The Accelerator

User's Reference

Anders Berkeman, Carl Drougge, and Sofia Hörberg

version: 772990f4

Contents

Chapter 1

Introduction

The Accelerator is a tool for fast 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 varous data processing tasks, such as recommendation systems, and more. It has a small footprint and runs on laptops as well as rack servers.

It 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, and Vodafone. Most project have been related to data analysis, some to optimisation, and some projects have been live-running recommendation systems for years. The Accelerator has evolved from being the core of these projects. In 2016, it was aquired 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.

1.1 Main Design Goals

The Accelerator is designed to process log-files. Most data can be represented in terms of log files, a format that brings determinism (i.e. repeatability) as well as transparency, which are important, but hard to achieve, factors in many projects. 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.
>
> Never recompute old results, always recycle 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.

Chapter 2

Overview

This chapter presents an overview of the Accelerator's features in a rather non-formal way.

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

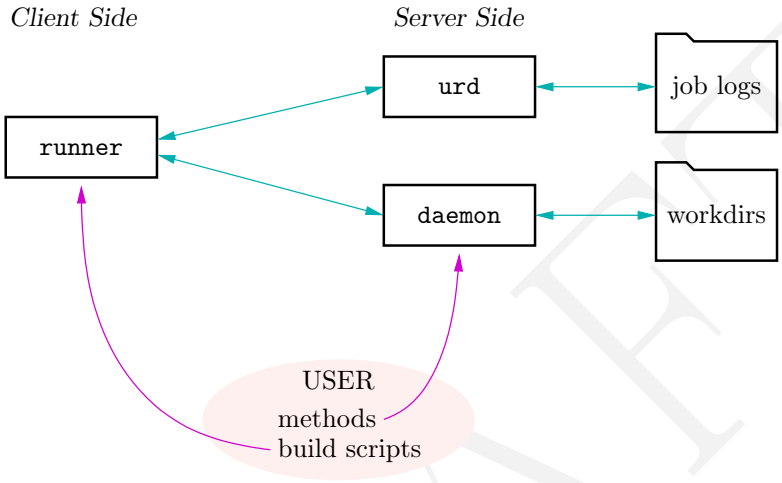

Figure 2.1: High level view of the Accelerator framework. See text for details.

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 will 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.

2.2 Jobs: Executing Code

The basic operation of the Accelerator is to execute small 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 this is very useful for avoiding unnecessary re-computing and instead rely on previosly 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, result sharing, and parameters.

2.2.1 Basic Job Running: "Hello, World"

Let's begin with a simple "hello world" program. We create a method with the following contents

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

This program does not take any input parameters. It just returns a string and exits. Running methods on the Accelerator is further explained in section 3.

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 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 ones right now are

`setup.json`, which contains job information; and

`result.pickle`, that contains the returned data.

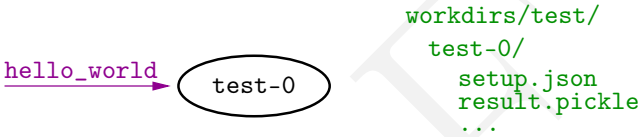

Figure 2.2: A simple hello world program, represented as graph and work directory.

2.2.2 Linking 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`, that 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 we will see in section 2.2.4 how to do this. The 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.

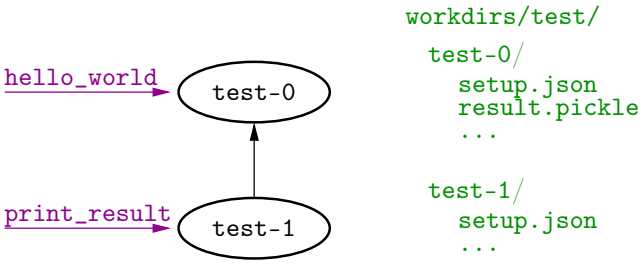

Figure 2.3: Jobid `test-0`, is used as input to the `print_result` job.

2.2.3 Job Execution Flow and Result Passing

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 paramerers 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 2.4 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.

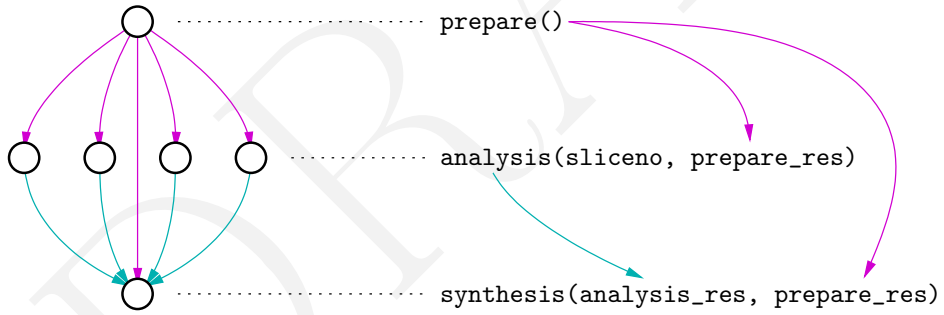

Figure 2.4: Execution flow and result propagation in a method.

2.2.4 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*, explained later.

See Figure 2.5. 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 make 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 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.

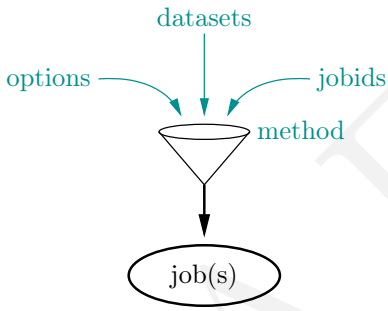

Figure 2.5: Execution flow of a method. The method takes optionally three kinds of parameters: `options`, `jobids`, and `datasets`.

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.

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`

Datasets will be described in more details later, and the `datasets` parameter is similar to the `jobids` parameter.

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 parallell access. Columns are accessed independently. Furthermore, datasets may be *hashed*, so that slicing is based on the hash value of a given column. In many practical applications, hashing makes parallel processes independent, minimising the need for complicated merging operations. This is explained further in chapter 6.

2.3.1 Importing Data

Let's have a look at the common operation of *importing*, i.e. creating a dataset from a file. See figure 2.6.

Figure 2.6: Importing `file0.txt`.

The standard method `csvimport`, is designed to parse a pletoria 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. The created dataset is 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.

2.3.2 Linking Datasets, Chaining

Just like jobs can be linked to eachother, datasets can link to eachother 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 2.7.

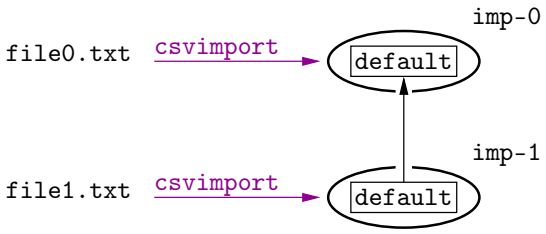

Figure 2.7: Chaining the import of `file1.txt` to the previous import of `file0.txt`.

The `imp-1` (or `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 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. The only thing that needs to be done is to set a link, so the overhead is minimal.

Now we'll see that is it equally simple to add new columns to an existing dataset. Adding columns is also a common operation and the Accelerator handles it efficiently using links.

The idea is very simple. Assume that we have a "source" dataset to which we want to add a new column. We create a new dataset containing *only* the new column, and while creating it we instruct the Accelerator to link the source dataset to the new column.

Accessing the new one-column dataset will transparently access all the data in the source dataset too, making it indistinguishable from a single dataset. See Figure 2.8.

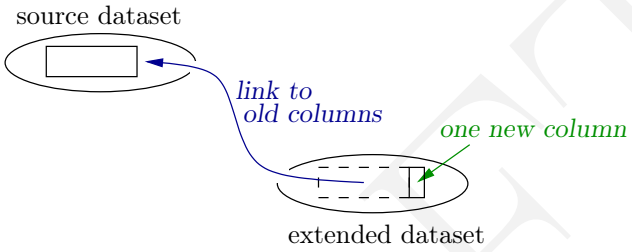

Figure 2.8: caption here

2.3.4 Multiple Datasets in a Job

We've seen that datasets are created by methods and stored in job directories. Typically, a method creates a single dataset in the job directory, but there is no limit on how many datasets that could be created and stored in a single job directory. This leads to some interesting applications.

One application where it is convenient to create multiple datasets in a job is when splitting data into subsets based on some condition. For example, assume that we want to separate a dataset into two disjoint datasets based on a column storing a boolean value. See Figure 2.9.

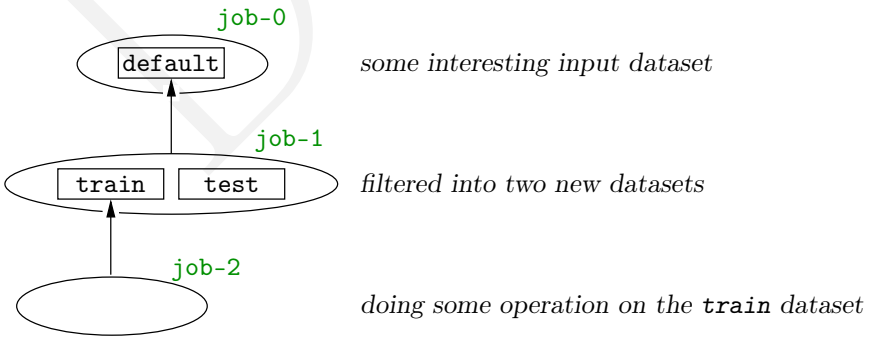

Figure 2.9: `job-1` separates the dataset `job-0/default` into two new datasets, named `job-1/train` and `job-1/test`.

The figure shows how `job-1` has created two datasets, `job-1/train` and `job-1/test`, based on the input dataset `job-0/default`. A third job, `job-2` is then accessing the `job-1/train` dataset. (Note that `job-1` does not have a `default` dataset.)

Let us look at an example of when such dataset splitting makes sense, and how it relates to the design methodology that the Accelerator is based upon. Assume that we have a (perhaps large) dataset that we want to split into, say, a training set and a validation set. Even if one set is small, it makes sense to split the dataset into two disjoint sets. This way, we "physically" separate the data into two sets, while keeping all the data in the same place. This is good for transparency reasons, and any method following the split may iterate over both subsets to read the complete data.

2.3.5 Parallel Dataset Access and Hashing

As shown in detail in section 6, data in datasets are stored in multiple files, allowing for fast parallel reads. The parameter `slices` determines how many slices that the dataset should be partitioned into. This parameter also sets the number of parallel `analysis`-processes, so that each `analysis` process operates on a unique slice 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 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 some parallel processing tasks.

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. Several of these types are designed to make importing data from text files straightforward, without parse errors, overflows etc.

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)
```

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. 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 of the `synthesis` function, which is called 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.

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. 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. The `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.

It is also possible to have the iterate 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.

Chapter 3

Basic Build Scripting

Build scripts are used to instruct the Accelerator about which jobs to build. The Accelerator will consult its internal database to see if such a job already exists, before attempting to build it. This chapter describes the basics of job building. More advanced features, using the `urd` server, is presented in chapter 4.

3.1 Build Scripts

Build scripts are executed by the `runner` server. A build script must contain the function `main` as shown below

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

At run time, the `runner` inserts an object of the `Urd` class as argument to the `main` function. This object has a number of member functions and attributes useful for job building. It also records all jobs that are build, together with their input parameters and some more meta information described in the next sections.

3.1.1 Building a Job

The `build` function is typically used to build a job from a method. Here is an example of how to build the method `method1`.

```
jobid = urd.build('method1', options={}, datasets={}, jobids={}, name='', caption='')
```

All options are optional, and `options`, `datasets`, and `jobids` depend on the method to be executed. The `name` will override the default name, which is equal to the name of the method, when the job is recorded by Urd. This is useful if several jobs are build based on the same method. Naming them uniquely makes it easier to tell them apart later. In addition, it is also possible to assign a caption to a job.

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

In addition, a name and a caption may be specified too

```
jobid = urd.build('method1', name='myjob', caption='looking for something')
```

The name will override the default name (which is the name of the method) in the Urd list. In this case, this job will now be referred to as `myjob` (instead of default `method1`). A jobid to the finished job is returned upon successful completion.

3.1.2 Handling Consecutive Jobs

Using the output jobid from the `build` function, it is straightforward to connect jobs in series. For example

```
jid_filter = urd.build('filter_data', datasets=dict(source=<some input>))
jid_reduce = urd.build('reduce', datasets=dict(source=jid_filter))
```

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

If the first method or its input data is changed, the job will run again. This will cause the jobid `jid_filter` to change too, which in turn will trig execution of the `reduce` job.

3.1.3 Building Chained Jobs

It is also possible to build chained jobs implicitly using the `build_chained` function

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

which takes the same options as the standard `build` method, with the exception that name is mandatory, since it is used to find the previous job of matching type.

Chapter 4

High Level Control: Urd

4.1 Introduction to Urd

Urd is the processing flow controller in the framework. It is the primary job dispatcher as well as the bookkeeper of all jobs executed. Events in urd are quantified into what is called *sessions*. The core of Urd is a transaction log database storing these sessions together with meta information. The result is a server providing lookups for all jobs executed together with their context.

The Urd database is partitioned into what is called *lists*. Lists are where information about executed jobs are stored, and how they relate to eachother. Lists are globally readable, but writing requires authentication, so that, for example, only the production user may publish a model to go live.

With the exception of experimental work, all work initiated by Urd is run in closed sessions, with well defined starting and ending points. The input dependencies to these sessions are recorded, together with the resulting output.

4.2 Urd Sessions

A minimal example of creating an Urd session is as follows

```
def main(urd):
    urd.begin('test')
    ...
    urd.finish('test', '2016-10-25')
```

Every job that is dispatched between `begin` and `finish` will be appended to the `test` Urd list with timestamp `20161025`. In addition, all lookups of jobs done by Urd will be appended to the Urd list as dependencies.

The Urd list will be updated only when the `finish` function is called, since it is responsible for updating the urd transaction log. Before `finish`, nothing is stored, and it is perfectly okay to omit `finish` during development work.

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

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

and the following applies

> `path` is the name of the Urd list, and the same `path` must be specified in both `begin` and `finish`.
>
> `timestamp` is mandatory, but could be set in either `begin`, `finish`, or both. `finish` overrides `begin`.
>
> `caption` is optional, and can be set in either `begin` or `finish`. `finish` overrides `begin`.

There is also an `update` option that will be discussed in section 4.7.

4.2.1 Timestamp resolution

Timestamps may be specified in various resolution depending on the application. The time format is

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

and it can be truncated as shown in the following examples covering 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
```

4.2.2 Aborting an Urd Session

When an Urd session is initiated, a new session cannot be started until the current session has finished. A session may be aborted, however, like this

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

Aborted sessions are not stored in the Urd transaction log.

4.3 Building Jobs

Jobs are dispatched in Urd sessions using the `build` function. The syntax is just the same as in the previous section that did not use sessions.

```
jobid = urd.build('method1', options={}, datasets={}, jobids={}, ...)
```

Here, `options`, `datasets`, and `jobids` are optional, depending on the method to be dispatched. In addition, a name and a caption may be specified too

```
jobid = urd.build('method1', name='myjob', caption='looking for something')
```

The name will override the default name, which is the name of the method, in the Urd list. In this case, this job will now be referred to as `myjob` (instead of default `method1`). This is useful to separate jobs if the same method is used multiple times in the same list. A jobid to the finished job is returned upon successful completion.

4.3.1 Handling Consecutive Jobs

Using the output jobid from the `build` function, it is straightforward to connect jobs in series. For example

```
jid_filter = urd.build('filter_data', datasets=dict(source=<some input>))
jid_reduce = urd.build('reduce', datasets=dict(source=jid_filter))
```

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

If the first method or its input data is changed, the first job will be run again. This will cause the jobid `jid_filter` to change too, which will, in turn, trig execution of the `reduce` job. The Urd list will contain jobids to both jobs, and all input parameters as well, so it is clear which `filter_data` job that was used as input for the `reduce` job.

4.3.2 Building Chained Jobs

Using the `build_chained` function, it is possible to build chained jobs implicitly, like this

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

This function takes the same options as the standard `build` method, with the exception that name is mandatory, since it is used to find the previous job of matching type.

4.4 Urd Sessions with Dependencies

forst en import-automata har, och sedan anvand den...

A job may have dependencies, such as other jobs or datasets. These dependencies are input to the job using the corresponding arguments to the `build` function. Locating these jobs or datasets, however, is exactly a design goal of Urd. Urd implements a `get` function that looks up jobids and dependencies from a key that is composed of an Urd list name plus a timestamp. There are also, for convenience, `first` and `latest` functions to get the first and latest job in an Urd list.

Here is an example. Assume that we have a build script that imports data files. Information about the import jobs is stored in the `import` Urd list.

```
def main(urd):
    now = '20180403'
    urd.begin('import', now)
    jid_prev = urd.latest('import')
    urd.build('csvimport',
        options=dict(filename='log' + now + '.txt',),
        datasets=dict(previous=jid_prev)
    )
    urd.finish('import')
```

Note how the `previous` dataset is assigned from the output of the `urd.latest` call, making the import jobs *chained*. The call to `latest` will be recorded in the `import` Urd list as well.

Now, assume that a method `computesomething`, uses these imported datasets. When dispatching `computesomething`, it should be using the latest available `import`. This example shows again how the function `latest` is used for this purpose

```
def main(urd):
    urd.begin('test')
    latest_import = urd.latest('import').joblist.jobid
    urd.build('computesomething', datasets=dict(source=latest_import))
    urd.finish('test', '20180403')
```

Two things have happened here. First, urd has provided a jobid link to the latest available `import` dataset. Second, the dependency of exactly this version of `import` to `computesomething` is recorded in the urd list `test` for timestamp `20180403`. So, if there is a question in the future which version of the `import` database that was used on that date for the `computesomething` function, it is immediately available from urd.

The more general form is `get`, which is shown below together with its derived convenience-functions

```
  urd.get('test', '20161001')
  urd.latest('test')
  urd.first('test')
```

And here is an example of running `computesomething` on `import` data from previous month

```
def main(urd):
  urd.begin('test')
  import = urd.get('import', '20160925').joblist.jobid
  urd.build('computesomething', datasets=dict(import=import))
  urd.finish('test', '20161025')
```

4.5 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` and `peek_latest`, like this:

```
urd.peek('test', '20161025')
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.

4.6 More on Finding Items in Urd

There is a `list` function that returns what lists are recorded in the database:

```
print(urd.list())
# ['ab/test', 'ab/live']
```

And there is also a `since` function that returns a list of all timestamps after the input argument

```
print(urd.since('20161005'))
# ["20161006", "20161007", "20161008", "20161009"]
```

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.

```
print(urd.since('0'))
# ["20160101", "20161004", "20161005", "20161006", "20161007", "20161008"]
print(urd.since('2016'))
# ["20160101", "20161004", "20161005", "20161006", "20161007", "20161008"]
print(urd.since('20161'))
# ["20161004", "20161005", "20161006", "20161007", "20161008"]

print(urd.since('2016105'))
# ["20161006", "20161007", "20161008"]
...
print(urd.since('2016105 000000'))
# ["20161006", "20161007", "20161008"]
```

4.7 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 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.

To update a list, use the `update` argument

```
urd.begin('test', '20161025', update=True)
```

If update is True, the entry in the test list at '20161025' will be updated, unless there has been no change. and in order to set a marker in the database indicating that everything before a certain date in time should be discarded, do like this

```
urd.truncate('test', '20160930')
```

This will rollback everything that has happened in the `test` list back to '20160930'. Remember, internally Urd stores the complete history in a log file in plain text.

4.8 More on Joblist and Jobtuple

Urd is using the type joblist to keep track of successfully executed jobs. Each item in the joblist is of type jobtuple. This section will start by describing jobtuple first and then joblist.

Jobtuple

The `JobTuple` type is used to group method names and corresponding jobids. It is basically a tuple with some extra properties, such as a conversion of a jobtuple to `str`, which happens for example when printing it, returns the jobid as a string.

```
>>> jt = JobTuple('imprt', 'jid-0')

>>> jt
('imprt', 'jid-0')
```

as expected, and

```
>>> jt.method
'imprt'

>>> jt.jobid
'jid-0'
```

but note that

```
>>> print(jt)  # str and encode return jobid only
jid-0
```

JobList

The `JobList` is a list with add-ons for bookkeeping and finding jobs. It stores instances of `JobTuple`. Here is an example. First, define a `JobTuple`

```
>>> jt = JobTuple('imprt', 'jid-0')
```

then define a joblist initiated with the same tuple. Then append some more jobs directly using the `append` method.

```
>>> jl = JobList(jt)
>>> jl.append('learn', 'lrn-0')
>>> jl.append('imprt', 'imp-1')
```

Let's see how and what is stored in the `JobList`. The `pretty` method is quite useful, but note that just printing the object will show the last jobid only.

```
>>> print(jl.pretty)
JobList(
   [  0]  imprt : jid-0
   [  1]  learn : lrn-0
   [  2]  imprt : imp-1
)

>>>  print(jl)  # jobid of latest appended JobTuple
imp-1
```

It is easy to retrieve the last job with a particular `method` name, either by lookup or by using `find`.

```
>>> jl['imprt']          # latest jobid with name 'imprt'
('imprt', 'imp-1')

>>> print(jl['imprt'])   # jl['imprt'] is JobTuple
imp-1
```

The Find method returns a `JobList`. Slicing also returns `JobLists`

```
>>> jl.find('imprt')
JobList([('imprt', 'jid-0'), ('imprt', 'imp-1')])

>>> jl[:2]
JobList([('imprt', 'jid-0'), ('learn', 'lrn-0')])
```

Looking up by index returns `JobTuple`.

```
>>> jl[0]
('imprt', 'jid-0')
```

These conveniences are also supported

```
>>> jl.all              # list of all jobids
'jid-0,lrn-0,imp-1'

>>> jl.method           # last method
'import'

>>> jl.jobid            # last jobid
'imp-1'
```

4.9 Talking directly to Urd: The Urd HTTP-API

In some situations it is convenient to make calls to urd directly without using the framework. Urd will react to HTTP requests, so a tool like `curl` suffice.
Show all stored lists like this

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

Looking up the latest stored job in the test list

```
% curl http://localhost:8833/ab/test/latest
{"caption": "", "automata": "test", "user": "ab", "deps": {},
  "timestamp": "20161025", "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
{"caption": "", "automata": "test", "user": "ab", "deps": {},
  "timestamp": "20161025", "joblist": [["method1", "test-56"],
  ["method2", "test-59"], ["method3", "test-60"]]}
```

See what is inside the test list stored at `20161025`

```
% curl http://localhost:8833/ab/test/20161025
{"caption": "", "automata": "test", "user": "ab", "deps": {},
  "timestamp": "20161025", "joblist": [["method1", "test-56"],
  ["method2", "test-59"], ["method3", "test-60"]]}
```

And what is avaible in the test list that is more recent than `20161024`

```
% curl http://localhost:8833/ab/test/since/20161024
["20161025"]

% curl http://localhost:8833/ab/test/since/20161026
[]
```

Chapter 5

Jobs

The Accelerator is producing *jobs*. A job is a storage of all information associated with a computation, including source code, input parameters, outputs (results), and profiling information. Jobs are static, and cannot be altered or removed by the Accelerator.

The source files that are executed are called *methods*. When the Accelerator executes, or *builds*, a method, the result is a *job*. Jobs are dispatched either by build scripts, see 3, or from methods, using *subjobs*, that will be explained in this chapter. At build time, the job is typically assigned some *input parameters*.

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 retreiving data. Building subjobs.

5.1 Method Directories

Methods are stored in and grouped into method directories, corresponding to file system directories. The recommended location of method directories is one level above the Accelerator's home directory. This is set up automatically if the Accelerator is installed using the Project Skeleton repository **??**.

Method directories are just like any directories, with the addition of two specific files, `__init__.py` and `methods.conf`. The first file is required for Python to see it as a library, and the latter is there to limit execution to permitted files only. If a method is not listed in the method directory's `methods.conf`, it can not be executed by the Accelerator. The Accelerator has a history of use in live production environments, and in such scenarios it is important to keep track of which code that is allowed to execute.

Here is a typical setup

```
project_directory/
    accelerator                     # accelerator home
    accelerator/standard_methods    # method directory for bundled methods
    dev                             # a method directory, here be methods
    my_dev                          # another method directory
```

For a method directory to be visible by the Accelerator, two things are needed

1. it must be specified in the Accelerator's configuration file, see section A.1.2;
2. the method directory must contain an empty file named `__init__.py`; and
3. the method directory must have a `methods.conf` file.

Creating a New Method Directory

Make sure the current directory is either one step above the Accelerator directory (preferred), or is the Accelerator directory.

1. Create and populate the directory, here called `<dirname>`:

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

2. Add the method directory `<dirname>` to the Accelerator's configuration file, see A.1.2.
3. Restart the Accelerator

5.2 Method Source Files

The Accelerator searches method directories for methods to execute. In order to reduce risk of executing the wrong files, 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 5.13; 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.

Adding a New Method

1. Create the method in a method directory 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 5.13.

3. (Make sure that the method directory is in the Accelerator's configuration file.)

5.3 Method Already Built Check

Prior to building a method, the Accelerator checks if an equivalent job has been build 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.

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:

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

It is possible to specify either Python module objects or a filename relative to the method's location.

5.4 Avoiding Rebuild

A different scenario is when the method source code needs to be modified, but the modification does not alter its behavior, so rebuilding jobs should be actively avoided. A simple example would be changing the name of a variable in the method. For this situation, there is an `equivalent_hashes dict` that can be used to specify which versions of the source code that are equivalent. For example

```
equivalent_hashes = {'verifier': ('0c573685f713ac6500a6eda7df1a7b3f',)}
```

The verifier string is a hash that depends on everything in the method except the `equivalent_hashes` block. When the hash value of the method does not correspond to the verifier (as will happen after editing the method) the Accelerator will tell what the correct verifier would be, so that it can be added to the list of verifiers if appropriate. The verifier list can contain any number of old hashes.

5.5 Method Parameters

A jobs parameters are available in the `params` variable. It contains both input parameters that are assigned from the caller, and parameters that are created when the job starts building. Input parameters will be described thoroughly in section 5.6. 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`, `datasets`, and `options` variables.

```
import json

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

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

And here is the corresponding output.

```
{
    "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

key	*description*
`package`	method directory 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`	current source code hash value
`options`	input parameter
`dataset`	input parameter
`jobids`	input parameter

[1] The Accelerator team recommends *not* using `seed`, since it voids determinism.

5.6 Input Parameters

A method is typically provided with input parameters at build time. There are three kinds of method input parameters: `jobids`, `datasets`, and `options`. These parameters are specified 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 then populated by the builder 4.

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. The `options` dictionary is used to input any other type of parameters to be used by the method at run time. Note that `jobids` and `datasets` are tuples (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.

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 may be used as a reference to the corresponding job.

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. The dataset class is described in a dedicated chapter 6.

All items in the `datasets` tuple must be assigned by the builder to avoid run time errors.

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.

Options are straightforward to use, but actually quite advanced, and the following sections introduce how to use them both from a formal perspective, and from a set of examples. On the highest level, options are specified like this

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

Formal Option Rules

Here are 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 6.8.1.) In a running method, the `JobWithFile` object has these members

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

5.7 Accessing Another Job's Parameters

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'
    print(job_params(jobids.anotherjob).options)
    # will print the options of anotherjob, perhaps something like {length: 3}
```

5.8 Execution Flow: prepare, analysis, and synthesis

There are three pre-defined functions in a method that are called by the Accelerator at build time. These are `prepare`, `analysis`, and `synthesis`, and they are always run 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.

All the three functions take the `params` variable as optional argument. 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.

5.8.1 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.

5.8.2 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.

5.8.3 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`.

5.9 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.

5.10 Share Data Between Jobs: the `blob` Module

The simplest way to share reasonable amounts of data between jobs is by using the `blob` 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 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()
    blob.load(filename)
```

Storing 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)
```

Save Files for Debugging

There is one more argument, `temp`, which controls persistence of the files. It is by default 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

`DEBUG` – file will be stored *only* in debug mode

`DEBUGTEMP` – file will always be stored, but *removed* upon job completion only in debug mode.

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)
```

5.11 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 4, 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 copy 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.

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')
   ```

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

5.12 Summary

A recap and bit more flesh on the bones regarding jobs

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 its *current working directory* 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 infomation, 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.

5.13 Methods.conf

Each method directory requires a `methods.conf` file. This file specifies which methods in the directory that are available for building, i.e. it acts as a method *filter*. The reason for this is that it provides a way to know for certain which methods that are allowed to run, something that makes a lot of sense in any production environment.

The `methods.conf` is a 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 on a line by first stating the name of the method, without the `a_` prefix and `.py` suffix, followed by one or more whitespaces and a token `py2` or `py3`, indicating which python version the method should be built by. For example

```
# this is a comment

test2     py2

test3     py3  # newer
#bogusmethod    py3
```

This file declares two methods corresponding to the filenames `a_test2.py` and `a_test3.py`, written in Python2 and Python3 respectively. Another method `bogusmethod` is commented out and will cause an error if trying to being built.

Chapter 6

Datasets

The Dataset class provides fast and simple access to data. It is the prefered 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 typically 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 even random, 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.

6.1 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 2.3.6.

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 6.1. The number of slices is set initally 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 signinficant overhead from disk `seek()` if using rotating disks. The Accelerator mitigates this by using single files with offset-indexing when appropriate.

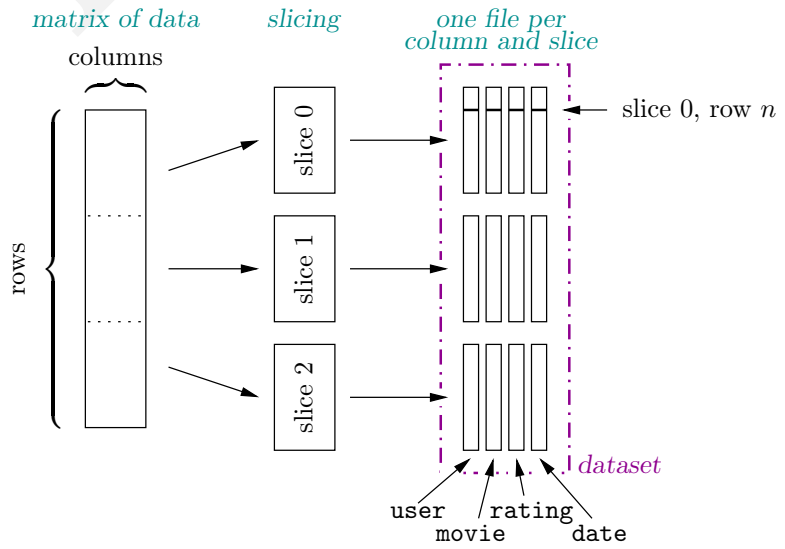

Figure 6.1: A "movie rating" dataset composed of four columns sliced into three slices.

6.2 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 7), which may continue iterating over the next dataset in the chain when the current dataset is exhausted.

6.3 Slicing and Hashing

Datasets are by default sliced into a number of slices specified by the Accelerator's configuration file A.1.2. 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.

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 mathematical terms, round robin would be

$$\text{row } n \longrightarrow \text{slice}(n \bmod N) \tag{6.1}$$

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

$$\text{row } n \longrightarrow \text{slice}(\text{hash}(\text{word}) \bmod N) \tag{6.2}$$

The advantage of this method is that data in each slice becomes independent in many application, which is ideal for parallel processing of the dataset. The hash function used by the Accelerator is a well-known function called `siphash24` that is available from the `gzutil` library

```
from gzutil import siphash24
```

6.4 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.

6.5 Datasets from Jobids

Although a not so common operation, a Dataset object can be instantiated from a jobid, like this

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

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

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

6.6 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.

Column Names

All columns in a dataset may be aquired 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.

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 7.3.

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.

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.

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
```

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 accessable 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
```

Chains

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

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

6.7 Column Typing

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.

typename	explanation
`number`	float or int
`number:int`	int
`float64`	64 bit (double) float
`float32`	32 bit float
`int64`	64 bit signed integer
`int32`	32 bit integer
`bool`	True or False
`date`	date
`time`	time
`datetime`	complete date and time object
`bytes`	raw input, avoid
`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

6.7.1 Arbitrary precision numbers: `number`

The type `number` is integer when possible and float otherwise. it can handle very large numbers, up to $\pm(2^{1007}-1)$. Integers are enforced using `number:int`, and it 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.

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.

6.7.2 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.

6.7.3 Booleans

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

6.7.4 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')
```

6.7.5 String Types

There are three string types, `bytes`, `ascii`, and `unicode`. The `bytes` type is what is assigned to columns by `csvimport`. For columns with text, `unicode` is the prefered choice, and it executes faster in Python 3.

6.7.6 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.

6.7.7 `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'`.

6.8 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

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

6.8.1 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.

6.8.2 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.

6.8.3 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.

6.8.4 More Advanced Dataset Creation

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

6.9 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 "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. The following example appends one column to an existing dataset while maintaining the chain. Note that appending a column does only apply to one single dataset, and not to the complete chain of datasets, if present.

```
from dataset import DatasetWriter

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

def prepare():
    dw = DatasetWriter(
        parent=datasets.source,
        previous=datasets.previous,
        caption='with the new colum'
    )
    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.

It is strongly recommended that new columns are added in `analysis`, and **not** in `synthesis`. This is because reading data is one full slice at a time, while writing data is round-robin per row, and unless being very careful, new column values will not be added to the correct rows.

Chapter 7

Iterators

Iterators are members of the `dataset` class. They are available in methods for streaming dataset data one row at a time into a program.

The iterator iterates over one or more specified data columns. Each output is a tuple corresponding to the specified columns for one data row. In case of iterating over a single column, the output may optionally be a scalar for more efficient computing. Iterators can be parallel, in `analysis`, or sequential, in `prepare` or `synthesis`. There are three iterators available,

`iterate()`, for single dataset iteration,

`iterate_list()` for iterating over a list of datasets, and

`iterate_chain()` for iterating over dataset chains.

Many common iterator usecases require only two options, `sliceno` (mandatory) and `columns`, so a typical call may look like this

```
for x in dataset.source.iterate(sliceno, ('movie', 'user',)):
```

All 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.
`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 it is 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. Ideally, rehashing should be made using the `dataset_rehash` method 8.4.
`filters`	*None*	Filters decide which rows to include. Explained in section 7.7
`translators`	*None*	Translators transform data values. Explained in section 7.6
`status_reporting`	*True*	Give status when pressing `C-t`. Unless manually `zip`ing iterators, this should be set to default *True*. See `dataset.py` 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.
`sloppy_range`	*False*	Used with `range`, but will iterate over full datasets for those datasets that have values within range. This might be faster.
`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.
`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.

7.1 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. (If we assume that the dataset is hashed 6.3, 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.

```
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 that will iterate over the datasets in the chain 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_id=jobids.params.source):
        h[user].add(item)
```

Assuming that `previous` is a jobid to the previous invocation of the method, `jobids.params.source` will be the input `source` dataset to that job.

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 tuples populated with all column values for each row. The columns will be in sorted column name order.

Iterators output tuples, but in the case of iterating over a single column it is 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

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

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

7.2 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_id`

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

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

A more advanced, but very useful, method is to stop at a dataset that is input to another job

Halting Using Another Job's Input Parameters

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

This will iterate until reaching end or the dataset `job_params(jobids.preprocess).datasets.source`.

7.3 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 specifying `sloppy_range`, which will iterate over all datasets that contain part of the range.

7.4 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 always in the forward direction within each dataset!

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

7.5 Hashed Datasets and Hashing Datasets

Hashing a dataset on a particular columns, see 6.3 may simplify the code in methods using the dataset. This code will not work properly if it turns out that it is being fed with unhashed datasets. Therefore, 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')}
```

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`. 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.

7.6 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 that they provide a way to modify code behavior by supplying functions as iterator options. Using translators and filters, it is possible to write re-useable functions that can have different behaviour depending on context.

Callable Translator

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`

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.

7.7 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 a callables or a `dict`s.

Callable Filter

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 by reading 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

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

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

Filter by Column Values

Column values could also be used directly, i.e. the values get evaluated by Python. 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`.

7.8 Callback

The iterator may be assigned callback functions that are called before starting iterating a new dataset, and after the current dataset is exhausted. 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, 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(id):
    print(id)

for user, item in datasets.source.iterate(sliceno, ('user', 'item',),
```

```
                                        pre_callback=prefun):
    ...
```

Next is an example of an iterator running over all slices. The callback function is executed before each new slice is iterated. Note the difference between this example and the previous. The callback function in this example takes *two* arguments, while the previous takes only one.

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

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 have the iterator skip a slice, do

```
# Raise this in pre_callback to skip iterating the coming slice
# (if your callback doesn't want sliceno, this is the same as SkipJob)
raise SkipSlice
```

To skip the next dataset do

```
# Raise this in pre_callback to skip iterating the coing job
raise SkipJob
```

And to abort iterating completely

```
# Raise this to quit iterating, but with the side effect that
# post_callback will not run.
raise StopIteration
```

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

This method is used to import a text file in tabular format (CSV, Comma Separated Values) into a dataset. A wide range of data formats is supported, and the method reads plain text files as well as `gzip` compressed files. A number of options are available to customise importing to specific cases.

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 A.1.2. 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 this option is enabled.

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.

8.2 dataset_type – Typing Datasets

This method will read a source dataset and create a new dataset that is typed. This is useful 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.

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. Removing rows from a dataset is not possible. Instead, the `dataset_type` method will create a new dataset containing only rows with 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
`source`	*mandatory*	Dataset to type.
`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', ...,
  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 options in detail.

Numbers

The *number* type is int or float.

`number`	`int` or `float`
`number:int`	`int`, converts floats to int.

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 deteministic 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 reversability).

Using `strict` will cause errors if unparsable.

8.3 csvexport – Exporting Text Files

The dataset_export metod 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.

8.4 dataset_rehash – Hash Partition a Dataset

Dataset rehash 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 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
```

8.4.2 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

This method removes columns from a dataset. It is typically run before applying methods that operate on all columns of a dataset when 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 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 is typically significantly below 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.

Options

name	default	description
`sort_columns`	*mandatory*	A column or a list of columns. If a list is specified, sorting will be carried out 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.

Thus, 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.

8.7 dataset_datesplit, dataset_datesplit_discarded

8.8 dataset_checksum, dataset_checksum_chain

Chapter 9

Command Line Tools

9.1 `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`. 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.

Appendix A

Practicalities

There are three main components in the Accelerator framework, the `daemon`, the `runner`, and `urd`. How to configure and run these components is the topic of this chapter.

A.1 Daemon

A.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 **??**
`--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.

A.1.2 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`. Both are available for reading, but only the latter may be written to. 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.
`target_workdir`	*workdir* used to write jobs to. There can only be one target workdir.
`source_workdirs`	a comma separated list of workdirs available for reading. These will be the only workdirs that the Accelerator can "see". `standard_methods` is bundled with the Accelerator and is commonly used.
`method_directories`	a comma separated list of directories containing methods. These will be the only directories where the Accelerator can "see" methods.

`result_directory`	A common path that is available to all jobs. It can be accessed in a method like this
	`def synthesis(RESULT_DIRECTORY):`
	`  ...`
	It has been used very sparsely by the Accelerator team since it voids the possibility to see where a file comes from.

`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 seting up a similar system on another physical machine.)
`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 Runner

A.2.1 Invocation

The `runner` is used to execute build scripts. it is invocated like this

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

assuming the current work directory is the texttttAccelerator 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`.

A.2.2 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]
```

A.3 Urd

A.3.1 database

The `passwd` file.

A.3.2 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

A.4 Workdirs

Workdirs is where jobs are stored. 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.

Any directory empty directory can be a workdir, and by adding it as target workdir, it will be initiated at daemon startup. The initiation process creates a file named `<workdir>-slices.conf`, that will contain the number of slices that is used for the workdir.

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, since the output of the method is available without knowing the exact jobdir name.

A.4.1 Creating a Workdir

To create a new workdir, stop the `daemon`, add the workdir information to the configuration file, make it the *default_workdir*, and start the `daemon` again.

A.5 Progress Indication

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`. If in `analysis`, it will list all analysis processes that are active at the moment. If iterating over a dataset, there will be information about that, including which dataset in a chain that is currently iterated by which `analysis` process.

Note that `C-t` could be pressed either in the `daemon` or `runner` shells.

A.6 Typical Installation

Traditionally, the Accelerator is installed as a `git submodule` to the current project it is used in. This makes it possible 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`. These files are version controlled using `git`. The `accelerator` directory is a `submodule`, which means that the project repository is storing the repository location *and* the commit of the Accelerator.