1 Creating a New Method

In order to create a new method, follow these steps

1. Create the method in a package viewable to the Accelerator using an editor. Make sure the filename is `a_<name>.py` if the method's name is `<name>`.
2. Add the method name `<name>` (without the prefix "`a_`" and suffix "`.py`") to the `methods.conf` file in the same method directory. See section 4.3.2.
3. (Make sure that the method directory is in the Accelerator's configuration file.)

4.3.2 Methods.conf

For a package to be useful to the Accelerator, it requires a `methods.conf` file. This file specifies which methods in the directory that are available for building. Files not specified in `methods.conf` cannot be executed. The file `methods.conf` provides an easy way to specify and limit which source files that can be executed, which is something that makes a lot of sense in any production environment.

The `methods.conf` is a plain text file with one entry per line. Any characters from a hash sign ("`#`") to the end of the line is considered to be a comment. It is allowed to have any number of empty lines in the file. Available methods are entered first on a line by stating the name of the method, without the `a_` prefix and `.py` suffix. If required, this can be followed by one or more whitespaces and a token "`py2`" or "`py3`", indicating which Python version to use when executing the method. The field is left blank if the method works on both versions. Here is an example

```
# this is a comment

test2           py2
```

```
test3           py3  # newer
test23               # works on both Python2 and Python3
#bogusmethod    py3
```

This file declares three methods corresponding to the filenames `a_test2.py`, `a_test3.py`, and `a_test23.py`, where the first two require Python2 and Python3 respectively. The method `test23` works on both versions. Another method `bogusmethod` is commented out and trying to build this will issue an error.

4.4 Job Already Built Check

From chapter 7 it is known that a method is built using the `build()` function in a build script, like this

```
def main(urd):
    urd.build('themethod', options=..., ...)
```

Prior to building a method, the Accelerator checks if an equivalent job has been built in the past. If it has, it will not be executed again. This check is based on two things:

1. the output of a hash function applied to the method source code, and
2. the method's input parameters.

The hash value is combined with the input arguments and compared to all jobs already built. Only if the hash and input parameter combination is unique will the method be executed.

4.5 Depend on More Files: `depend_extra`

A method may import code located in other files, and these other files can be included in the hash calculation as well. This will ensure that a change to an imported file will indeed force a re-execution of the method if a build is requested. Additional files are specified in the method using the `depend_extra` list, as for example:

```
from . import my_python_module

depend_extra = (my_python_module, 'mystuff.data',)
```

As seen in the example, it is possible to specify either Python module objects or filenames relative to the method's location.

The Accelerator daemon will suggest adding modules to a source file in the output log like this:

```
================================================================
WARNING: dev.a_test should probably depend_extra on myfuncs
================================================================
```

4.6 Avoiding Rebuild: `equivalent_hashes`

A change to a method's source code will cause a new job to be built upon running `build()`, but sometimes it is desirable to modify the source code and *not* causing a re-build. This happens, for example, when new functionality is added to an existing method. The way it worked before the code change is the same, so existing jobs do not need to be re-built. For this situation, there is an `equivalent_hashes` dictionary that can be used to specify which versions of the source code that are equivalent. The Accelerator helps creating this, if needed. This is how it works.

1. Find the hash `<old_hash>` of the existing job in that job's `setup.json`.

2. Add the following line to the method's source code

```
equivalent_hashes = {'whatever': (<old_hash>,)}
```

3. Run the build script. The daemon will print something like

```
============================================================
WARNING: test_methods.a_test_rechain has equivalent_hashes,
but missing verifier <current_hash>
============================================================
```

4. Enter the `current_hash` into the `equivalent_hashes`:

```
equivalent_hashes = {<current_hash>: (<old_hash>,)}
```

This line now tells that `current_hash` is equivalent to `old_hash`, so if a job with the old hash exists, the method will not be built again. Note that the right part of the assignment is actually a list, so there could be any number of equivalent versions of the source code. Some Accelerator standard methods like `csvimport` are good examples of this.

4.7 Method Execution

Methods are not executed from top to bottom. Instead, there are three functions that are called by the method dispatcher that controls the execution flow. These functions are

`prepare()`,

`analysis()`, and

`synthesis()`.

4.7.1 Execution Flow

The three functions `prepare`, `analysis`, and `synthesis` are called one at a time in that order. `prepare` and `synthesis` executes as single processes, while `analysis` provides parallel execution. None of them is mandatory, but at least one must be present for the method to execute.

4.7.2 Function Arguments

There are a few constants that can be passed into the executing functions at run time. Here is a complete list

name	description
`params`	A dictionary containing all parameters input from both the calling function and the Accelerator. This will be explained in detail in section 4.9.
`sliceno`	(For `analysis` only.) A unique identifier for each parallel `analysis` process.
`SOURCE_DIRECTORY`	A path defined in the Accelerator's configuration file, see A.6.1.
`RESULT_DIRECTORY`	A path defined in the Accelerator's configuration file, see A.6.2.

The input parameters `options`, `jobids`, and `datasets` are global, so they do not need to be explicit in a function call. The `analysis` function is special and takes a required argument `sliceno`, which is an integer between zero and the total number of slices minus one. This is the unique identifier for each `analysis` process, and is described in the next section.

4.7.3 Parallel Processing: The `analysis` function, Slices, and Datasets

The number of analysis processes is always equal to the number of dataset slices that the Accelerator has in its configuration file. The idea is that each slice in a dataset should have exactly one corresponding `analysis` process, so that all slices in a dataset can be processed in parallel. The input parameter `sliceno` to the `analysis` function is in the range from zero to the number of slices minus one. The number of slices is defined in the Accelerator's configuration file.

4.7.4 Return Values

Return values may be passed from one function to another. What is returned from prepare is called `prepare_res`, and may be used as input argument to `analysis` and `synthesis`. The return values from `analysis` is available as `analysis_res` in `synthesis`. The `analysis_res` variable is an iterator, yielding the results from each slice in turn. Finally, the return value from `synthesis` is stored permanently in the job directory. Here is an example of return value passing

```
# return a set of all users in the source dataset
options = dict(length=4)
datasets = (source',)

def prepare(options):
    return options.length * 2

def analysis(sliceno, prepare_res):
    return set(u for u in datasets.source.iterate(sliceno, 'user'))

def synthesis(analysis_res, prepare_res):
     return analyses_res.merge_auto()
```

In the current implementation, all return values are stored as Python `pickle` files.

Note that when a job completes, it is not possible to retrieve the results from `prepare` or `analysis` anymore. Only results from `synthesis` are kept.

4.7.5 Merging Results from `analysis`

In the example above, each analysis process returns a `set` that is constructed from a single slice. In order to create a set of all users in the `source` dataset, all these sets have to be merged. Merging could be done using a `for`-loop, but merging is dependent of the actual type, and writing merging functions is error prone. Therefore, `analysis_res` has a function called `merge_auto()`, that is used for merging. This function can merge most data types, and even merge container variables in a recursive fashion. For example,

```
h = defaultdict(lambda: defaultdict(set))
```

is straightforward to merge using `merge_auto`.

4.8 Method Input Parameters

There are three kinds of method input parameters assign by the `build` call: `jobids`, `datasets`, and `options`. These parameters are stated early in the method source code, such as for example

```
jobids = ('accumulated_costs',)
datasets = ('transaction_log', 'access_log',)
options = dict(length=4)
```

The input parameters are populated by the builder, see 7.

The `jobids` parameter list is used to input links, or jobids, to other jobs, while the `datasets` parameter list is used to input links to datasets. These parameters must be populated by the build call.

The `options` dictionary, on the other hand, is used to input any other type of parameters to be used by the method at run time. Options must not be populated by the build call, and this can be used for "global constants" in the method. An option assigned by the build call will override this, however.

Note that `jobids` and `datasets` are `tuples` (or `lists` or `sets`); and a single entry has to be followed by a comma as in the example above; while `options` is a dictionary. Individual elements of the input parameters may be accessed with dot notation like this

```
jobids.accumulated_cost
datasets.transaction_log
options.length
```

Each of these parameters will be described in more detail in following sections.

Input Jobids

The `jobids` parameter is a tuple of jobids linking this job to other jobs. Inside the running method, each item in the `jobids` tuple is of type `str` and may be used as a reference to the corresponding job. All items in the `jobids` tuple must be assigned by the builder to avoid run time errors. It is also possible to specify lists of jobids, see this example

```
jobids = ('source', ['alistofjobs'],)
```

where `source` is a single `jobid`, whereas `alistofjobs` is a list of `jobids`.

Input Datasets

The `datasets` parameter is a tuple of links to Datasets. In the running method, each item in the `datasets` variable is a tuple of objects from the `Dataset` class that works like a `list` with some add-on functionality. The `Dataset` class is described in a dedicated chapter 5. All items in the `datasets` tuple must be assigned by the builder to avoid run time errors. It is also possible to specify lists of jobids, see this example

```
datasets = ('source', ['alistofdatasets'],)
```

where `source` is a single `dataset`, whereas `alistofdatasets` is a list of `datasets`.

Input Options

The `options` parameter is of type `dict` and used to pass various information from the builder to a job. This information could be integers, strings, enumerations, sets, lists, and dictionaries in a recursive fashion, with or without default values. Assigning options from the build call is not necessary, but an assignment will override the "default" that is specified in the method. Options are specified like this

```
  options = dict(key=value, ... )  # or
  options = {key: value, ...}
```

Options are straightforward to use, but actually quite advanced. A formal overview is presented in section 4.15.

4.9 Reading all Method Parameters: `params`

There are two sources of parameters to a running method,

parameters from the caller, i.e. the `build()`-call, and

parameters assigned by the Accelerator when the job starts building.

All parameters are available in the `params` dictionary. Build parameters will be described thoroughly in section 4.8, and parameters from the Accelerator will be brought up at the end of this section.

Consider the following example method that pretty-prints the `params` variable. Note that the method explicitly expects input parameters to be assigned by the caller in the `jobids` and `datasets` constants, while `options` keeps the default value unless assigned in the build call.

```
import json

jobids = ('jid',)
datasets = ('source', 'parent',)
options = dict(mode='testing', length=3)

def synthesis(params):
    print(json.dumps(params, indent=4))
```

The corresponding output may look something like this

```
{
    "package": "dev",
    "method": "my_example_method",
    "jobid": "EXAMPLE-12",
    "starttime": 1520652213.4547446,
    "slices": 16,
    "options": {
        "mode": "testing",
        "length": 3
    },
    "datasets": {
        "source": "EXAMPLE-3",
        "parent": "EXAMPLE-2"
    },
    "caption": "fsm_my_example_method",
    "seed": 53231916470152325,
    "jobids": {
        "jid": "EXAMPLE-0"
    },
    "hash": "42af401251840b3798e9e78da5b5c5b4ef525ecc"
}
```

and a description of its keys

name	description
package	Python package for this method
method	name of this method
jobid	jobid of this job
starttime	start time in epoch format
caption	a caption
slices	number of slices of current Accelerator configuration
seed	a random seed available for use[1]
hash	source code hash value
options	input parameter
dataset	input parameter
jobids	input parameter

[1] The Accelerator team recommends *not* using `seed`, unless non-determinism is actually a goal.

4.10 Accessing Another Job's Parameters: `job_params`

The previous sections show that the the `params` data structure contains all input parameters and initialization data for a job. Sometimes it is useful to access another job's `params`. There is a special function for that, called `job_params`, and it is used like this

```
from extras import job_params

jobids = ('anotherjob',)

def synthesis():
    print(jobids.anotherjob)
    # will print something like 'jid-0_0'
    if jobids.anotherjob is not None:
        print(job_params(jobids.anotherjob).options)
        # will print the options of anotherjob
```

Note that if `jobids.anotherjob` is not defined, i.e. `None`, `job_params` will return the parameters of the *current* job. So in order to be safe, it makes sense to first check that the job exists.

4.11 Accessing Another Job's Datasets

Accessing another job's dataset parameters is done using the `Dataset` constructor like this

```
from dataset import Dataset

jobids = ('previous',)

def synthesis():
    # use the "default" dataset if unspecified
    ds_default = Dataset(jobids.previous)
    # specify the "source" dataset
    ds_source = Dataset(jobids.previous, 'source')
```

assuming that `jobids.previous` has the datasets `default` and `source`.

4.12 Job Directories

A successfully build of a method results in a new job directory on disk. The job directory will be stored in the current workdir and have a structure as follows, assuming the current workdir is `test`, and the current jobid is `test-0`.

```
workdirs/test/
    test-0/
        setup.json
        method.tar.gz
        result.pickle
        post.json
```

The `setup.json` will contain information for the job, including name of method, input parameters, and, after execution, some profiling information. `post.json` contains profiling information, and is written only of the job builds successfully. All source files, i.e. the method's source and `depend_extra`s are stored in the `tar`-archive `method.tar.gz`. Finally, the return value from `synthesis` is stored as a Python pickle file with the name `result.json`.

If the job contains datasets, these will be stored in directories, such as for example `default/`, in the root of the job directory.

4.13 Accessing Files in Another Job Directory

Files residing in another job is found using the `resolve_jobid_filename` function. This function returns the full path to a file specified by a jobid and a filename, like in this example

```
from extras import resolve_jobid_filename

fn = resolve_jobid_filename('test-43', 'names.txt')
print(fn)

# /home/acc/workdirs/test/test-43/names.txt
```

This function works in a build script too.

There is also a method *input option*, `JobWithFile`, that is worth mentioning in this context. This function is described in section 4.15.

4.14 Subjobs

Jobs may launch subjobs, i.e. methods may build other methods in a recursive manner. As always, if the jobs have been built already, they will immediately be linked in. The syntax for building a job inside a method is as follows, assuming we build the jobs in `prepare`

```
import subjobs

def prepare():
    subjobs.build('count_items', options=dict(length=3))
```

It is possible to build subjobs in `prepare` and `synthesis`, but not in `analysis`. The `subjobs.build` call uses the same syntax as `urd.build` described in chapter 7, so the input parameters `options`, `datasets`, `jobids`, and `caption` are available here too. Similarly, the return value from a subjob build is a jobid to the built job.

There are two catches, tough.

1. If there are datasets built in a subjob, these will not be explicitly available to Urd. A workaround is to link the dataset to the building method like this

```
from dataset import Dataset

def synthesis():
    jid = subjobs.build('create_dataset')
    Dataset(jid).link_to_here(name='thename')
```

with the effect that the building job will act like a Dataset, even though the dataset is actually created in the subjob. The `name` argument is optional, the name `default` is used if left empty, corresponding to the default dataset. It is possible to override the dataset's previous using the `override_previous` option, which takes a jobid (or *None*) to be the new `previous`.

```
Dataset(jid).link_to_here(name='thename', override_previous=xxx)
```

2. Currently there is no dependency checking on subjobs, so if a subjob method is changed, the calling method will not be updated. The current remedy is to use `depend_extra` in the building method, like this

```
import subjobs

depend_extra = ('a_childjob.py',)

def prepare():
    subjobs.build('childjob')
```

There is a limit to the recursion depth of subjobs, to avoid creating unlimited number of jobs by accident.

4.15 Formal Option Rules

This section covers the formal rules for the `options` parameter.

1. Typing may be specified using the class name (i.e. `int`), or as a value that will construct into such a class object (i.e. the number 3). See this example

```
options = dict(
    a = 3,      # typed to int
    b = int,    #          int
    c = 3.14,   #          float
    d = '',     #          str
)
```

   Values will be default values, and this is described thoroughly in the other rules.

2. An input option value is required to be of the correct type. This is, if a type is specified for an option, this must be respected by the builder. Regardless of type, *None* is always accepted.

3. An input may be left unassigned, unless

   - the option is typed to `RequiredOptions()`, or
   - the option is typed to `OptionEnum()` without a default.

   So, except for the two cases above, it is not necessary to supply option values to a method at build time.

4. If typing is specified as a value, this is the default value if left unspecified.

5. If typing is specified as a class name, default is *None*.

6. Values are accepted if they are valid input to the type's constructor, i.e. 3 and '3' are valid input for an integer.
7. *None* is always a valid input unless
   - RequiredOptions() and not none_ok set
   - OptionEnum() and not none_ok set

   This means that for example something typed to `int` can be overridden by the builder by assigning it to *None*. Also, *None* is also accepted in typed containers, so a type defined as `[int]` will accept the input `[1, 2, None]`.
8. All containers can be specified as empty, for example `{}` which expects a `dict`.
9. Complex types (like `dicts`, `dicts` of `lists` of `dicts`, ...) never enforce specific keys, only types. For example, `{'a': 'b'}` defines a dictionary from strings to strings, and for example `{'foo': 'bar'}` is a valid assignment.
10. Containers with a type in the values default to empty containers. Otherwise the specified values are the default contents. Example

```
options = dict(
    x = dict,           # will be empty dict as default
    y = {'foo': 'bar'}  # will be {'foo': 'bar'} as default
)
```

The following sections will describe typing in more detail.

Unspecifieds

An option with no typing may be specified by assigning `None`.

```
options = dict(length=None)  # accepts anything, default is None
```

Here, `length` could be set to anything.

Scalars

Scalars are either explicitly typed, as

```
options = dict(length=int)   # Requires an intable value or None
```

or implicitly with default value like

```
options = dict(length=3)     # Requires an intable value or None,
                             # default is 3 if left unassigned
```

In these examples, intable means that the value provided should be valid input to the `int` constructor, for example the number 3 or the string `'3'` both yield the integer number 3.

Strings

A (possibly empty) string with default value *None* is typed as

```
options = dict(name=str)     # requires string or None, defaults to None
```

A default value may be specified as follows

```
options = dict(name='foo')   # requires string or None, provides default value
```

And a string required to be specified and none-empty as

```
from extras import OptionString
options = dict(name=OptionString)        # requires non-empty string
```

In some situations, an example string is convenient

```
from extras import OptionString
options = dict(name=OptionString('bar') # Requires non-empty string,
                                        # provides example (NOT default value)
```

Note that "bar" is not default, it just gives the programmer a way to express what is expected.

Enums

Enumerations are convenient in a number of situations. An option with three enumerations is typed as

```
# Requires one of the strings 'a', 'b' or 'c'
from extras import OptionEnum
options = dict(foo=OptionEnum('a b c'))
```

and there is a flag to have it accept *None* too

```
# Requires one of the strings 'a', 'b', or 'c'; or None
from extras import OptionEnum
options = dict(foo=OptionEnum('a b c', none_ok=True))
```

A default value may be specified like this

```
# Requires one of the strings 'a', 'b' or 'c', defaults to 'b'
from extras import OptionEnum
options = dict(foo=OptionEnum('a b c').b
```

(The `none_ok` flag may be combined with a default value.) Furthermore, the asterisk-wildcard could be used to accept a wide range of strings

```
# Requires one of the strings 'a', 'b', or any string starting with 'c'
options = dict(foo=OptionEnum('a b c*'))
```

The example above allows the strings "a", "b", and all strings starting with the character "c".

Lists and Sets

Lists are specified like this

```
# Requires list of intable or None, defaults to empty list
options=dict(foo=[int])
```

Empty lists are accepted, as well as *None*. In addition, *None* is also valid inside the list. Sets are defined similarly

```
# Requires set of intable or None, defaults to empty set
options=dict(foo={int})
```

Here too, both *None* or the empty `set` is accepted, and *None* is a valid set member.

Date and Time

The following date and time related types are supported:

`datetime`,

`date`,

`time`, and

`timedelta`.

A typical use case is as follows

```
# a datetime object if input, or None
from datetime import datetime
options = dict(ts=datetime)
```

and with a default assignment

```
#  a datetime object if input, defaults to a datetime(2014, 1, 1) object
from datetime import datetime
options = dict(ts=datetime(2014, 1, 1))
```

More Complex Stuff: Containing Types

It is possible to have more complex types, such as dictionaries of dictionaries and so on, for example

```
# Requires dict of string to string
options = dict(foo={str: str})
```

or another example

```
# Requires dict of string to dict of string to int
options = dict(foo={str: {str: int}})
```

As always, containers with a type in the values default to empty containers. Otherwise, the specified values are the default contents.

A File From Another Job: `JobWithFile`

Any file residing in a jobdir may be input to a method like this

```
from extras import JobWithFile
options = dict(usefile=JobWithFile(jid, 'user.txt')
```

There are two additional arguments, `sliced` and `extras`. The `extras` argument is used to pass any kind of information that is helpful when using the specified file, and `sliced` tells that the file is stored in parallel slices.

```
options = dict(usefile=JobWithFile(jid, 'user.txt', sliced=True, extras={'uid': 37}))
```

(Creating sliced files is described in section 5.8.1.) In a running method, the `JobWithFile` object has these members

```
usefile.jobid
usefile.filename
usefile.sliced
usefile.extras
```

4.16 Jobs - a Summary

The concepts relating to Accelerator jobs are fundamental, and this section provides a shorter summary about the basic concepts.

1. Data and metadata relating to a job is stored in a job directory.

2. Jobids are pointers to such job directories.

The files stored in the job directory at dispatch are complete in the sense that they contain all information required to run the job. So the Accelerator job dispatcher actually just creates processes and points them to the job directory. New processes have to go and figure out their purpose by themselves by looking in this directory.

A running job has the process' *current working directory (CWD)* pointing into the job directory, so any files created by the job (including return values) will by default be stored in the job's directory.

When a job completes, the meta data files are updated with profiling information, such as execution time spent in single and parallel processing modes.

All code that is directly related to the job is also stored in the job directory in a compressed archive. This archive is typically limited to the method's source, but the code may have manually added dependencies to any other files, and in that case these will be added too. This way, source code and results are always connected and conveniently stored in the same directory for future reference.

3. Unique jobs are only executed once.

Among the meta information stored in the job directory is a hash digest of the method's source code (including manually added dependencies). This hash, together with the input parameters, is used to figure out if a result could be re-used instead of re-computed. This brings a number of attractive advantages.

4. Jobs may link to eachother using jobids.

Which means that jobs may share results and parameters with eachother.

5. Jobs are stored in workdirs.

6. There may be any number of workdirs.

This adds a layer of "physical separation". All jobs relating to importing a set of data may be stored in one workdir, perhaps named `import`, and development work may be stored in a workdir `dev`, etc. Jobids are created by appending a counter to the workdir name, so a job `dev-42` may access data in `import-37`, and so on, which helps manual inspection.

7. Jobs may dispatch other jobs.

It is perfectly fine for a job to dispatch any number of new jobs, and these jobs are called *subjobs*. A maximum allowed recursion depth is defined to avoid infinite recursion.

Chapter 5

Datasets

The Dataset class provides fast and simple access to data. It is the preferred way to store data using the Accelerator. Datasets are created by methods, and are therefore located inside job directories. There can be any number of Datasets in a job. Datasets are lightweight – adding new columns to a dataset, or appending datasets to eachother are instantaneous operations.

The most obvious way to generate a dataset is using the `cvsimport` method that creates a dataset from an input file. But much more advanced use is possible since a job may contain more than one Dataset. Being able to create several Datasets at once allows for efficient storage and access of data in some common practical situations. For example, a filtering job may split the input Dataset into two or more output Datasets that can be accessed independently.

For performance reasons, datasets are split into several slices, where each data row exists in exactly one of the slices. The actual slicing may be carried out in different ways, like round robin, or randomly, but an interesting approach is to slice according to the hash value of a certain column. Slicing according to a hashed column ensures that all rows with a certain column value always ends up in the same slice. Hash-based slicing often makes completely parallel processing of the dataset possible, since related data is not spread over different slices.

5.1 Dataset Internals

On a high level, the dataset stores a *matrix* of rows and columns. Each column is represented by a column name, or *label*, and all columns have the same number of rows. Columns are typed, and there is a wide range of types available. Typing will be introduced in section 5.7.

The dataset is further split into disjoint slices, where each slice holds a unique subset of the dataset's rows. Slicing makes simple but efficient parallel processing possible. See Figure 5.1. The number of slices is set initially by the user, and all workdirs that are used together in a project must use the same number of slices.

On a low level, there is one file stored on disk for each slice and column. A job that needs to read only a subset of the total number of columns may open and read from the relevant files only.

A technical note: If the number of slices is large and files are small, there will be a significant overhead from disk `seek()` if using rotating disks. The Accelerator mitigates this by changing the storage model to using single files with offset-indexing when appropriate.

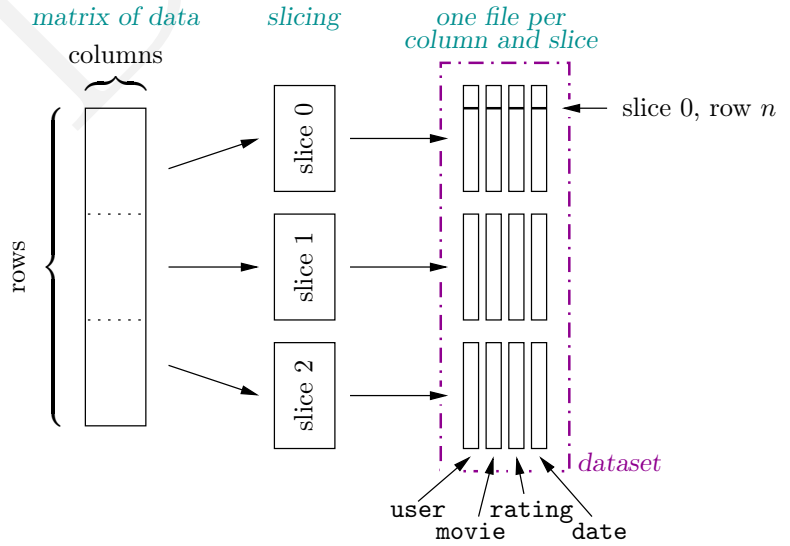


Figure 5.1: A "movie rating" dataset composed of four columns sliced into three slices.

5.2 Chaining

When a dataset is created, it is optional to input a link to another dataset using the parameter `previous`. This is called *chaining*. Chaining provides a lightweight way to append rows to datasets, simply by linking datasets together. A typical use case is the import of log files. A new dataset is created from each new log file, and each dataset chains to the previous. Reading the full chain will access all log rows. This has effect on the dataset *iterators* (see chapter 6), which may continue iterating over the next dataset in the chain when the current dataset is exhausted. Here is a an example of how `csvimport` jobs can be chained

```
jid1 = urd.build('csvimport', options=dict(filename='file1.txt'))
jid2 = urd.build('csvimport', options=dict(filename='file2.txt'),
                              datasets=dict(previous=jid1))
```

5.3 Slicing and Hashing

Datasets are by default sliced into a number of slices specified by the Accelerator's configuration file A.1. Slicing means that the rows of data in the dataset are distributed into different sets, called slices. Typically, there is one file on disk for each slice *and* column. The main reason for doing this is performance. All files could be read in parallel, and only files relevant to the task at hand are read.

Datasets can be sliced in a number of different ways. A simple method is to use round-robin, which cycles through the slices when writing. Round-robin will balance the number of rows per slice as equal as possible, which is a good thing in many scenarios. In semi-mathematical terms, round robin would be

$$n \longrightarrow n \bmod N$$

Meaning that input data row n is stored in slice $n \bmod N$. Another way is to slice by looking at the values of a fixed single column and put all rows with equal values in the same column. This way, data will be sliced "by content", and the number of rows per slice may vary significantly. In the context of the Accelerator, this is called *hashing*, and a dataset can be hashed on any single column. Written as an equation, it will look like this

$$n \longrightarrow \text{hash(data)} \bmod N$$

where "data" is the value of row n in the hashing column.

In many practical applications, data may be sliced using a hashing function so that data in each slice is *independent*. Independent data means that processing of the dataset can be carried out in a completely parallel fashion.

The hash function used by the Accelerator is a well-known function called `siphash24` that is available from the Accelerator's `gzutil` library

```
from gzutil import siphash24

y = siphash24(x)
```

5.4 Dataset as Input Parameter

Datasets may be input to a method using the `datasets` input parameter list. In a running job, the items in this list are object of the `Dataset` class. This class has a number of member functions, for example

```
datasets = ('source',)

def synthesis():
  print(datasets.source.shape)
```

will print the number of rows and columns of the `source` Dataset.

5.5 Datasets from Jobids

A Dataset object can be instantiated from a jobid, like this

```
from dataset import Dataset
...
ds = Dataset('foo-0')
```

In this case, `ds` will be an object based on the default dataset residing at jobid `foo-0`. If the dataset is stored by a different name, i.e. different from `default`, it may be accessed like this

```
ds = Dataset('foo-0/bar')
# or
ds = Dataset(('foo-0', 'bar'))
```

A common operation is to fetch a dataset from another jobid

```
ds = Dataset((jobids.previous, 'source'))
```

5.6 Dataset Properties

The Dataset class has a number of member functions and attributes that is intended to make it simple to work with. These functions will be described in the next sections.

5.6.1 Column Names

All columns in a dataset may be acquired using the `columns` property, like this

```
datasets = ('source',)

def synthesis():
  print(datasets.source.columns.keys())
  # may print something like
  # ['GTIN', 'date', 'locale', 'subsource']
```

The `columns` attribute is actually a dictionary from column name to properties, as will be shown in the next section.

Not all column names are valid, see section 5.8.5 for more information.

5.6.2 Column Properties

For each column, the name, type, and if applicable, the minimum and maximum values are accessible like this

```
print(datasets.source.columns['locale'].type)
# number

print(datasets.source.columns['locale'].name)
# locale

print(datasets.source.columns['locale'].min)
# 3

print(datasets.source.columns['locale'].max)
# 107
```

Creation of the `max` and `min` values is a simple operation that is done in linear time when the dataset is created. Maximum and minimum values are used for example when iterating over chains of sorted datasets, to quickly decide if a dataset is outside range and can be skipped in its entirety, see section 6.4.

5.6.3 Rows per Slice

It may be interesting to see how many rows there are per slice in a dataset. This information is available as a list, for example

```
print(datasets.source.lines)
# [5771, 6939, 6212, 6312, 6702, 6341, 5988, 6195,
#  6741, 6587, 6518, 5840, 6327, 5933, 6745, 6673,
#  6536, 6405, 6259, 6455, 6036, 6088, 6937, 6245,
#  6418, 6437, 6360, 6106, 6878]
```

The first item in the list is the number of rows in slice 0, and so fourth. The total number of rows in the Dataset is the sum of these numbers.

5.6.4 Dataset Shape

The shape of the dataset, i.e. the number of rows and columns, is available from the shape attribute

```
print(datasets.source.shape)
# (4, 184984)
```

The second number is exactly the sum of the number of lines for each slice from above.

5.6.5 Hashlabel

If the dataset is hashed on a particular column, the name of this column is stored in the hashlabel attribute

```
print(datasets.source.hashlabel)
# GTIN
```

5.6.6 Filename and Caption

The dataset may have a filename associated to it. This makes sense in situations for example where the dataset is created from an input data file using `csvimport` or similar. The filename is accessible using the `filename` attribute:

```
print(datasets.source.filename)
# /data/incoming/raw_repository_5391.gz
```

Furthermore, it is possible to set a caption at dataset creation time. The caption is entirely user-defined and has no function in the Accelerator. The caption is accessible like this

```
print(datasets.source.caption)
# rehash_of_raw_data
```

5.6.7 Chains

The previous dataset in a dataset chain is found in the `previous` attribute:

```
print(datasets.source.previous)
# import-4893/default
```

A list containing all datasets in a chain is returned by

```
print(datasets.source.ds_chain)
```

5.7 Column Data Types

The dataset columns are typed. This means, for example, that if a column's type is `date`, each value read from the column will be in Python's `date` format, ready for processing. The same goes for all types, including `json`, which may return rather complex datatypes.

All available types are shown in the following table. More details follow in the next sections.

name	description
`bytes`	raw data
`number`	float or int
`float64`	64 bit (double) float
`float32`	32 bit float
`int64`	64 bit signed integer
`int32`	32 bit integer
`bits64`	64 bit bitmask
`bits32`	32 bit bitmask
`bool`	True or False
`date`	date
`time`	time
`datetime`	complete date and time object
`ascii`	ascii is faster in python2, otherwise use unicode
`unicode`	use for strings
`json`	a datastructure that is jsonable
`parsed:number`	int, float or string parsing into `number`
`parsed:float64`	int, float or string parsing into `float64`
`parsed:float32`	int, float or string parsing into `float32`
`parsed:int64`	int, float or string parsing into `int64`
`parsed:int32`	int, float or string parsing into `int32`
`parsed:json`	string containing parseable json

5.7.1 Arbitrary precision numbers: `number`

The type `number` is integer when possible and float otherwise. it can handle very large numbers, up to $\pm(2^{1007}-1)$. The `number` type occupies a minimum of nine bytes on disk, where eight is for the number itself and the additional byte is a marker.

5.7.2 Standard Fixed Size Numbers

The common `int` and `float` types in 32 and 64 bit versions are available for use when the range of the data is known.

5.7.3 Booleans

The `bool` type is used to store logical *`True`* or *`False`* values only.

5.7.4 Types Relating to Time

The `date`, `time`, and `datetime` are compatible with Python's corresponding classes, where `datetime` is the combination of `date` and `time`. A column that is typed to any of these may directly take advantage of the high level time related methods, like for example

```
for ts in datasets.source.iterate(sliceno, 'timestamp'):
    print(ts.strftime('%Y-%m-%d')
```

5.7.5 String Types

There is a `unicode` type for strings. On Python2, the `ascii` type could be used as well. The `unicode` type executes faster on Python3.

5.7.6 Raw Data

The `bytes` type is used to store raw data, such as binary image files. The upper storage limit for a value typed as `bytes` is almost 2GB ($2^{31} - 1$ bytes). The `csvimport` standard method uses this type for all data in its output dataset.

5.7.7 Bitmasks

BITMASKS?

5.7.8 JSON Type

It is possible to store more complex data structures using the JSON format. The `JSON` type accept a JSON-able datastructure as input.

5.7.9 `parsed` Types

In addition, there are a few types prefixed with `parsed:` that allow for a more flexible assignment of values. For example, the `parsed:number` type accepts both `int`s and `float`s, as well as strings that are parseable to a number, such as `'3.14'`.

5.8 Create a New Dataset

Datasets are created by methods using the `DatasetWriter` class. The most common scenario is to set up the new dataset in `prepare`, and write data to it in parallel in `analysis`, but is is also possible to write a dataset in an entirely serial fashion in `synthesis`. When a dataset-creating method terminates, it will create and store all required meta-information, such as min/max values, for the created dataset(s) automatically.

The most common arguments to `DatasetWriter` are

name	description
`filename`	if there is a filename associated, store it here
`caption`	additional caption
`hashlabel`	name of column to hash by when slicing
`previous`	previous Dataset, for chaining
`name`	dataset name, default set to `default`
`parent`	parent Dataset when adding columns

5.8.1 Create in `prepare` + `analysis`

The following example will use `DatasetWriter` to create a Dataset with three columns. The name of the dataset will be `firstset`. If the name is omitted, the the name `default` will be used instead. The writer will be initialised in `prepare`, and data will be written to the Dataset in `analysis`. Note that the example creates a dataset *chain*, linking the dataset under creation to the dataset named `previous` from the input parameters. `X`.

```
from dataset import DatasetWriter
datasets = ('previous',)

def prepare():
    dw = DatasetWriter(
        previous = datasets.previous,
        name = 'firstset'
    )
    dw.add('X', 'number')
    dw.add('Y', 'unicode')
    dw.add('Z', 'time')
    return dw

def analysis(sliceno, prepare_res):
    dw = prepare_res
    ...
    for x, y, z in some_data:
        dw.write(x, y, z)
```

The order of the variables in the dw.write function call is the same as the order of the add calls in prepare. There are a few alternative ways of writing data, as shown here

```
dw.write_dict({column: value})

dw.write_list([value, value, ...])

dw.write(value, value, ...)
```

Several Datasets can be created simultaneously using multiple writers with different names.

5.8.2 Create in synthesis

There are two possible ways to create a Dataset in `synthesis`. One is to first set a slice number

```
dw.set_slice(sliceno)
```

before writing data into that slice. The other is to use one of the `split_write` functions

```
dw.get_split_write_dict()({column: value})

dw.get_split_write_list()([value, value, ...])

dw.get_split_write()(value, value, ...)
```

These writers will write round-robin if the dataset is not hashed, and to the "right" slice if the dataset is hashed.

5.8.3 Completing Dataset Creation

Normally, there is no need to tell the `DatasetWriter` that the last line of data is written. This is handled automatically when the method exits. In some situations, such as when a dataset is to be used by a `subjob` launched from the creating method, it is necessary to manually tell the writer that the dataset is complete. This is done by calling `finish()` as shown below

```
dw.finish()
```

The `finish()`-call returns a dataset object, so it is possible to start using the finished dataset immediately like this

```
ds = dw.finish()
it = ds.iterate(None, 'user')
...
```

5.8.4 Creating Hashed Datasets

Creating a hashed dataset is accomplished by setting the `hashlabel` argument of `DatasetWriter`. It is up to the dataset generating method to make sure that each row is written to the correct slice, according to the hash function value of the hashlabel column. A row should go into slice n if and only if

```
from gzutil import siphash24
assert siphash24(hashcol) % options.slices == n
```

Otherwise an exception will occur. It is possible to override this behavior by calling

```
dw.enable_hash_discard()
```

first in each slice or after each `set_slice()`. Then, writes that belongs to another slice are silently ignored.

5.8.5 Column Name Restrictions

Column names must be valid Python identifiers. Invalid characters are replaced by the underline(_) character. The underline character is also used to make column names unique when necessary. The table below shows some examples.

input	converted	comment
`"-"`	`"_"`	Converting to valid python identifier.
`"a b"`	`"a_b"`	Converting to valid python identifier.
`"42"`	`"_42"`	Converting to valid python identifier.
`"print"`	`"print_"`	`print` is a keyword (in py2).
`"print@"`	`"print__"`	`print_` is taken.
`"None"`	`"None_"`	`None` is a keyword (in py3).

5.8.6 More Advanced Dataset Creation

Currently out-of-scope of this manual. Please see the file `dataset.py` for full information.

5.9 Appending New Columns to an Existing Dataset

With minimal overhead, existing datasets could be extended with new columns. Internally, this is implemented by storing the new column data together with a pointer to the original, "parent", dataset.

Appending new columns works the same way as when creating a dataset, with the exception that a link to a dataset that is to be appended to is input to the writer constructor. Columns can be appended either in `analysis` or `synthesis`, as shown in the two following sections. Note that appending a column does only apply to one single dataset, and not to the complete chain of datasets, if present.

5.9.1 Appending New Columns in Analysis

The following example appends one column to an existing dataset `source`, while chaining to the dataset `previous`.

```
from dataset import DatasetWriter

datasets = ('source', 'previous',)

def prepare():
    dw = DatasetWriter(
        parent=datasets.source,
        previous=datasets.previous,
        caption='with the new column'
    )
    dw.add('newcolname', 'unicode')
    return dw

def analysis(sliceno, prepare_res):
    dw = prepare_res
    ...
    dw.write(value)
```

The `DatasetWriter` will automatically check that the number of appended rows does match the number of rows in the parent dataset. Otherwise, an error will be issued and execution will terminate.

5.9.2 Appending New Columns in Synthesis

A straightforward way to append columns to a dataset in synthesis is using the `set_slice()` function, as shown in the example below.

```
def synthesis(params):

    dw = DatasetWriter(parent=datasets.source)
    dw.add('newcolumn', 'json')

    for sliceno in range(params.slices):
        dw.set_slice(sliceno)
        for data in datasets.source.iterate(sliceno, ...):
            ...
            dw.write(x)
```

Note the `for`-loop over `sliceno`, which controls both the reading iterator *and* the dataset writer.

Chapter 6

Iterators

The basic idea of the Accelerator's datasets is to make it really easy to create parallel programs that can read or write data at a very high speed. Accessing dataset data is done using *iterators*.

An iterator yields one `tuple` at a time containing elements from one or more specified data columns, one row at a time. In case of iterating over a single column, the output may optionally be a scalar instead of `tuple` for cleaner code and more efficient computing.

6.1 The Three Iterators

Technically, iterators are members of the `Dataset` class. Iterators can be parallel, in `analysis`, or sequential, in `prepare` or `synthesis`. There are three iterators available:

`iterate()`, for single dataset iteration,

`iterate_chain()` for iterating over dataset chains, and

`iterate_list()` for iterating over a list of datasets.

In many common use cases it is sufficient to provide only two arguments to the iterator: `sliceno` (mandatory) and `columns`. These, and all possible arguments are presented in detail shortly. A typical call may look like this

```
for m, u in dataset.source.iterate(sliceno, ('movie', 'user',)):
    # do_something_with m and u here...
```

or, if the purpose is to compose a `dict`,

```
n2d = dict(dataset.source.iterate(sliceno, ('name', 'date',)))
```

All three iterators share these arguments

name	default	description
`sliceno`	*mandatory*	Slice number to iterate over, or *None* to iterate over all slices sequentially.
`columns`	*None*	Tuple of column labels or a single name if iterating over one column. *None* selects all columns in alphabetic order.
`hashlabel`	*None*	Name of hash column. If the code relies on a dataset being hashed on a particular column, set this to make the iterator verify that this is actually the case. Execution will terminate if the hashlabel is incorrect.
`rehash`	*False*	Setting this to *True* will rehash the dataset on-the-fly based on the `hashlabel` column. (Rehashing on-the-fly is slower, so ideally datasets should be rehashed using the `dataset_rehash` method 8.4.)
`translators`	*None*	Translators transform data values. Explained in section 6.8
`filters`	*None*	Filters decide which rows to include. Explained in section 6.9
`status_reporting`	*True*	Give status when pressing `C-t`. Unless manually `zip`ing iterators, this should be set to default *True*. See `dataset.py` source code for full information.

In addition, `iterate_chain` takes these arguments too

name	default	description
`length`	−1	Number of datasets in a chain to iterate over. Default is −1, which corresponds to all datasets in a chain.
`range`	*None*	Filter rows based on a column's value being within a range, see section 6.4
`sloppy_range`	*False*	Used with `range`, but will iterate over full datasets for those datasets that have values within range, see section 6.4.
`reverse`	*False*	Iterate chain backwards. Default is to iterate forward, i.e. from oldest to newest dataset.
`stop_ds`	*None*	Iterate back to this dataset. Actually, setting this will iterate from `stop_ds` to the newest dataset in the chain.
`pre_callback`	*None*	A function that will be called before iterating each dataset.
`post_callback`	*None*	A function that will be called after iterating each dataset.

while `iterate_list` takes a `datasets` parameter

name	default	description
`datasets`	*None*	List of datasets to iterate over.

6.2 Basic Iteration

Basic use include iterating in parallel or serial over one dataset or a chain of datasets.

Parallel Iterator Invocation

For parallel iteration in `analysis`, the iterator needs to know the number of the current slice. The following is an example of iteration that happens independently in each slice.

```
datasets = ('source',)

def analysis(sliceno):
    h = defaultdict(set)
    for user, item in datasets.source.iterate(sliceno,
                                              columns=('user', 'item',)):
        h[user].add(item)
```

The program creates dictionaries mapping `users` to sets of `items` for the `source` dataset. (Assuming that the dataset is hashed (see 5.3), this operation is entirely parallel and there is no need to merge all the results from the analysis processes later.

Sequential Iterator Invocation

By specifying the `sliceno` parameter to *None*, the iterator will run through all slices of the dataset, one at a time, like in this example

```
def synthesis():
    h = defaultdict(set)
    for user, item in datasets.source.iterate(None,
                                              columns=('user', 'item',)):
        h[user].add(item)
```

Slices will be iterated one at at time in increasing order.

Iterate Over Chains

To iterate over several datasets in a chain, use `iterate_chain`. The following example will iterate over the last three datasets in the chain, oldest dataset first.

```
datasets = ('source',)

def analysis(sliceno):
    h = defaultdict(set)
    for user, item in datasets.source.iterate_chain(
                      sliceno, columns=('user', 'item',), length=3):
        h[user].add(item)
```

Using `iterate_chain` without explicitly specifying `length` will default to a `length` of -1, which corresponds to all datasets in the chain.

Here is an interesting example of a method that will iterate over all chained `source` datasets that are new since the last invocation of the method.

```
datasets = ('source',)
jobids = ('previous',)

def analysis(sliceno):
    h = defaultdict(set)
    for user, item in datasets.source.iterate_chain(
            sliceno,
            columns=('user', 'item',),
            stop_ds={jobids.previous: 'source',}):
        h[user].add(item)
```

Special Cases, Iterating Over All or a Singe Column

It is possible to iterate over all columns in a dataset by specifying an empty list of column names, like this

```
for items in dataset.source.iterate(sliceno, None):
    print(items)  # is a tuple of all columns
```

The iterator will output a `tuple` populated with all column values for each row. The columns will be in sorted column name order.

If iterating over a single column, it makes little sense to keep the output values in a one-dimensional tuple. Then, it is probably more efficient to output scalars instead. Here are the two different ways to iterate over a single column

```
# alternative 1, use lists/tuples
for user in datasets.source.iterate(sliceno, ('USER',)):
    userset.add(user[0])  # user is a tuple
```

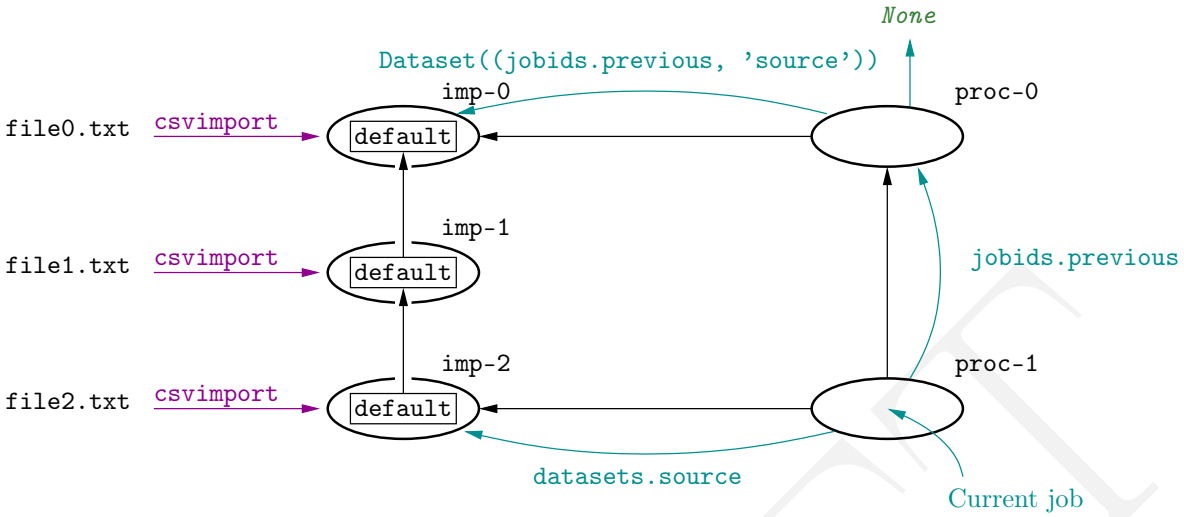


Figure 6.1: Example of import and processing jobs. Blue text and arrows relate to the *current job*, `proc-1`.

```
# alternative 2, specify column as string, not list
for user in datasets.source.iterate(sliceno, 'USER',):
    userset.add(user)      # user is a scalar!
```

Both styles are supported by filters and translators introduced later i this chapter.

An Example

Consider the case where new files are added to a project in a continuous fashion. These files are imported and chained, so that each new imported dataset links to the previously imported dataset, and so on. This is illustrated in the left part of figure 6.1.

Once in a while, but not necessarily at the same rate as new files are added, some processing of the data is performed. A simple example of a processing job could be to count the number of lines added to the dataset chain since the last execution of the processing method. A more interesting case could be to *update* a machine learning model with the newly added data imported since the last model update.

In these cases, the processing job should iterate over a part of the the dataset chain only, from the first dataset after the last processing job up to the most recent imported dataset. Again, see figure 6.1. The right part of the image illustrates two builds of the processing method. The first operates on the dataset `imp-0`, and the last on `imp-1` and `imp-2`. The input parameters to the first processing job, `proc-0`, are

```
jobids.previous = None

datasets.source = 'imp-0'
```

and the input parameters for the second processing job, `proc-1`, are

```
jobids.previous = 'proc-0'

datasets.source = 'imp-2'
```

The processing job should iterate on the `datasets.source` dataset, and set the `stop_id=` parameter to the `previous` job's `source` dataset, and the code may look something like this

```
datasets = ('source',)
jobids = ('previous',)

def analysis(sliceno):
```

```
    for ... in datasets.source.iterate_chain(
            ...
            stop_ds={jobids.previous: 'source',}):
    ...
```

6.3 Halting Iteration

Iteration over a dataset chain will continue until all data is exhausted or some stop criteria is fulfilled. There are several mechanisms for stopping, and they may be combined in a single expression. If so, iteration will be over the shortest range of the conditions.

Halting Using `length`

```
for user, item in datasets.source.iterate_chain(
    sliceno, ('user', 'item',),
    length = options.length):
```

This will iterate for the last `options.length` number of datasets. Note that a length of -1 is default and will iterate without bounds.

Halting Using `stop_ds`

Similar to using `length`, but will stop when reaching a certain dataset.

```
for user, item in datasets.source.iterate_chain(
    sliceno, ('user', 'item',),
    stop_ds = 'foo-3'):
```

Stopping at a constant dataset has limited value. Next section shows how to stop iterating based on previous jobs.

Halting Using Another Job's Input Parameters

```
for user, item in datasets.source.iterate_chain(
    sliceno, ('user', 'item',),
    stop_ds = {jobids.previous: 'source',}):
```

This will iterate until reaching the `source` dataset of the `jobids.previous` job.

6.4 Iterating Over a Data Range

It is possible to iterate over rows having a specified column's value within a certain range. This works best on datasets that are sorted on the specified column.

```
for user, item in datasets.source.iterate_chain(
    sliceno, ('user', 'item',),
    range={timestamp, ('2016-01-01', '2016-01-31'),}):
```

This example will limit the iterator to exactly the range of lines that fulfill the range condition. It is relatively costly to filter each line, and there is a speed advantage by instead specifying `sloppy_range`, which will iterate over all datasets that contain part of the range:

```
for user, item in datasets.source.iterate_chain(
    sliceno, ('user', 'item',),
    sloppy_range={timestamp, ('2016-01-01', '2016-01-31'),}):
```

Here, all datasets that *contain* any line containing values within the range will be included in the iteration. Still, if the datasets are sorted, and there are many datasets, there will be a limited side-effect caused by reading too many lines, and in many cases this could be remedied with little effort.

6.5 Iterating in the Reverse Direction

By default, iterating over a chain of dataset starts at the oldest dataset and ends at the latest dataset. This behavior can be reversed by specifying `reverse=True`. But note that row iteration is still in the forward direction within each dataset!

```
for user, item in datasets.source.iterate_chain(
    sliceno, ('user', 'item',),
    reverse=True):
```

6.6 Hashed Datasets and on-the-fly Rehash

Hashing a dataset on a particular columns, see section 5.3, may really simplify the parallel programming of methods using the dataset. however, such parallel code will not work properly if it turns out that the input data is not hashed in the expected way. For that reason, it is a good idea to *assert* the hashlabel by entering it into the iterator function, like this

```
s = {user: item for user, item datasets.source.iterate_chain(
     sliceno, ('user', 'item',), hashlabel='user')}
```

and execution will terminate if the hashlabel is not correct.

It is possible to rehash the dataset on-the-fly. This is done by setting the `rehash` argument to the iterator to `True`, like this

```
for user, item in datasets.source.iterate_chain(
        sliceno, ('user', 'item',),
        rehash='item'):
    # only lines with items such that
    # has(item) % slices == sliceno here
```

While this works, the preferred way to rehash is to use the `dataset_rehash` method 8.4, since it will store the rehashed dataset for later use, which in most scenarios will be more efficient.

6.7 Callbacks

The iterator may be assigned callback functions that are called before starting iterating a new dataset, and after the current dataset is exhausted. Callbacks are useful for example to aggregate data by dataset when iterating over a dataset chain.

There are two independent callbacks for these two cases, called `pre_callback` and `post_callback`. If `sliceno=None`, i.e. iteration runs over all slices of all datasets in one process, it is even possible to have callback between slice changes.

The example below will print the dataset identifier for each dataset prior to iterating over it.

```
# pre_callback once per dataset
def prefun(dataset_id):
    print(dataset_id)

for user, item in datasets.source.iterate(sliceno, ('user', 'item',),
                                          pre_callback=prefun):
    ...
```

The argument to the callback is the dataset id to the dataset to be iterated next.

Next is an example of an iterator running over all slices. The callback function is executed before each new slice is iterated. The callback takes two arguments in this scenario, first, the dataset id as per the example above, and second the number of the slice.

```
# callback once per slice
def prefun(dataset_id, sliceno):
    print(dataset_id, sliceno)

for user, item in datasets.source.iterate(None, ('user', 'item',),
                                          pre_callback=prefun):
    ...
```

The `post_callback` function is defined similarly.

Skipping Datasets and Slices from Callbacks

It is possible to skip dataset iterations by raising exceptions, as follows.

- To skip the next dataset do

  ```
  raise SkipJob
  ```

- To have the iterator skip a slice, do a

  ```
  raise SkipSlice
  ```

- And to abort iterating completely

  ```
  raise StopIteration
  ```

  In this case, a `post_callback` will never be run.

6.8 Translators

Translators transform iterator output data values on-the-fly. A translator is either a callable or a `dict`. Translators are similar to *filters* (explained later), and always executed *before* filtering. The idea behind translators and filters is to provide a way to modify code behavior by supplying functions as options to iterators. Using translators and filters, it is possible to write re-useable functions that can have different behaviour depending on context.

Callable Translator, translating `tuples`

A translator function is a function from an input `tuple` (of column values) to an output `tuple` of the same length. Individual items may be passed through or modified, and it is possible to mix different columns with each other before sending them to the iterator output. Here is an example

```
def merger(user, item):
    return ("%s:%s" % (user, item), None)

for merge, _ in datasets.source.iterate_chain(None, ('user', 'item',),
                                              translator=merger):
    ...
```

The purpose of this translator is to convert each `(user, item)` tuple to a string `user:item`. This is the first output of the translator and iterator and is stored in the `merge` variable. The second output variable is not used in this application, but a variable still has to be assigned, so it is set arbitrarily to `None`.

Translator `dict`, translating columns independently

One or more columns may be translated independently using a translator dictionary. Such a dictionary is specified as {`name`: `translation`}. A translation may be either a `dict` or a callable. Examples of both kinds are shown below. First an example illustrating the use of a translation `dict`. Here, integers are translated into more comprehensible strings.

```
mapper = {2: 'HUMANLIKE', 4: 'LABRADOR', 5: 'STARFISH',}
for animal in datasets.source.iterate(None, 'NUM_LEGS',
                                      translator={'NUM_LEGS': mapper,}):
    ...
```

This translator will substitute the integers 2, 4, and 5 into strings. Items missing in a translation-`dict` yield *None*. The next example illustrates a callable inside a translator `dict`. In this example, each `user` string is output from the iterator reading right-to-left.

```
def reverse(x):
    return x[::-1]
for resu, item in dataset.source.iterate(None, ('USER', 'ITEM',),
                                         translator={'USER': reverse,}):
    ...
```

The `item` values will be passed through, but the `user` strings will be reversed.

6.9 Filters

Filter are used to decide which output data rows that are allowed to reach the iterator output. Filters are run *after* translators. As for translators, filter are either callables or `dicts`.

Callable Filter, filtering `tuples`

Callable filters receive the iterator `tuple` as input. The output of a filter must be *True* for the `tuple` to be output from the iterator, otherwise the iterator skips and continues to the next row. The following two examples will iterate over all animals that have at least two legs and a trunk. The first example is without filter, using an `if`-statement, and the second example uses iterator `filters`.

```
# Ex. 1. Using if-statement
for animal, nlegs, wtrunk in datasets.source.iterate(None,
        ('animal', 'num_legs', 'has_trunk'), filters=filter):
        if nlegs>=2 and wtrunk:
    ...

# Ex. 2.  Using iterator filters
filter = lambda line: line[1]>=2 and line[2]

for animal, nlegs, wtrunk in datasets.source.iterate(None,
        ('animal', 'num_legs', 'has_trunk'), filters=filter):
    ...
```

Note the indexing in the `lambda` function. Index zero corresponds to the `animal` column, which is not includes in the filtering expression.

Filter Dict, filtering independent columns

It is possible to filter on one or more columns independently using a `dict`. If there is more than one filter, all filters must be *True* for a line to be output from the iterator. Below are two examples of filter `dicts`. The first example will remove all rows except the ones with valid users.

```
# keep valid users only
validusers={'user1', 'user2', 'user3'}
filters={'user': validusers.__contains__}
```

The second example will only keep rows with valid users and movie items.

```
# keep valid users with movie items
validusers={'user1', 'user2', 'user3'}
validitems={'movie': True, 'book': False}
filters={'user': validusers.__contains__, 'item': validitems.get}
```

The fact that book is *False* is actually redundant, since missing keys will never evaluate to *True* and thus result in discarded lines.

Filter by Column Values

Filtering could also be by column value directly. For example, assume there is a column `has_trunk` with values being Boolean integers, i.e. `1` or `0`. Animals with trunks may be iterated using

```
for animal, wtrunk in datasets.source.iterate_chain(None,
                      ('animal', 'has_trunk',),
                      filters={'has_trunk': None}):
    trunked.add(animal)
```

This may seem strange at first, but it works because the key for the `has_trunk` column exists, and the value is *None*, which is neither a callable nor a `dict`.

Chapter 7

High Level Control: Urd

This chapter is the continuation of chapter 3, "Basic Build Scripting". Please read about build script and joblists before proceeding.

7.1 Introduction to Urd

Urd comes into play when simple build scripting is not enough. A wide variety of advanced tasks can be handled without it, but the capabilities added by Urd in terms of job organisation, storage, and retrieval makes it possible to handle much larger and more advanced projects while maintaining in full control.

Using Urd, a project could be separated into functionally independent parts, and all dependencies between jobs inside as well as between these parts is logged. It is possible to re-construct the state of any part the way it were at any instance in time.

More formally, Urd provides two things:

1. Separation between build scripts, and a way to share information about built jobs between different scripts (or the same script at different points in time).

2. A searchable transaction log database of all jobs built together with their dependencies.

The interesting transaction log database and many other aspects of Urd will be explained in the rest of this chapter.

7.2 A Simple Use case

Assume a project where, say, movie recommendation data is to be analysed. Every hour, recommendations generated during the last hour will appear in the shape of a new log file. The project is using two build scripts:

> The first build script is used to look for new files, and import and chain them as they appear. For each new file imported, the build script will tell the Urd server the timestamp of the file as well as a list of all created jobs associated with that file.
>
> The second build script is used for the data analysis work, and is perhaps run less regularly. This script needs to know the jobid to the latest imported file, and this is a straightforward thing to ask Urd. All analysis jobs are also stored in Urd together with a corresponding timestamp.

Here, Urd is used to forward information about executed jobs from first build script to the second. In this sense, Urd provides *isolation* by message passing.

Urd can also be used to tell which input data that was used for a particular data analysis job. When querying Urd about a data analysis job, it will respond with information about those jobs *as well as* information about all Urd queries that was necessary for the jobs to run. This information is stored automatically in the build script and it is there to ensure transparency and reproducibility.

7.3 Urd Sessions and Lists

A simple file import script will be used as example in this section:

```
def main(urd):
    urd.build('csvimport', options=dict(filename='txn1.txt'))
```

In order to use this import job in a future context, a *session* is created by wrapping the code by the `urd.begin()` and `urd.finish()` functions, like this

```
def main(urd):
    urd.begin('import/txn', '2018-05-03')
    urd.build('csvimport', options=dict(filename='txn1.txt'))
    urd.finish('import/txn')
```

Everything that happens between `begin()` and `finish()` makes up the session. The `finish()` function makes sure that the session is stored permanently to disk for future reference. In this case, the session can be retrieved knowing its *list* name

```
import/txn
```

and its *timestamp*

```
2018-05-03
```

The list identifier is composed of two parts, `<user>/<list>`, where `<user>` is for authorisation purposes. Each user can have any number of lists, but only the correct user may write to them, as will be explained later.

The next sections will explain how to search the Urd database for matching sessions in various ways.

7.4 A First Urd Query

The list created in the previous section can now be used by other build scripts. For example, here is a build script that does some processing on the previously imported file

```
def main(urd):
    urd.begin('process/test')

    import_session = urd.latest('import/txn')

    import_timestamp = import_session.timestamp
    import_jobid     = import_session.joblist.jobid

    urd.build('process', datasets=dict(source=import_jobid))

    urd.finish('process/test', import_timestamp)
```

The first thing that happens is that all processing is covered in a session named `process/text`. At `begin()`, the timestamp is still unknown, it is to be set to the same timestamp as the import job has.

The script is then retrieving the recently created urd session stored in list `import/txn`. Two things are extracted from this data, the *timestamp* and the *joblist*. The timestamp will be used for this session as well, to indicate that processing is based on data with that particular timestamp. The *jobid* to the `csvimport` job is then extracted from the joblist and fed to the `process` job as an input dataset parameter. (The slightly clumsy syntax for this is explained in section 3.2.2.)

7.5 The Contents of the Stored Session

Calling `urd.finish()` will update the Urd database with the contents of the current *session*. Each session is addressable using a *list* name (in the format `<user>/<list>`) and a *timestamp*. Session data is stored internally in the `json` format, and in build scripts it appears as Python `dicts`. The example presented earlier in this chapter may have been recorded similarly to this

```
{
    "user": "processing",
    "automata": "test",
    "timestamp": "2018-05-03",
    "caption": "",
    "joblist": [
        [
            "process",
            "TEST-37"
```

```
        ]
    ],
    "deps": {
        "import/txn": {
            "timestamp": "2018-05-03",
            "caption": "",
            "joblist": [
                [
                    "csvimport",
                    "TEST-34"
                ]
            ],
        }
    },
}
```

(This example states that at timestamp `2018-05-03` in list `processing/test`, there exists a `process` job with jobid `TEST-34` that used a `csvimport` job with jobid `TEST-34`. This job also exists in the urd list `import/txn` at timestamp `2018-05-03`.)

The most important keys are

name	description
`timestamp`	Timestamp of session
`caption`	A caption
`user/automata`	Name of Urd list
`joblist`	An object of type `joblist`, containing all jobs built in the session. For more information, see 3.2.
`deps`	A dictionary of dependencies from `user/automata` to urd sessions: `{'user/automata': session}`.

7.6 Urd Sessions: `begin()` and `finish()`

There are a number of options associated with a session, as shown here,

```
urd.begin(urdlist, timestamp, caption=None, update=False)
urd.finish(urdlist, timestamp, caption=None)
```

and the following applies

name	description
`urdlist`	is the name of the Urd list, and the same `urdlist` must be specified in both `begin()` and `finish()`. The `urdlist` is specified as `<user>/<list>`, where the `<user/>` part is optional. The `user` string is also for authentication, and must correspond to the current `URD_AUTH` settings, see section A.2.
`timestamp`	is *mandatory*, but could be set in either `begin()`, `finish()`, or both. `finish()` will override `begin()`.
`caption`	is *optional*, and can be set in either `begin()` or `finish()`. `finish()` will override `begin()`.
`update`	If set to *`True`*, the last item in the list may be updated. This option will be discussed in section 7.12.

The Urd transaction database will be written to only when the `finish()` function is called. Before calling `finish()`, nothing is stored, and it is perfectly okay to omit `finish()` to avoid storage or during development work.

7.6.1 What if a Build Script is Run Again?

Running a build script for the first time will cause jobs to be built. The second time the same script is run, the Accelerator will look up already built jobs and immediately return jobids instead of building anything. As long as there are no changes, re-defining Urd sessions that already exists is not a problem - they will be silently ignored. But if there are any discrepancies, such as a job being rebuilt, Urd will complain and refuse to store the differing session.

Normally, a build script can be written in such a way that re-running it will be consistent will the Urd database and everything is fine. A mismatch with Urd is then an indication of some error. But there are cases when re-writing Urd history is the desired option, and this will be discussed in section 7.12.

7.7 Timestamp Resolution

Timestamps may be specified in various resolution depending on the application. The full time format is (See Python's `datetime` module for explanation.)

```
'%Y-%m-%dT%H:%M:%S'
```

A specific timestamp could be shorter than the above specification in order to cover wider time ranges. The following examples cover all possible cases.

```
'2016-10-25'                # day resolution
'2016-10-25T15'             # hour resolution
'2016-10-25T15:25'          # minute resolution
'2016-10-25T15:25:00'       # second resolution
```

7.8 Finding Items in Urd

There are several ways to find stored sessions. This section will describe the `get()`, `first()`, and `latest()` function calls. For any of these calls to work, they have to be issued from

within a session, i.e. after a `begin()` call. Otherwise Urd would not be able to record all session dependencies.

More ways to find sessions is described in section 7.14.

7.8.1 Finding an Exact or Closest Match: `get()`

The `get()` function will return the single session, if available, corresponding to a specified list and timestamp, like this

```
urd.begin('ab/anotherlist')
urd.get("ab/test", "2018-01-01T23")
```

The timestamp must match exactly for an item to be returned. This strict behaviour can be relaxed by prefixing the timestamp with one of

"<", "<=", ">", or ">=".

For example

```
urd.get("ab/test", ">2018-01-01T01")
```

may return an item recorded as `2018-01-01T02`. Relaxed comparison is performed "from left to right", meaning that

```
urd.get("ab/test", ">20")
```

will match the first recorded session in a year starting with "20", while

```
urd.get("ab/test", "<=2018-05")
```

will match the latest timestamp starting with "`2018-05`" or less, such as "`2018-04-01`" or "`2018-05-31T23:59:59`".

If there is no matching item, the `get()`-call will return an *empty session*, i.e. something like this

```
{'deps': {}, 'joblist': JobList([]), 'caption': '', 'timestamp': '0'}
```

7.8.2 Finding the Latest Session: `latest()`

The `latest()` call will return the session with most recent timestamp. Example

```
urd.begin('ab/anothertestlist')
urd.latest('ab/test')
```

will return a complete item like this

```
{'automata': 'test', 'caption': '', 'user': 'ab', 'deps': {}, \
 'joblist': JobList([('example', 'TEST-34')]), 'timestamp': '2018-05-06'}
```

If the list is non-existing or empty, an *empty session* will be returned.

7.8.3 Finding the first item: `first()`

The `first()` function works similarly to `latest`, but will instead return the session with the *oldest* timestamp.

7.9 Aborting an Urd Session: `abort()`

When an Urd session is initiated, a new session cannot be started until the current session has finished. A session may therefore be aborted, and the `abort()` function is used for this, like so

```
urd.begin('test')
urd.abort()
```

Similar to unfinished sessions, aborted sessions will not be stored in the Urd transaction log.

7.10 Building Jobs: `build()`

Jobs are dispatched in Urd sessions using the `build` function. Here is the complete call with all possible parameters.

```
jobid = urd.build(
    method,
    options={},    datasets={},    jobids={},
    name='',       caption='',
    workdir=None
)
```

Explanation of `build` parameters:

name	description
`method`	Name of method to build. Enter `test` here if the method filename is `a_test.py`.
`options={}`	a `dict` of options to the method. This overrides options defined in the method itself, but adding options not prototyped in the method is *not* allowed.
`jobids={}`	a `dict` of jobids to the method. It is possible to specify a list of jobids like this `jobids=dict{alljobs=[jobid1, jobid2,...]}`
`datasets={}`	a `dict` of datasets to the method. Datasets may be lists too, just like `jobids` above.
`workdir=None`	If specified, the job will be built in this workdir, assuming the workdir is specified in the configuration file as either source or target.
`name`	A string associated with the job. Use it to distinguish several jobs created from the same method.
`caption`	A caption string. For decorative purposes only, this has no practical use.

The `build()` function will only build a job when it has to, otherwise it will just return a link (jobid) to a matching existing job. In order to match, an existing job must have

- exactly the same source code, i.e. the *hash* of the source code must match,
- exactly the same options, datasets, and jobids.

If the source code is changed, a job rebuild can be prevented using the `equivalent_hashes` variable as explained in section 4.6.

7.11 Changing workdir: `set_workdir()`

The target workdir specified in the configuration file is the only workdir that is written to by default. Any other workdir is read only. This behaviour can be overridden, either

per job, using the `workdir=...` option to `urd.build` as shown in section 7.10, or

using `urd.set_workdir()`.

The latter,

```
def main(urd):
    urd.set_workdir(<workdir>)}
```

will set the workdir for all coming `build` calls in the current build script. It can still be overridden using the `workdir=` option to `urd.build`.

7.12 Truncating and Updating

Since the Urd database is designed using log files, it will always keep a consistent history of all events taken place. It is not possible to erase or modify old entries, but it is okay to update the latest item, or set a marker in the log indicating that the list is starting over from a certain date and everything before this marker should not be considered anymore. This makes it possible to both keeping the full history *and* being able to rewrite it. There is full transparency and reproducibility – all sessions before an update or restart marker are always kept in the Urd log file.

7.12.1 Updating the last item

To update the last item in a list, set the `update` argument to *True*

```
urd.begin('test', '2016-10-25', update=True)
```

If update is *True*, the entry in the test list at '2016-10-25' will be updated, unless the new information is equivalent. The `update()` call will simply add a new line to the Urd log database, and if the timstamp is the same as the previous entry, the new entry will be selected.

7.12.2 Truncating a list

In order to insert a marker in the database indicating that everything before a certain timestamp should be discarded, use the `truncate()` function like this

```
urd.truncate(ab/'test', '2016-09-30')
```

This will rollback everything that has happened in the `ab/test` list back to '2016-09-30'. There is also a special case,

```
urd.truncate('ab/test', 0)
```

that will erase all items from memory and cause the list to start over again. Remember, internally Urd stores the complete history in a log file in plain text. Files can only be appended to, nothing is ever removed. It is always possible to recover any old result or processing state.

7.12.3 Truncation Consequences: Ghosts

When a list is truncated, all items after a specified timestamp are made invisible. Assuming that another list has stored a dependency of an item that is truncated, the jobs in this list are now without dependencies that can be looked up. We call them "ghosts". Ghosts cannot be looked up in Urd, but they are still in the database, marked as ghosts.

7.13 Avoiding Recording Dependency

Dependency-recording will be activated on use of the `get()`, `latest()`, and ()first functions. If, for some reason, the point is to just have a look at the database to see what is in there, it can be done using the *peek* functions, `peek()`, `peek_first()`, and `peek_latest()`, like this:

```
urd.peek('test', '2016-10-25')
urd.peek_latest('test')
urd.peek_first('test')
```

Note that this is in general not recommended. These functions will look up Urd lists with jobids that may be used to build new jobs, but these dependencies will not be stored in the current Urd session, causing a loss of continuity and visibility.

7.14 More Search Functions

There are two more functions for finding information in the Urd database: `list` and `since`.

7.14.1 Listing all urd lists: `list()`

The `list()` function will return a list of all lists recorded in the database:

```
print(urd.list())
```

may show something like

```
['ab/test', 'ab/live']
```

7.14.2 Listing all Items After a Specific Timestamp: `since()`

The `since()` function is used to extract lists of *timestamps* corresponding to recorded session. In its most basic form, it is called with a timestamp like this

```
urd.since('2016-10-05')
```

which returns a list with all existing timestamps more recent than the one provided

```
['2016-10-06', '2016-10-07', '2016-10-08', '2016-10-09', '2016-10-09T20']
```

The `since` is rather relaxed with respect to the resolution of the input. The input timestamp may be truncated from the right down to only one digits. An input of zero is also valid. For example, these are all valid

```
urd.since('0')
urd.since('2016')
urd.since('2016-1')
urd.since('2016-10-05')
urd.since('2016-10-05T20')
urd.since('2016-10-05T20:00:00')
```

7.15 The Urd HTTP-API

Urd can be accessed directly without using the Accelerator by calling its HTTP API. These calls are easy to use and improve transparency since the database contents can be peeked without writing a program. Here is a list of all API endpoints

```
/list
/<user>/<list>/first
/<user>/<list>/latest
/<user>/<list><timestamp>
/<user>/<list>/since/<timestamp>
```

All calls return data in the `json` format. In the following the standard tool `curl` is used, and it is assumed that there is an Urd server running on `localhost`, port `8833`.

7.15.1 The `list` endpoint

To show all stored lists issue

```
% curl http://localhost:8833/list
["ab/test"]
```

All available lists are returned in a json list.

7.15.2 The `since` endpoint

The `since` endpoint is used to get a list of all entries more recent than a timestamp. Fr example, to see what is more recent than `2016-10-24` do

```
% curl http://localhost:8833/ab/test/since/2016-10-24
["2016-10-25"]
```

```
% curl http://localhost:8833/ab/test/since/2016-10-26
[]
```

Results are in json list format.

7.15.3 The `first` and `latest` endpoints

Looking up the latest stored job in the `ab/test` list

```
% curl http://localhost:8833/ab/test/latest
{"caption": "", "automata": "test", "user": "ab", "deps": {},
  "timestamp": "2016-10-25", "joblist": [["method1", "test-56"],
  ["method2", "test-59"], ["method3", "test-60"]]}
```

And see the first stored job in the test list

```
% curl http://localhost:8833/ab/test/first
```

works similarly. The returned data is an Urd item, described in section 7.5, in json format.

7.15.4 The `get` endpoint

Actually, there is no explicit `get` endpoint. Instead, the API is just called by the name of the list and a timestamp. For example, to see what is inside the test list stored at `2016-10-25`

```
% curl http://localhost:8833/ab/test/2016-10-25
{"caption": "", "automata": "test", "user": "ab", "deps": {},
  "timestamp": "2016-10-25", "joblist": [["method1", "test-56"],
  ["method2", "test-59"], ["method3", "test-60"]]}
```

The timestamp may be truncated to the right, and prefixed by `>`, `>=`, `<`, and `<=`, just as described in section 7.8.1. Make sure to quote the request if these characters are used in a call from the shell.

7.16 Profiling a Build Script: `print_profile()`

There is a helper function in the urd object that can be used to print profiling information. The following example is self-explanatory

```
def main(urd):
    ...
    urd.print_profile()
```

This will print execution times for all jobs in the session to `stdout`. It may for example look like this

```
Time per method:
   color2        23.7 seconds  (25%)
   csvexport     17.5 seconds  (18%)
   lowpass2      15.6 seconds  (17%)
   newcol        14.4 seconds  (15%)
   black          5.7 seconds  (6%)
   colimage       5.4 seconds  (6%)
   sync           4.7 seconds  (5%)
   clamp          3.8 seconds  (4%)
   dataset_type   2.7 seconds  (3%)
   csvimport      1.4 seconds  (1%)
Total time 94.8 seconds
```

The methods are sorted by execution time, top to bottom.

7.17 Passing Flags from the Command Line

It is possible to add a comma separated list of flags to the runner like this

```
./automatarunner.py --flags=verbose,skiptest
```

The flags will appear in the `urd`-object like this

```
def main(urd):
    if 'verbose' in urd.flags:
       print('verbosity')
```

7.18 Urd Internals

Urd can be accessed by a large number of clients. Each client may add to or truncate any list at any time. In order to avoid race conditions and make the database deterministic, all `add`- and `truncate`-requests appears in a sequential manner to the Urd server. Each request is assigned with an unique timestamp, and stored in the requested list.

When Urd is restarted, it reads all the database files, and sorts all rows in order of the receive timestamp. Thereafter, each row is applied in increasing time order to the internal, RAM-based database. Due to the unique timestamping, the result is a deterministic replica of the previous run.

Chapter 8

Standard Methods

The Accelerator is shipped with a set of common standard methods. These are found in the method directory `./standard_methods`.

8.1 `csvimport` – Importing Data Files

The `csvimport` method is used to import a text files into a dataset. The method can be chained, so any number of text files can be connected in a dataset chain. Input data is assumed to be in a tabular format, i.e. it is composed of a number of rows, each having the same number of columns separated by a separator token. A common format of this type is the Comma Separated Values (CSV) format, but `csvimport` is much more flexible, as seen in the table of options below. For example, `csvimport` can handle any separator character, skip or parse labels on the first line, and supports advanced quote support. It also deals with "broken" input data in a predicted and user controlled way.

8.1.1 Options

name	default	description
`filename`	*mandatory*	Name of file to import. The filename is mandatory and the file may either be a plain text file or a gzipped file. It is also possible to specify a filename including a path. If the path begins with a slash, it is absolute. Otherwise, the path is relative to the `source_directory` configuration parameter specified in the configuration file, see section A.1. A relative path makes it possible to relocate files to a different directory without trigging job remake.
`separator`	","	Field separator. Any character, except "\0", "\n", and "\r" will work.
`labelsonfirstline`	`True`	If set to *True*, data on the first line of the file will be used as column labels. If *False*, labels must be entered using the `label` option, see `labels` below.
`labels`	`[]`	If `labelsonfirstline` (see above) is set to *False*, labels must be provided using this option. For example `labels = ['foo', 'bar',]`.
`hashlabel`	*None*	If not specified, the dataset will be created "round-robin", so that rows in the input file will be separated evenly into the dataset slices. If a valid label is specified, each input data row will be allocated to a slice depending on the output of a hash function applied to the columns data. *Note that typing of a dataset typically changes how the data is represented, which most likely voids hashing. Use this option only for datasets that will not be typed. Otherwise, use `dataset_rehash` after typing.*
`quote_support`	*False*	If set to *True*, it is possible to read CSV files with quoted values, such as `'foo'` and `"bar"`.
`rename`		This option makes it possible to change the column names read from the first line of the input file. Renaming happens first. It accepts a dictionary of type `{old_name: new_name,}`.
`discard`	`set()`	labels in the discard set will not be stored in the dataset.
`allow_bad`	*False*	Relaxes the input file strictness if set to *True*. If set to *False*, which is the default, `csvimport` will assert an error if there are format errors in the input data. Setting it to *True* makes importing silently ignoring any format errors. It is recommended to check the resulting dataset if enabling this option!

8.1.2 Datasets

name	default	description
previous	*None*	Previous dataset if creating a chain.

8.1.3 Output

The result of the `csvimport` is a dataset named `default`. All columns will be of type `bytes`. Typically, the dataset from `csvimport` is fed to a `dataset_type` job for column typing.

8.1.4 Example Invocation

An example invocation is the following

```
urd.build(csvimport',
    options=dict(
        filename='inputfile.txt',
        separator='\0',
    )
)
```

this will import the file `inputfile.txt` assuming that there are labels on the first line and the column separator is a null character (`0x00`, `'\0'`).

8.2 dataset_type – Typing Datasets

The dataset_type method will read a source dataset and create a new dataset that is typed. This is primarily used for typing datasets created by csvimport.

Default behaviour is to append new columns with typed data. These columns will have the same name as the untyped version of the data, making the untyped data "inaccessible", even if it is still in the dataset. Using the rename options, typed columns could be assigned a name that differs from the original columns, so that both typed and untyped data is available simultaneously. This brings transparency to the typing process. (But even if the untyped data is "inaccessible" in the typed dataset, it is still available from the input dataset.)

In order to type the data, it is subject to parsing. Some datasets may have contain data that is incorrect in the sense that it causes parsing errors when typing. Unparseable data could either be replaced by a default value or removed from the dataset. Since the dataset type does not permit removal of rows, dataset_type will create a new dataset containing only the rows containing typeable data.

8.2.1 Datasets

name	default	description
source	*mandatory*	Dataset to type.
previous	*None*	Previous dataset if creating a chain.

8.2.2 Options

name	default	description
column2type		A dictionary from column label to type, for example {'movie': 'unicode:UTF-8',}.
defaults		A dict from column name to default value, for example {'COLNAME': value}. Method will fail if data is unconvertible unless filter_bad = *True*.
rename		A dictionary from old name to new name, for example {'old': 'new'} The old name and data will be preserved, unless a new dataset is created , and the column with the new name will contain the typed data.
caption	*empty string*	A caption.
discard_untyped	*None*	Force creation of new dataset.
filter_bad	*False*	remove rows containing untypeable data. Will create new dataset.
numeric_comma	*False*	If *True*, write decimal number as "3,14" instead of default "3.14".

8.2.3 Example Invocation

An example invocation is the following

```
urd.build('dataset_type',
    datasets=dict(
        source=...,
        previous=...,
    ),
    options=dict(
        column2type=dict(
            auct_start_dt='datetime:%Y-%m-%d',
            brand='json',
            item_id='number',
            comp='unicode:utf-8',
        ),
    )
)
```

8.2.4 Typing

This section describes all typing possibilities in detail. Default behaviour when typing numbers (i.e. `floats`, `ints`, and `numbers`) is that any number of whitespaces before and after the actual number are silently discarded.

Numbers

The `number` type is integer or floating point.

`number`	`int` or `float`
`number:int`	`int`, will convert `floats` to `ints`.

Integers are enforced using `number:int`, and the type accepts trailing decimal zeroes like `7.0`, `4.000` etc. This is useful when typing datafiles where numbers actually are integers but have trailing zero decimals.

Floating Point Numbers

Floating point numbers may be stored as 32 or 64 bits. In addition, there are six parsing options that are useful in different scenarios. The *ignore* option ignores any trailing characters after the number. Then there are *exact* that causes error if the number does not fit, and *saturate* that silently saturates a non-fitting number. These can also be used in combination, see table below for all alternatives

`float32`	`float64`	*default*
`float32i`	`float64i`	*ignore*, will discard trailing garbage
`float32e`	`float64e`	*exact*, error if parsed number does not fit in type
`float32s`	`float64s`	*saturate*, saturate to min/max if number does not fit in type
`float32ei`	`float64ei`	*exact* + *ignore*
`float32si`	`float64si`	*saturate* + *ignore*

Integers

Integers are stored as either 32 or 64 bits. Parsing takes base into account, so in addition to decimal numbers, it is also straightforward to parse octal and hexadecimal numbers. The *ignore* option causes parsing to ignore trailing garbage characters.

`int32_0`	`int64_0`	*auto*, avoid and use a deterministic type if possible
`int32_0i`	`int64_0i`	*auto*, ignore trailing garbage
`int32_8`	`int64_8`	*octal*
`int32_8i`	`int64_8i`	*octal*, ignore trailing garbage
`int32_10`	`int64_10`	*decimal*
`int32_10i`	`int64_10i`	*decimal*, ignore trailing garbage
`int32_16`	`int64_16`	*hexadecimal*
`int32_16i`	`int64_16i`	*hexadecimal*, ignore trailing garbage

Integers Stored as Floats

There are also a parsing options for integers that are represented in a floating point format in the source data. This is useful if integer data is stored with decimals, such as `5.0`. In pseudocode, the parsing basically runs `int(float(value))` for each such value.

`floatint32e`	`floatint64e`	*exact*, error if parsed number does not fit in type
`floatint32s`	`floatint64s`	*saturate*, saturate to min/max if number does not fit in type
`floatint32ei`	`floatint64ei`	*exact* + *ignore*
`floatint32si`	`floatint64si`	*saturate* + *ignore*

Convert to Boolean

It is common that a column holds values that are to be interpreted as either `False` or `True`. The following types handles strings and floats.

`strbool`	*`False`* if value in (*`False`*, `0`, `f`, `no`, `off`, `nil`, `null`, “”) *`True`* otherwise
`floatbool`	*`True`* when float has bits set. Is *`False`* otherwise.
`floatbooli`	same + *ignore*

Time and Date

There are three types relating to time available, `date`, `time`, and `datetime`. Each of these has a corresponding version that ignores trailing garbage characters. All time types require a format specification as described below

`date:*`	a date with format specifier
`datei:*`	same + *ignore*
`time:*`	a time with format specifier
`timei:*`	same + *ignore*
`datetime:*`	a date + time with format specifier
`datetimei:*`	same + *ignore*

The format is standard Python time formats, like shown in these examples

```
# will match for example '2017-03-22'
auct_start_dt='date:%Y-%m-%d'
# will match for example '183000', i.e. half past six in the evening
tod='time:%H%M%S'
# will match for example '2017-03-22 18:30:15'
timestamp='datetime:'%Y-%m-%d %H:%M:%S'
```

Strings and Byte Sequences

There are a number of ways to read string and byte data, depending on how the raw input data is to be interpreted. The basic types are shown first, and the more advanced variations and options will be described below.

`bytes`	list of bytes
`bytesstrip`	list of bytes, strip characters 8-13, 32 from start and end
`ascii`	list of ascii characters
`asciistrip`	list of ascii characters, strip characters 8-13, 32 from start and end

When typing to unicode and ascii, there are several ways to handle individual unparsable characters. For unicode, there are two types,

`unicode:*`	list of unicode characters
`unicodestrip:*`	list of unicode characters, strip characters 8-13, 32 from start and end

The asterisk represents options that take the form

```
"codec" #or
"codec/errors"
```

`unicode:codec/errors` will read bytes encoded in `codec` and write "unicode" (which is stored as utf-8, but that's invisible to the Python side). `codec` is often `utf-8`, but could be for example `utf-8`, `ascii`, `iso-8859-1`, `iso-8859-15`, `cp437`, or `windows-1252` etc. See the Python documentation

`https://docs.python.org/2/library/codecs.html#standard-encodings`

for more information. The `errors` part is optional, and can be one of

`strict`	The default, an error marks this row as bad
`ignore`	All unparsable bytes are discarded.
`replace`	All unparsable bytes are replaced by the unicode replacement character (`"\ufffd"`).

Using `strict` will cause errors if unparsable. For example, typing the string `"ab\xffc"` will give an error (`strict`), `"abc"` (ignore), or `"ab\ufffdc"` (replace).

Ascii is similar, there are two types

`ascii:*`	list of ascii characters
`asciistrip:*`	list of ascii characters, strip characters 8-13,32 from start and end

where the argument is one of

`strict`	The default, an error marks this row as bad
`ignore`	All unparsable bytes are discarded
`replace`	All unparsable bytes are replaced by an octal escapes `"\ooo"`
`encode`	Like replace except `"\"` is also replaced by `"\134"` (for full reversibility).

Using `strict` will cause errors if unparsable.

8.3 `csvexport` – Exporting Text Files

The `dataset_export` method is used to export datasets to column based text files (CSV, Comma Separated Values). It can export plain files and gzip-compressed files, export a chain of datasets, export one output file per slice, and more. Read the Options section for full details.

Options

name	default	description
`filename`	*mandatory*	Name of output file. File will by default be stored in the job's job directory. The filename has to end with "`.csv`" for plain text files, and "`.gz`" for gzipped output.
`separator`	','	Column separator.
`labelsonfirstline`	*`True`*	If *`True`*, write column names on first row.
`chain_source`	*`False`*	If *`True`*, read a dataset chain from `datasets.source` back to `jobids.previous`
`quote_fields`	*empty string*	Export quoted fields. Must be empty (no quote character, default), "`'`", or "`"`".
`labels`	`[]`	Specify which labels to export. An empty list corresponds to all labels in dataset.
`sliced`	*`False`*	Each slice is exported in a separate file when *`True`*. If so, use `"%02d"` or similar in filename as placeholder for the slice number.

Datasets

name	default	description
`[source,]`	*mandatory*	A single dataset or a *list* of datasets.

Jobids

name	default	description
`previous`	*`None`*	Jobid to previous `csvexport` if chained.

8.3.1 Example Invocation

An example invocation is the following

```
urd.build(csvexport',
    datasets=dict(
        source='test-3/foo',
    ),
    options=dict(
        filename='output.txt.gz',
        separator=' ',
        quote_fields="\'",
        ),
    )
)
```

8.4 dataset_rehash – Hash Partition a Dataset

The dataset_rehash method will create a new dataset based on its source dataset. The new dataset will be hashed on a column specified in the options.

Options

name	default	description
hashlabel	*mandatory*	column for hashing, required. Note that columns typed as list, set, or json cannot be used for hashing.
length	-1	Go back at most this many datasets in a chain. Default is -1, which goes back to previous.source if it exists, or to the first dataset in the chain otherwise.
caption		Optional caption. A reasonable caption is created automatically if left blank
as_chain	*False*	True generates one dataset per slice, False generates one dataset. Default False.

Datasets

name	default	description
source	*mandatory*	Source dataset to rehash
previous	*None*	Previous dataset to chain to.

8.4.1 Example Invocation

An example invocation is the following

```
urd.build('dataset_rehash',
          datasets=dict(source=jid,),
          options=dict(hashlabel='start_date',))
```

8.4.2 Hashing Details

This method will create a new dataset based on all the data in the source dataset. The difference between input and output is in which slices the rows will be stored. For each row, the target slice is determined based on the output value of a hashing function applied to a certain column (the hashlabel) of that row. In code, the operation is similar to

```
from gzutil import siphash

target_sliceno = siphash(cols[hashlabel]) % params.slices
```

8.4.3 Notes on Chains

1. The default operation is to rehash a complete chain of datasets from `source` back to `previous.source`. This is controlled by the `length` option.

2. Internally, `dataset_rehash` always generates one dataset per slice in a chain. This is also what is returned if `as_chain == True`. Otherwise, all datasets will be concatenated into one. Thus, there is a choice of either having the output as a chain of datasets – or as a single dataset. The chain will execute faster, since the concatenation step is omitted.

8.5 `dataset_filter_columns` – Removing Columns from a Dataset

The `dataset_rehash` method removes columns from a dataset. It is typically run before applying methods that operate on all columns of a dataset and only a subset of the columns are required. A typical example is `dataset_rehash` that operates on all columns of a dataset. If not all columns are needed, time and storage can be saved by removing columns using this method prior to applying `dataset_rehash`.

Note that this method only updates soft links, and no data is actually copied. So execution time is typically a fraction of a second and no redundant data is written to disk.

Options

name	default	description
`columns`	`[]`	A list of columns to keep.

8.6 `dataset_sort` – Sorting a Dataset

The method `dataset_sort` is used to sort relatively large datasets. One or more columns may be selected for sorting, and it will sort one column at a time. The sorting algorithm is stable, meaning that things with equal sorting keys will keep their order.

None and `NaN` values will sort the same as the smallest/largest value possible in a comparable type.

Options

name	default	description
`sort_columns`	*mandatory*	A column or a list of columns. If a list is specified, sorting will be carried out from left to right.
`sort_order`	`ascending`	Could be reversed by specifying `descending`
`sort_across_slices`	*False*	If *False*, only sort within slices. Otherwise sort across slices.

Datasets

name	default	description
`source`	*mandatory*	A dataset to sort.
`previous`	*None*	A previous dataset to chain to.

8.6.1 A Practical Limitation

Internally, the method works by reading the columns to sort by, and create an indexing column that stipulates the sorting order. Each column is then read in turn and sorted according to the sorting column.

Therefore, the method has limited sorting capability. Internally, it sorts one column at a time, and it needs to hold that complete column plus an indexing column in memory simultaneously. Still, a standard computer can sort very large datasets without trouble.

8.7 dataset_checksum, dataset_checksum_chain

The `dataset_checksum` method is used to create a single checksum from a dataset based on one or more columns. The chained version returns a single checksum from a dataset chain. It is mainly intended as a debugging aid, enabling comparison of datasets across machines, even if they have different slicing.

If `options.sort=False`, hashing will depend on the actual row order of the dataset. If, on the other hand, `options.sort=True`, hashing will be *slice invariant* and *row order invariant*, meaning that the methods only look at the contents of the dataset(s).

Chain limits will affect the checksum of a chain, so if checksumming two chains containing the same data, but with different number of chained datasets, their checksums will differ.

Note that sorting uses about 64 bytes per row, upper limiting the size of hashable datasets. This corresponds to about 1GB of RAM per 20 million lines or so.

Options

name	default	description
`columns`	`set()`	A set of columns to base the checksum on. Leave blank for all columns
`sort`	`True`	Sort dataset before hashing, see text.
`chain_length`	`-1`	Number of datasets in chain to hash.

Datasets

name	default	description
`source`	*mandatory*	A dataset to sort.
`stop`	chained version only	Stop hashing at this dataset.

Chapter 9

The Executables

9.1 daemon Accelerator Server

The daemon.py is the main Accelerator daemon/server. This program will run in the background and serve runner requests...

9.1.1 Invocation

```
daemon.py [-h] [--debug] [--config CONFIG_FILE] [--port PORT | --socket SOCKET]
```

Optional arguments

-h --help	show help message and exit.
--debug	Start in debug mode. See section ??
DEBUG, please explain! --config CONFIG_FILE	configuration file, default ../conf/framework.conf
--port PORT	listen on TCP port (default *None*)
--socket SOCKET	listen on unix socket, default socket.dir/default

The Accelerator and Runner will connect using a unix socket by default. There is no need to configure anything. Setting a port will make communication happen over that port instead.

9.2 urd Job Database Server

Urd is currently run as a stand-alone server.

9.2.1 Invocation

```
urd.py [-h] [--port PORT] [--path PATH]
```

-h --help	show help message and exit.
--port PORT	listen on TCP port
--path PATH	path to database

9.3 runner Build Script Runner

The runner is used to execute build scripts.

9.3.1 Invocation

Runner is invoked like this

```
automatarunner.py [options] [script]
```

assuming the current work directory is the Accelerator directory. The script is either a filename, or the suffix to a filename starting with automata_.

When the runner starts, it will first instruct the Accelerator to scan all method directories to see if there are any new or changed methods. Thereafter, the Accelerator will proceed and scan all source workdirs to see if any new jobs have been created (by another Accelerator daemon). Thereafter, it will execute the build script.

-h --help	show help message and exit.
-p PORT --port=PORT	Accelerator listening port
-H HOSTNAME --hostname=HOSTNAME	framework hostname
-S SOCKET --socket=SOCKET	Accelerator unix socket (default ./socket.dir/default)
-s SCRIPT --script=SCRIPT	build script to run. package/automata_<SCRIPT>.py. Defaults to "automata". Can be bare arg too.
-A --abort	abort (fail) current job(s).
-P PACKAGE --package=PACKAGE	Run build script from this method directory. Useful if the same script name exists in several method directories, for example for testing purposes.
-f FLAGS --flags=FLAGS	Comma separated list of flags, exposed as the set urd.flags in build script.
-q --quick	skip method updates and workdirs checking for new jobs.
-w --just_wait	just wait for running job, do not run a build script.
--verbose=VERBOSE	verbosity level, one of no, status. dots, or log.
--quiet	same as --verbose=no
--horizon=HORIZON	Time horizon - dates after this are not visible in urd.latest.

To run a build script automata_myscript, do

```
./automatarunner myscript
```

This works as long as the name of the build script is unique, that is, it exists in only one method directory. If not, the method directory can be specified using the -P option.

A build script named automata.py in a method directory dev can be launched by

```
% ./automatarunner -P dev
```

9.3.2 Authorization to Urd

Authorisation to Urd could be set in the `URD_AUTH` environment variable. A common way to invoke the runner with Urd authorisation is like this

```
% URD_AUTH=user:passwd ./runner [script]
```

Note that the purpose of the authentication is actually *identification*. It is used to get write access to certain Urd lists. Nothing more.

9.4 `dsinfo` – Dataset Information

The `dsinfo` command line tool gives a compact, but easy to read, overview of a dataset or a dataset chain. The tool is located in the `accelerator` home directory, and the full file name is `dsinfo.py`.

9.4.1 Invocation

```
dsinfo.py [dataset [dataset [...]]
```

Example invocation

```
% ./dsinfo.py test-20
```

The argument can be one or more jobids or dataset ids. If the argument is a jobid, it is assumed that the dataset name is `default`. If there are more than one dataset in the job, a list of dataset names will be returned.

9.5 `dsgrep` – Find Data in Dataset

The `dsgrep` command line tool is used to look at datasets or dataset chains.

9.5.1 Invocation

```
dsgrep.py [options] pattern ds [ds [...]] [column [column [...]]
```

The `pattern` is a regular expression and `ds` are datasets. For example

```
% dsgrep.py Alice test-0 test-1/special name
```

Will look for the string `Alice` in the `name` column of the two datasets `text-0` and `test-1/special`. Optional arguments are

`-h` `-help`	show help message and exit
`-c` `-chain`	follow dataset chains
`-i` `-ignore-case`	Case insensitive pattern

Strings and columns with special characters have to be quoted.

9.5.2 Abuse dsgrep to show datasets

The data in a dataset may be printed to `stdout` by `grep`ing using a regexp that always matches, like this

```
% ./dsgrep.py . test-0 | less
```

Appendix A

Miscellaneous

A.1 Daemon Configuration File

The Accelerator is shipped with a configuration file template with comments to each line. It is a good idea to copy and modify this when a new configuration file is needed.

Below is an example configuration file that defines two workdirs, called `import` and `processing`. The workdirs are specified relative to the user's home directory. The workdir `processing` is the default workdir for writing, but a build script could explicitly request to write to the `import` workdir as well.

Methods available for use are the `standard_methods` bundled with the Accelerator, and methods defined in the directory `dev` (if they are defined in `dev/methods.conf`).

```
workdir=import:${HOME}/accelerator/workdirs/import:16
workdir=processing:${HOME}/accelerator/workdirs/processing:16

target_workdir=processing

source_workdirs=import,processing

method_directories=dev,standard_methods

result_directory=${HOME}/accelerator/results

source_directory=/some/other/path

logfilename=${HOME}/accelerator/daemon.log

py2=/usr/bin/python2.7
py3=/usr/bin/python3.5
```

and here are explanations to all keywords

name	description
`workdir`	A *workdir*, defined as `name:path:slices`. At least one workdir needs to be defined. All workdirs used together must have the same number of slices. Shell variables are available too, such as `${HOME}`, please see text.
`target_workdir`	Default *workdir* that new jobs get written to. There can only be one target workdir. Please see `source_workdirs` below for more information.
`source_workdirs`	A comma separated list of workdirs available for reading. Workdirs specified by `workdir=...` but not listed as `target_workdir` or `source_workdirs` will be ignored. Note: 1. The `target_workdir` is automatically included in the `source_workdirs`, and does not need to be explicitly. 2. The `target_workdir` is the default workdir for writing new jobs, *but* a build script may write to *any* of the specified `source_workdirs`.
`method_directories`	A comma separated list of directories containing methods. These will be the only directories where the Accelerator can "see" methods. `standard_methods` is bundled with the Accelerator and is commonly used.
`result_directory`	A common path that is available to all jobs. It has been used sparsely by the Accelerator team since it voids the possibility to see where a file comes from. Nevertheless, in some projects it is valuable to have a common place where methods store results, see section A.6.2. A way to keep traceability is to store the file in the job directory as usual, and then create a soft link to it in `result_directory`.
`source_directory`	Default root path for `csvimport`. This is to avoid rebuilds of imports if source files are moved to another directory. (This typically happens when setting up a similar system on another physical machine.) See section A.6.1 on how to get access to `source_directory` from any method.
`urd`	If present, an URL to the Urd server.
`logfilename`	Location of the Accelerator's log.
`py2` and `py3`	path to Python executables. Current versions of the methods in the `standard_methods` directory require Python2. In the future, the Accelerator will require Python3, so it is safest to have both here.

It is possible to assign values in the configuration file using shell environment variables. In the example above, workdirs are specified relative to `${HOME}`, for example. In general, the assignment is `${VAR=DEFAULT}`.

A.2 Setting up Urd

The Urd Source Files

Urd is included in the `accelerator` repository, in the subdirectory `accelerator/urd`.

Dependencies

Urd requires the `bottle` library to run. It is bundled with Debian-like systems, and can be installed by

```
sudo apt-get install python-bottle
```

It can also be downloaded from the project's home page

`https://bottlepy.org/docs/dev/`

as a single python file, `bottle.py`. This file should be put in the `accelerator/urd` directory.

Creating a Database

A new database is created by making a new directory and adding a `passwd` file to it. Urd takes care of the rest. In practice,

```
mkdir database_root
vi database_root/passwd
```

where `vi` is just an editor picked by random.

Starting Urd

Urd is running as a daemon. It is started like this (make sure to `cd` into the `accelerator/urd`-directory.

```
./urd.py --path=<database_root> --port=<port>
```

Where `<database_root>` is a path to an Urd database, and `<port>` is a port number, for example `6502`.

A.2.1 The Urd Database

The Urd database has the following structure

```
database_root/
    passwd
    database/
        user1/
            list1
            list2
        user2/
            list3
```

The `passwd` file

The `passwd` file stores write access authentication. The file format is straightforward, each line is a user–password pair as follows

`user:password`

For example, if the file contains the following line

`ab:secret`

A build script issued like this

```
URD_AUTH=ab:secret ./automatarunner test
```

will have write access to all lists belonging to the user `ab`, such as for example the `ab/test` and `ab/import` lists. But it can not write to lists belonging to other users, such as `cd/import`. It can read all lists, though.

A.3 Workdirs

Jobs are stored in *workdirs*. The Accelerator must have one target workdir, and may optionally have several source workdirs. Target and source workdirs are specified in the daemon configuration file.

By default, the only workdir that is written to is the target workdir, while the source workdirs are for reading. It is possible to override this, however, by setting the `workdir=` option in the `urd.build()` call, see section 7.10.

Jobdirs are stored in the workdir by the daemon, and jobdirs will inherit the workdir name and add a suffix that is an incremental job counter. Here is an example of a workdir named `test`, that contains three jobdirs.

```
test/
    test-slices.conf
    test-0/
    test-1/
    test-2/
    test-LATEST -> test-2
```

The link `<workdir>-LATEST` is always pointing to the last jobdir created. This is useful for example when iteratively testing a method and accessing its data for example for plotting purposes. Each new build of the (modified) method will create a new job and jobid, but the link will always point to the most recent version.

A.3.1 Creating a Workdir

Any empty directory can be a workdir. To create a new workdir, create the directory and add it in the configuration file either as source or target workdir. Stop and restart the `daemon`. Upon startup, any uninitialised source or target workdir in the configuration file will be initialised.

The initiation process creates a file named `<workdir>-slices.conf` that indicates that the directory is now a workdir. The file itself will contain the number of slices that is used for the workdir.

A.4 Progress Indication: `C-t`

During job building, it is possible to press `C-t`, i.e. `Ctrl` + `t` simultaneously, to get status information. The status will cover the processing state, if it is in `prepare`, `analysis`, or `synthesis`, reading or writing files, etc.

If the `analysis` function is executing, `C-t` will list all analysis processes that are active at the moment. If iterating over a dataset, the status message will include which dataset (perhaps in a chain) that is currently being iterated.

Note that `C-t` could be pressed either in the `daemon` or `runner` shells.

A.5 Typical Installation

See the Accelerator Installation Manual for current installation guidelines.

Traditionally, the Accelerator is installed as a `git submodule` as part of the project it is used in. This makes it straightforward to update the Accelerator and project code independently, while linking a specific version of the Accelerator to the project. Thus, the project will always use a specific Accelerator commit, making the project independent of changes to the Accelerator.

Here is a typical setup

```
project/
    accelerator/
    dev/
    conf/
```

Methods are stored in the `dev` directory, and Accelerator configuration files in `conf`. All files are version controlled using `git`, and the `accelerator` directory is a `git submodule`, which means that the project repository is storing the repository location *and* the commit of the Accelerator.

A.6 Working with Relative Paths

In some situations, like importing data from files, it is convenient to store the absolute path of the files as a configuration parameter and then work only with relative paths in the source code. This has two advantages.

First, it makes it possible to move source files around without forcing a re-build of the import jobs, and

second, absolute paths will not be stored in the source code.

In order to make use of relative paths, store the "system dependent" left part of the path in the Accelerator's configuration file. There are two variables in the configuration file that can be used for this, and they have different purposes. The `source_directory` variable is intended for reading source files, and the `result_directory` is intended for writing output. See the following subsections for details.

A.6.1 The SOURCE_DIRECTORY

The `SOURCE_DIRECTORY` variable is used by the `csvimport` method, but could be used by any method reading input files, like in this example

```
import os
options = dict(filename=Optionstring)

def synthesis(SOURCE_DIRECTORY):
    fname = os.path.join(SOURCE_DIRECTORY, options.filename)
```

here, the `fname` is a concatenation of the `SOURCE_DIRECTORY` specified in the Accelerator's configuration file (see section B.3) and the input option `filename`.

A.6.2 The RESULT_DIRECTORY

It is possible to define a shared directory named `result_directory` in the Accelerator's configuration file. In a method, this variable may be accessed like in this example

```
def synthesis(RESULT_DIRECTORY):
    print(RESULT_DIRECTORY)
```

Methods could use this for storing for example plots and reports for a project in one easy accessible common location. Note however, that tracking these files is not possible, there is no information linking back from the result directory to a specific job. This may be overcome using for example soft file links, however, see section B.3.

A.6.3 Understanding Workdirs

When the `Daemon` is starting, it will read all `workdir`-definitions in the configuration file. It will check that all workdirs specified by `target_workdir` and `source_workdirs` are defined. All other defined workdirs are ignored. The daemon will also check that all workdirs have the same number of slices as specified in the configuration file, and that all workdirs to be used together have the same number of slices.

The `target_workdir` is the default location where new jobs will be stored. However, a build script may write jobs to any of the `source_workdirs` as well. Furthermore, the `target_workdir` may always be read from, and may therefore be omitted from the `source_workdirs` list.

A.6.4 How to Create New Workdirs

If a workdir defined in the configuration file does not exist on disk at the stated location, the Daemon will exit and print an error stating that a directory is missing. The first time the Daemon encounters a new directory it will initialise it in accordance with the configuration file. This is true both for source and target workspaces. So, new workdirs are created by adding them to the configuration file *and* creating the corresponding directories. The Accelerator will then initiate these directories on the next startup.

Appendix B

Helper Functions

This chapter is dedicated to the Accelerator's helper functions. Some, like the `blob` module, are very useful, while others are less likely to be included in a project.

B.1 Share Data Between Jobs: the `blob` Module

The simplest way to share reasonable amounts of data between jobs is by using the `blob` module. This module is a convenience-wrapper around the Python `pickle` module.

Note that the Accelerator will set the "current work directory" to the current job directory when building a method, so all files created by a job will be stored in the current job directory, unless the filename contains a path pointing elsewhere.

Storing/Loading a Single File

Data is saved in this way

```
import blob
def synthesis():
    data = ...  # some data created here
    blob.save(data, filename)
```

The data is loaded like this

```
import blob
def synthesis()
    data = blob.load(jobid, filename)
```

The `jobid` in `blob.load()` is not mandatory. It defaults to the current workdir unless specified.

B.1.1 Storing/Loading a Sliced File

It is also possible to use the `blob` module in `analysis`. From a user's perspective it will look like a single file is being handled, but there is actually one file per slice. This is how to do it

```
def analysis(sliceno):
    # save data in slices like this
    blob.save(data, filename, sliceno=sliceno)
    # load like this
    data = blob.load(filename, sliceno=sliceno)
```

Data can be passed "in parallel" between different jobs using this feature.

B.1.2 Default Value

The value of the `default` parameter is returned if trying to load a file that does not exist, for example

```
x = blob.load('thisfiledoesnotexist', default=dict())
```

will set `x` to an empty dict if loading fails.

B.1.3 Save Files for Debugging

The `temp` argument controls persistence of the stored files. By default it is being set to `False`, which implies that the stored file is *not* temporary. But setting it to `True`, like in the following

```
    blob.save(data, filename, temp=True)
```

will cause the stored file to be deleted upon job completion. The argument takes two additional values, `DEBUG` and `DEBUGTEMP`, working like this

temp=	"normal" mode	debug mode
False	stored	
True	stored and removed	stored and removed
DEBUG		stored
DEBUGTEMP	stored and removed	stored

Debug mode is active if the Accelerator is started with the `-debug` flag.
Example

```
from extras import Temp
def analysis(sliceno):
  # save only if --debug
  blob.save(data, filename, sliceno=sliceno, temp=Temp.DEBUG)
  # save always, but remove unless --debug
  blob.save(data, filename, sliceno=sliceno, temp=Temp.DEBUGTEMP)
```

B.2 Find the Full Path to a File in Another Job

Accessing a file stored in another job from within a method or build script is simple, and the functionality is implemented in `resolve_jobid_filename()`. The function takes two arguments, a *jobid* and a *filename*. See the example below

```
from extras import resolve_jobid_filename

jobids = ('oldjob',)

def synthesis():
    filename = resolve_jobid_filename(jobids.oldjob, 'nameoffile')
```

Note that this function works in a build script as well.

B.3 Symlinking

Creating a symlink, for example from the `result_directory` to current workdir, may be implemented in a simple and safe way like this

```
from extras import symlink

def synthesis(RESULT_DIRECTORY):

    # create file and write it to jobid
    ...
    with open(filename, 'wb') as fh:
        fh.write(...)

    # create a symlink to filename in RESULT_DIRECTORY
    symlink(filename, RESULT_DIRECTORY)
```

The `extras.symlink` function will write a soft link to `filename` in `RESULT_DIRECTORY`, overwriting it if it already exists.

B.4 job_params

missing!

B.5 job_post

The `job_post` function returns a job's post execution information as a Python `dict`

```
from extras import job_post
postinfo = job_post(jobid)
```

The post data contains mainly profiling information.

B.6 json_encode

```
from extras import json_encode
json_encode(variable, sort_keys=True, as_str=False)
```

name	description
`variable`	variable to be serialised. `sets` and `tuples` will be converted to `lists`.
`sort_keys`	Sort keys if *True*.
`as_str`	return a `str` if *True*, `bytes` otherwise.

B.7 json_decode

```
from extras import json_decode
x = json_decode(s)
```

Return a datastructure defined by the string `s`.

B.8 json_save

```
from extras import json_save
json_save(variable,
    filename='result',
    jobid=None,
    sliceno=None,
    sort_keys=True,
    _encoder=json_encode,
    temp=False
)
```

B.9 json_load

```
from extras import json_load
x = json_load(
    filename='result',
    jobid='',
    sliceno=None,
    default=None,
    unicode_as_utf8bytes=PY2
)
```

B.10 DotDict

```
    """Like a dict, but with d.foo as well as d['foo'].
    d.foo returns '' for unset values by default, but you can specify
    _attr_default and _item_default constructors (or None to get errors).
    Normally you should specify _default to set them both to the same thing.
    The normal dict.f (get, items, ...) still return the functions.
```

B.11 OptionEnum

```
A little like Enum in python34, but string-like.
(For JSONable method option enums.)

>>> foo = OptionEnum('a b c*')
>>> foo.a
'a'
>>> foo.a == 'a'
True
>>> foo.a == foo['a']
True
>>> isinstance(foo.a, OptionEnumValue)
True
>>> isinstance(foo['a'], OptionEnumValue)
True
>>> foo['cde'] == 'cde'
True
>>> foo['abc']
Traceback (most recent call last):
...
KeyError: 'abc'

Pass either foo (for a default of None) or one of the members
as the value in options{}. You get a string back, which compares
equal to the member of the same name.

Set none_ok if you accept None as the value.

If a value ends in * that matches all endings. You can only access
these as foo['cde'] (for use in options{}).
```

B.12 OptionString

```
    """Marker value to specify in options{} for requiring a non-empty string.
    You can use plain OptionString, or you can use OptionString('example'),
    without making 'example' the default.
```

B.13 RequiredOption

```
Specify that this option is mandatory (that the caller must specify a value).
    None is accepted as a specified value if you pass none_ok=True.
```

B.14 OptionDefault

```
"''Default selection for complexly typed options.
foo={'bar': OptionEnum(...)} is a mandatory option.
foo=OptionDefault({'bar': OptionEnum(...)}) isn't.
(Default None unless specified.)
```

B.15 gzutil

```
with gzutil.GzUnicodeLines(filename, strip_bom=True) as fh:
```

B.16 profile_jobs

see redmine

Index

`job_params`, **33**
`link_to_here`, 34
`override_previous`, 35
`urd.build`, 21

build script
    `main` function, 21
build scripts, 21

input options
    `JobWithFile`, 38
    `OptionEnum`, 36
    `OptionString`, 36
    `RequiredOptions`, 36

subjobs, 34