ExAx: The Accelerator

User's Reference

2020-12-14, **draft**

— Fast and Reproducible Data Processing —

Anders Berkeman, Carl Drougge, and Sofia Hörberg

version: 0b85fa06

Document History

date	git hash	description
2018-04-23	`772990f4`	First open version.
2018-05-28	`a6d6750b`	Updated parts of Urd chapters.
2019-06-11	`24b4b6c2`	Major makeover.
2019-06-24	`db47cc19`	More on `depend_extra`, valid column names, apped-ing columns in synthesis, and some formatting.
2020-01-22	`04fb2389`	New class based interface.
2020-02-14	`562e8d10`	PyPI dev release.
2020-04-22	`54031595`	PyPI dev release. Relative paths in config-file, `link_result()` in build scripts, `Job.files()`, s/daemon/server/g.
2020-10-12	`afa950a9`	PyPI dev release, updated chapters 3 – 8. Missing in this version: `board` and latest class additions.
2020-12-14	`0b85fa06`	Update Appendix A and B.

Contents

Chapter 1

Introduction

The Accelerator is a tool for fast reproducible data processing, capable of working at high speed with terabytes of data with billions of rows on a single computer. The speed in combination with its unique capabilites to ensure reproducibility makes the Accelerator a good choice for tasks where it is important to keep track of how data and results are connected. Typical applications include all kinds of data analysis work as well as live production systems for tasks such as recommendation systems, and more. The Accelerator has a small footprint, few dependencies, and runs on laptops as well as rack servers.

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

Data set sizes in these projects range from a few hundred lines up to several tens of billions rows and many 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, and about 1000 new commits have been created since the Accelerator was open sourced. Extensive testing has been done by Stefan Håkonsson. The Accelerator is written in Python, with the exception of some critical parts that are written in the C programming language.

1.1 Main Design Goals

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

> *Parallel processing* should be made simple. Modern computers come with several cores, it should be straightforward to make use of them.
>
> Data rates should be as fast as possible, i.e. close to the hardware bounds. It should be possible to process *large datasets*, even on commodity hardware.
>
> Any processing step should be *reproducible*. The Accelerator maps any output result to its corresponding input data and processing source code.
>
> Never recompute old results, always "recycle" old jobs, when possible. Also, *sharing results* between multiple users should be effortless.
>
> *Organise* and keep track of all jobs, files, and results in order to work with projects having 100.000s of input files and lots of programs and scripts processing them.

In addition, the Accelerator is originally designed to be used at all levels of a project, including data analysis, algorithm development, as well as production. Nevertheless, it still excels as a pure data analysis or data processing tool.

[1]CSV is short for Comma Separated Values, but any separator character can be used. CSV files store data into rows and columns of text. Classical "databases" could be generated from, and dumped to, CSV-files.

Chapter 2

Overview

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

2.1 High Level View

The Accelerator is a client-server based application, and from a high level, it can be visualised like in figure 2.1.

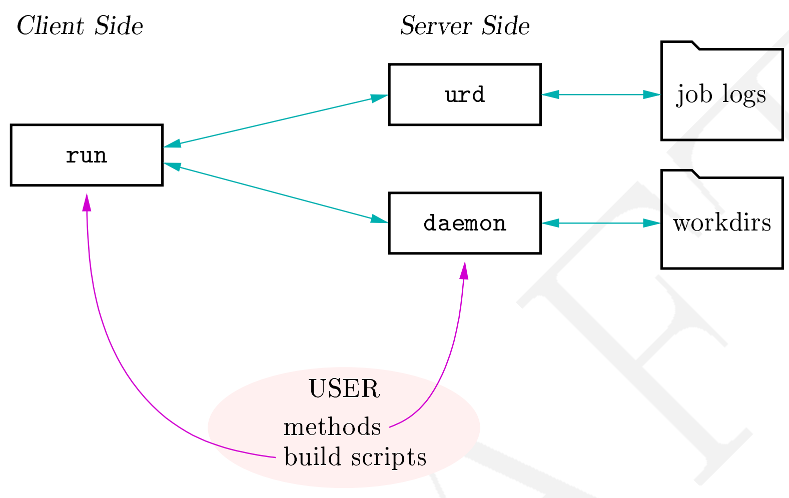

Figure 2.1: High level view of the Accelerator framework. See text for details.

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

In parallel, all jobs covered by a build script may be stored by the `urd` server into the *job logs* file system database. `urd` is also responsible for finding collections, or lists, of related previously executed jobs. The `urd` server ensures reproducibility and transparency, and it will be further discussed in chapter 7.

2.2 Jobs

The basic operation of the Accelerator is to execute small Python programs called *methods*. In a method, a few special functions 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*.

Jobs are stored in *job directories*. A dedicated directory will be created for each new job, and the directory will contain all information regarding the job, such as its input parameters, stored files, return values, profiling information, and more.

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

2.2.1 A Very Simple Job: "Hello, World"

The following example method is very simple. It does not take any input parameters and does almost nothing, it will just return the string "`hello world`" and exit.

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

In order to get the method to execute, it is called from a *build script* looking something like this

```
def main(urd)
    job = urd.build('hello_world')
    print(job.load())
```

The `urd` object contains functions for job building and organisation, and is described in chapter 7 and section B.7. Remember that during the job build process, a job directory is created that will contain everything associated with the build.

When execution is completed, a job object, of type `Job`, is returned to the user. This object provides a convenient interface to the data in the corresponding job directory, and contains member functions such as `.load()`, that is used in the example to read back the returned value from the job.

2.2.2 Jobs Can Only be Run Once

If the build script is executed again, the `hello_world` job will not be re-built, simply because the Accelerator remembers that the job has been built in the past, and its associated information is stored in a job directory. Instead, the Accelerator immediately returns a job object representing 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 method executing again, either the source code or input parameters need to change. If there are changes, the method will be re-executed, and a new job will be created that reflects these changes.

2.2.3 Back to the "Hello, World" example

Figure 2.2 illustrates the dispatch of the `hello_world` method. The created job gets the *jobid* `test-0`, and parts of the corresponding job directory information is shown in green. (Jobids are job identifiers, that are named by their corresponding *workdir* plus an integer counter value.) The job directory contains several files, of which the most important are

`setup.json`, containing job meta information;

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

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

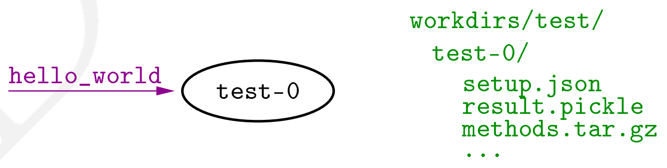

Figure 2.2: A simple hello world program, represented as graph and work directory.

The `Job` class provides a convenient way to access important files in this directory. For example, the job's return value can be loaded into a variable using the `.load()` function, like this

```
def main(urd)
    job = urd.build('hello_world')
    print(job.load())
```

Running this build script will print the string to the `run` program's standard output.

2.2.4 Workdirs and Sharing Jobs

Workdirs are used to separate jobs into different physical locations. The Accelerator can be set up to have any number of workdirs associated, but only one is used for writing.

If the same workdir is entered into two or more different user's configuration files, the workdir and its contents will be shared between the users. Each Accelerator server will update its knowledge about the contents of all workdirs before executing a build script, to make sure that the latest jobs are taken into account. The Urd database, as described in chapter 7, is very useful for sharing job information between users.

2.2.5 Linking Jobs

Using jobs, complex tasks can be split into several smaller operations. Jobs can be connected so that the next job will depend on the result of a previous job or set of jobs, and so on.

To continue the simple example, assume for a second that the "hello world"-job is computationally expensive, and that it returns a result that is to be used as input to further processing. To keep things simple, this further processing is represented by printing the result to standard output. A new method `print_result` is created, and it goes like this

```
jobs = {'hello_world_job',}

def synthesis():
    print(jobs.hello_world_job.load())
```

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

```
def main(urd):
    job1 = urd.build('hello_world')
    job2 = urd.build('print_result', hello_world_job=job1)
```

The `print_result` method then loads the result from the provided job and prints its contents 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` has `test-0` as input parameter, but `test-0` does not know of any jobs run in the future. Hence, arrows point to previous jobs.)

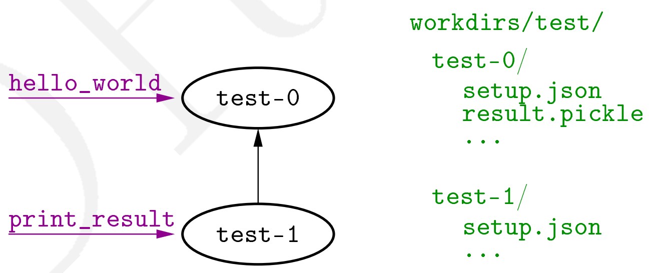

Figure 2.3: Job `test-0`, is used as input to the `print_result` job.

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

2.3 Datasets: Storing Data

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

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

Furthermore, datasets may be *hash partitioned*, so that slicing is based on the hash value of a given column. Slicing on, for example, a column containing some ID string

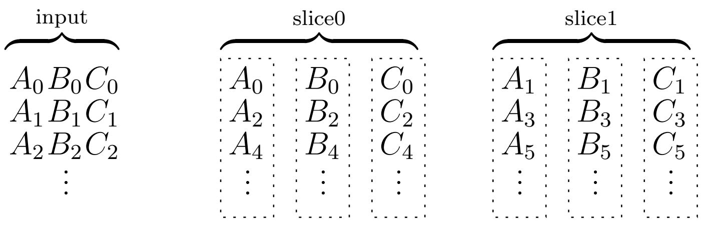

Figure 2.4: A dataset containing three columns, A, B, and C stored using two slices. Each dotted box corresponds to a file, so there are two files for each column, allowing for parallel read of the data using two processes.

will partition all rows such that rows corresponding to any particular ID is stored in a single slice only. In many practical applications, hash partitioning makes parallel processes independent, minimising the need for complicated merging operations. This is explained further in section 5.2.

2.3.1 Importing Data

A project typically starts with *importing* some data from a file on disk. The bundled method `csvimport` is designed to parse a plethora of "comma separated values"-file formats and store the data as a dataset. See figure 2.5. The method takes several input options in addition to

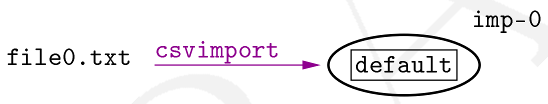

Figure 2.5: Importing `file0.txt`.

the mandatory filename to control the import process. Here is an example (non-simplified) invocation

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

When executed, the created dataset will be stored in the resulting job directory, 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. In general, a job could contain any number of datasets, but a single dataset is a common case.

2.3.2 Linking Datasets, Chaining

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

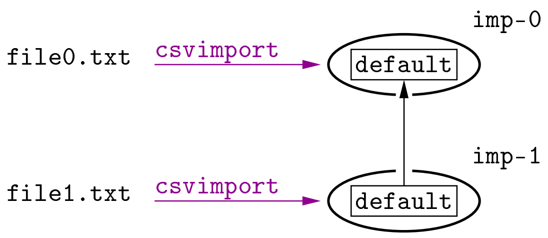

Figure 2.6: Chaining the import of `file1.txt` to the previous import of `file0.txt`.

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

Linking datasets containing related content is called *chaining*, and this is particularly convenient when dealing with data that grows over time. A good example is any kind of *log* data, such as logs of transactions, user interactions, and similar. Using chaining, datasets can be with more rows just by linking, which is a lightweight constant time operation.

2.3.3 Adding New Columns to a Dataset

In the previous section it was shown that datasets can be chained and thereby grow in number of rows. A dataset chain is created simply by linking one dataset to the other, so the overhead is minimal. In this section it is shown that it is equally simple to add new columns to existing datasets. Adding columns is a common operation and the Accelerator handles this situation efficiently using links.

The idea is very simple. Assume a “source” dataset to which one or more new columns should be added. A new dataset is created containing *only* the new column(s), and while creating it, the constructor is instructed to link all the source dataset’s columns to the new dataset such that the new dataset appears to contain all columns from both datasets. (Note that this linking is similar to but different from chaining.)

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

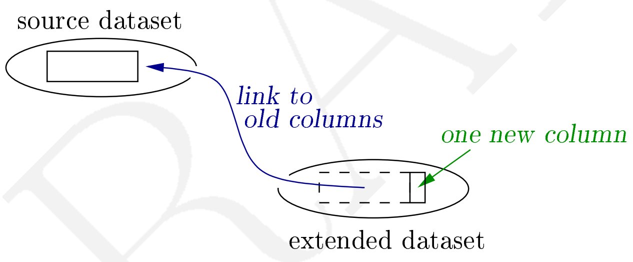

Figure 2.7: Adding one new column to the source dataset.

A common case is to compute new columns based on existing ones. In this case, values are written to the new columns in the new dataset while reading from the iterator iterating over the existing columns in the source dataset. This will be discussed in detail in section 5.10

2.3.4 Multiple Datasets in a Job

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

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

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

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

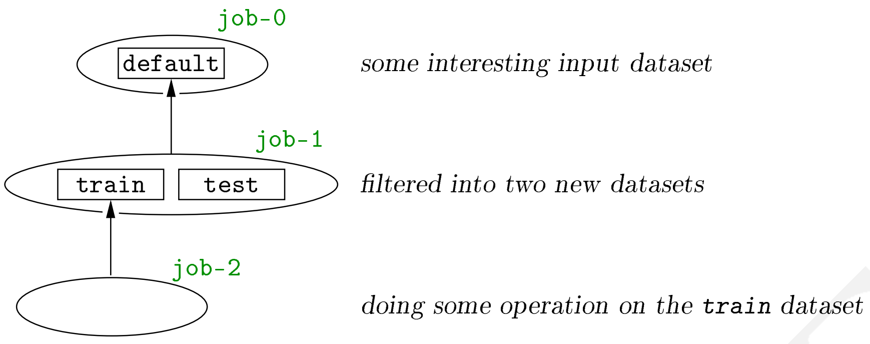
job-0
default
some interesting input dataset
job-1
train
test
filtered into two new datasets
job-2
doing some operation on the train dataset

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

2.3.5 Parallel Dataset Access and Hashing

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

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

Methods may be designed simpler and more efficient using hash partitioning, since the partitioning ensures some kind of data independence between slices and processes. If, however, the same method is used on data that is not partitioned in the expected way, it will not process the data correctly. To ensure that an assumption about hash partitioning is correct, there is an optional `hashlabel` parameter to the iterators that will cause a failure if the supplied column name does not correspond to the dataset's hashlabel.

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

2.3.6 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* (handling all kinds of encodings), and *json* types for storing arbitrary data collections. Most of these types come with advanced parsers, making importing data from text files straightforward with deterministic handling of errors, overflows, and so on.

2.3.7 Dataset Attributes

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

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

and so on.

2.4 Iterators: Working with Data

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

This section shows how iterators are used for reading data, how to take advantage of slicing to have parallel processing, and how to efficiently create new datasets.

2.4.1 Iterator Basics

Assume a dataset that has a column containing movie titles named `movie`, and the problem is to extract the ten most frequent movies. Consider the following complete example

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

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

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

2.4.2 Parallel Execution

The Accelerator is much about parallel processing, and since datasets are sliced, the program can be modified 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, `.iterate` is run 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 `Counter`s, `set`s, and composed types like `set`s of `Counter`s, and so on. For larger datasets, this version will run much faster.

2.4.3 Iterating over Several Columns

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

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

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

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

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

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

Here, `columns` will be a list of values, one for each column in the dataset.

2.4.4 Iterating over Dataset Chains

The `iterate` function is used to iterate over a single dataset. There is a corresponding function, `iterate_chain`, that is used for iterating over chains of datasets. This function takes a number of arguments, such as

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

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

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

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

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

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

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

2.4.5 Job Execution Flow and Result Passing

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

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

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

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

Figure 2.9 shows the execution order from top to bottom, and the data passed between functions in coloured branches. `prepare()` is executed first, and its return value is available to both the `analysis()` and `synthesis()` functions. There are `slices` (a configurable parameter) number of parallel `analysis()` processes, and their output is available to the `synthesis()` function, which is executed last.

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

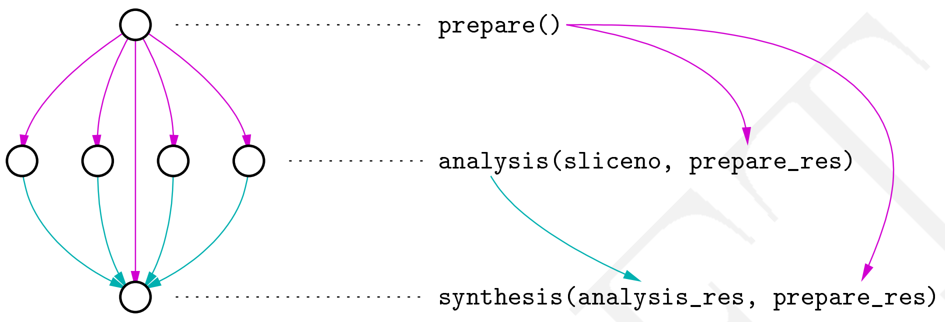
prepare()
analysis(sliceno, prepare_res)
synthesis(analysis_res, prepare_res)

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

2.4.6 Job Parameters

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

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

`options`, a dictionary of options; and

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

See Figure 2.10. Parameters are entered as global variables early in the method's source.

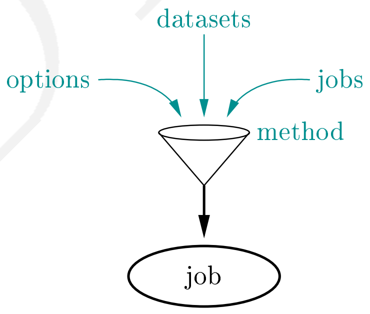

Figure 2.10: Execution flow of a method. The method takes optionally three kinds of parameters: `options`, `jobs`, and `datasets`.

2.5 A Class Based Programming Model

See figure 2.11.

2.6 Accelerator Exceptions

There are a number of custom defined `Exceptions` in the Accelerator code in order to simplify debugging.

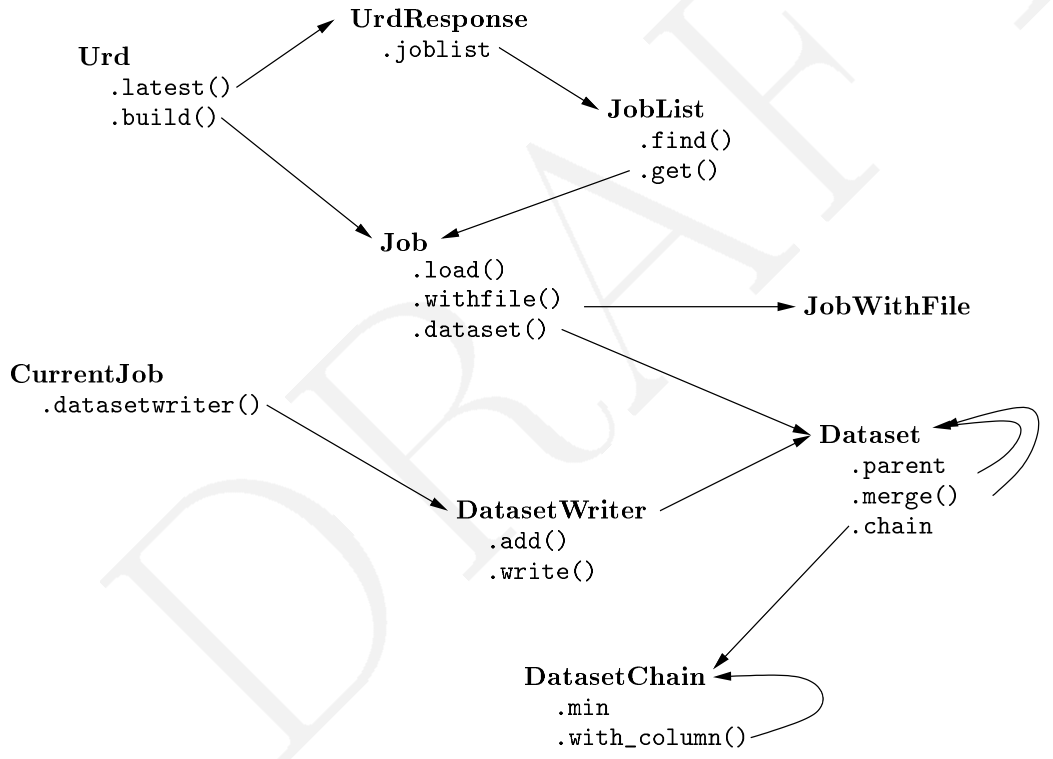

Figure 2.11: Most important relations between classes.

Chapter 3

Basic Build Scripting

Build scripts are used to execute jobs and control the job flow on the Accelerator. This chapter describes the basics of job building. More advanced features, using the Urd server, are presented in chapter 7.

3.1 Build Scripts

Build scripts are stored in *method directories*. The default build script is named "`build.py`", all other build scripts have to start with the string "`build_`". Build scripts are executed using the `run` command. For example,

```
ax run example
```

will look for a file named `build_example.py` and execute it, while the command

```
ax run
```

by itself will execute the default build script. (The `run` command is described in section 9.3.)

The `run` command will load the build script and execute its `main()` function. This function takes a *mandatory* argument named `urd`, so a basic build script looks like this

```
def main(urd):
    # do something here...
```

The `run` command inserts an object of the `Urd` class as the argument to the `main()` function. This `urd` object has a number of member functions and attributes useful for job building and tracking of old jobs. The `Urd` class is described in chapter 7 and in section B.7.

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

The `.build()` function is used to build a job from a method (i.e. source file). For example, the most simple build script that executes a method, here called `method1`, is

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

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

```
job = urd.build(method,
                options={}, datasets={}, jobs={},
                name='', caption='', workdir=None , **kw)
```

All parameters, except the name of the method, are optional and typically not used very often in practice. Any arguments to a job build can be added as simple *keyword arguments*, and the build process will automatically assign them by name to the corresponding `options`, `datasets`, or `jobs` parameter sets, assuming the naming is unique. Otherwise the argument has to be specified explicitly. Examples will be present in the following sections.

When the job is completed, Urd will record it using the name of the method as key, unless the `name=` is specified. The `name` parameter is particularly useful to tell jobs apart that are based on the same method. A common case would be the `csvimport` method, for example. It is also possible to assign a caption to a job, but this has no functional benefits.

When the job has been successfully built, the `build` function will return a reference of type `Job`. The `Job` class contains member functions and attributes that can be used to extract information, such as generated files or text written to `stdout`, from the job. The `Job` class is described in detail in chapter 4 and in section B.1.

Similarly, if the job to be built already exists in a configured *workdir*, the `build` function will immediately return a `Job` object corresponding to the existing job without executing anything.

3.1.2 Connecting Jobs

Jobs may be connected, or linked, together so that a new job may use the output generated by previous jobs. In this way, jobs could share and re-use previous results, and complicated tasks can be broken down into several smaller jobs. A link is created by feeding the job object from an existing job into the `.build()`-call of a new job, like in this example

```
def main(urd):
    job_import  = urd.build('csvimport', filename='inputfile.txt')
    job_process = urd.build('process',   source=job_import)
```

In the example above, the first job, `csvimport`, imports the file "`inputfile.txt`". The second job, `process`, takes the imported dataset as input for further processing.

3.1.3 Replaying Build Scripts

When the example build script from the previous section is run, both the `csvimport` and the `process` jobs will be built. But what happens if the same build script is run a second time? Remember now that the Accelerator stores all jobs in associated workdir(s). If there has been no changes to the code since the last run, the Accelerator will immediately find the job reference to the `csvimport` without needing to execute it again. The reference will be input to the `.build()` call of the second method, and since the Accelerator has seen this call before too, it will immediately look up the reference to this job as well, instead of executing it. A second run of the build script will only take a fraction of a second to execute, but it will still return all job references.

On the other hand, if something has been modified, such as a method's source code or any of the input parameters, the affected job(s) will be re-executed. For example, assume that the `csvimport` build call is modified to import "`anotherfile.txt`" instead. This will cause the `csvimport` method to be executed again, leading to a new job with a new jobid. This job reference is input to the `process` job, causing it to be re-executed too. Here, both jobs were re-executed, since they both were affected by the modification, but in general, only those jobs affected by the modification will be re-executed!

A successful "replay" of a build script ensures the integrity and dependencies of all involved calculations. If there are no changes, the same result remains. If, however, some of the code has been modified, the Accelerator will compute new jobs to reflect the new situation. The result may be different, and the user is notified.

3.2 Working with Build History: `urd.joblist`

Information about previously executed jobs is stored in the `urd.joblist` variable. This variable is of type `JobList`, which is basically a standard ordered Python `list` with some additional features for searching, profiling and pretty-printing. The `JobList` class is further explained in section B.3.

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

Create a `JobList` and pretty-print it

```
def main(urd):
    job1 = urd.build('first')
    job2 = urd.build('second', first=job1)
    print(urd.joblist.pretty)
```

which results in

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

(The actual jobids will most likely be different.) The name in the `joblist` is either the name of the method, or, if present, the name given explicitly using the `urd.build(name=)` option.

3.2.2 Finding Jobs in a Joblist

There are several ways to extract jobs or list of jobs from a `joblist`.

Using `find` to Extract Jobs to a New Joblist

The `find()` function finds matching jobs and returns them in a new `JobList`. For example,

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

will create a `JobList` of all `csvimport` jobs in `urd.joblist`.

Using `get` to Potentially Extract a Single Job

The `get` function will return a job reference to the most recent matching job. For example,

```
job = urd.joblist.get('csvimport')
```

If no matching job is found, `get` will return `None`.

Using Square Brackets to Extract a Singe Job

Accessing jobs directly with a key like this

```
job = urd.joblist['csvimport']
```

is similar to `get`, but will return an error if a matching job is not found.

3.2.3 Indexing and Slicing a `JobList`

Since the `JobList` class is an extension of the (list) class, it is possible to do some `list`-like operations on joblist objects.

A common special case is to provide an integer to the `.get()` function. This will cause a lookup of the item with the corresponding index in the list. It is often used like this

```
latest_job = urd.joblist.get(-1)
```

as a way to return the last item in the joblist, should it exist, or `None`.

It is also possible to index and slice like in the following examples

```
joblst = jl[3]
```

or

```
joblst = jl[-2:]
```

3.3 Configuration Information: `urd.info`

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

name	description
`slices`	Configured number of slices.
`urd`	An URL to the Urd server.
`result_directory`	see section A.4.
`input_directory`	see section A.4.

3.4 Summary

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

Chapter 4

Jobs

4.1 Definitions

4.1.1 Methods and Jobs

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

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

In the Accelerator context, the notation is as follows

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

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

– any number of output files created by the running method, as well as

– a number of job meta information files containing all information that was needed to run the job in the first place.

The exact contents of the job directory will be discussed in section 4.1.3, but note here that the directory contains everything needed to run a specific piece of code with inputs and source code, *as well as* any output generated during execution.

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

– a *build script*, see chapter 7, or

– by a method, using *subjobs*, see section 4.9

The following figure illustrates how a job `example-0` is built from the method `a_method.py`. The job is stored in the `example` work directory. The job identifier (in this case `example-1`) is always unique so that it can be used as a reference to that particular job.

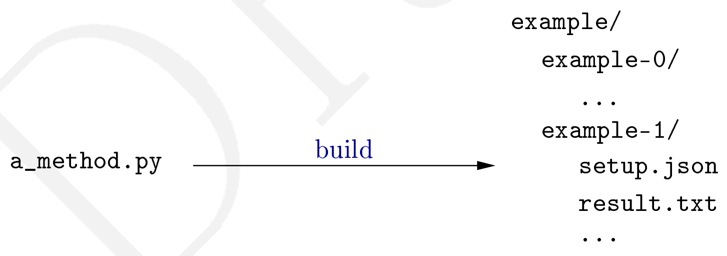

Figure 4.1: When a method is built, a job directory is created in the target work directory, containing files with all data an meta information regarding the job.

4.1.2 Jobids

A **jobid** is a string that can be used as a reference to a job. This unique string is created by appending an incrementing integer to the name of the work directory in which the job is stored. In the example above, the job is uniquely identified by the string `example-0`.

4.1.3 Work Directories and 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* (i.e. work directory) 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
        OUTPUT/
        datasets.txt
        default/
```

The following table shows examples of files commonly found in a job directory.

name	description
`setup.json`	Contains information about the job build, including name of method, input parameters, and, after execution, some profiling information.
`post.json`	Contains profiling information, and is written only if the job builds successfully.
`method.tar.gz`	All source files, i.e. the method's source and any `depend_extras` are stored in this gziped `tar`-archive.
`result.pickle`	The return value from `synthesis()` stored in the Python "pickle" format.
`default/`	If the job contains datasets, these will be stored in directories, such as for example `default/`, in the root of the job directory.
`datasets.txt`	List of all datasets in job in a human readable format.
`OUTPUT/`	Any output to `stdout` and `stderr` will be stored in the `OUTPUT/` directory.

4.1.4 The `Job` and `CurrentJob` Convenience Wrappers

In order to simplify access to job directory data, common job data operations are made available by the `Job` class. There is also an extended version of this, called the `CurrentJob` class, that also contains information and helper functions to a *running* job. See section B.1 for details about these classes.

4.2 Python Packages

Methods are stored in standard Python packages, i.e. in directories that are

– reachable by the Python interpreter, and

– contain the (perhaps empty) file "`__init__.py`".

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

– the package must contain a file named `methods.conf`, and

– the package must be added to the Accelerator's configuration file under the key "`method packages`", see section A.4.

4.2.1 Creating a new Package

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

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

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

For security reasons, the Accelerator only looks for packages explicitly specified in the configuration file using the "`method packages`" assignment. See chapter A.4 for detailed information about the configuration file.

4.3 Method Source Files

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

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

4.3.1 Creating a New Method

In order to create a new method, follow these steps

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

4.3.2 Making Methods Executable: `methods.conf`

The file `methods.conf` provides an easy way to specify and limit which methods (source files) that can be executed, and optionally, which Python interpreter that should be used for each of them. Methods not specified in `methods.conf` cannot be executed. (This is something that makes a lot of sense in any production environment where control of what is executable is key.)

The `methods.conf` is a plain text file with one entry per line. Any characters from a hash sign ("`#`") to the end of the line is considered to be a comment. It is permitted to have any number of empty lines in the file. Available methods are entered first on a line by stating the name of the method, without the `a_` prefix and `.py` suffix.

The method name can optionally be followed by one or more whitespaces and a name specifying the actual Python interpreter that will be used to execute the method. This is particularly useful to have individual methods run from individual virtual environments. Thus, each method can run from its own virtual environment, with its own dependencies. A list of valid Python interpreters is defined in the configuration file using the key "`interpreters`", see section A.4.

The default interpreter is selected if the second field is left empty, where default corresponds to the one that the currently running Accelerator server is using. The Accelerator and its `standard_methods` library are compatible with both Python 2 and Python 3.

Here is an example `methods.conf`

```
# this is a comment

test2                # will use default Python
test3           py3  # py3 as specified in accelerator.conf
testx           tf   # a Tensorflow virtual env defined in accelerator.conf
#bogusmethod    py3
```

This file declares three methods corresponding to the files `a_test2.py`, `a_test3.py`, and `a_testx.py`. These are the only methods that can be built in this method package. Note that some methods are using virtual environments specified in the Accelerator's configuration file.

4.4 Job Building or Job Recycling

Since the Accelerator keeps track of a job's dependencies and results, it can in an instant determine if a job to be built has been built before. If the job has been built before, the Accelerator will immediately return a job instance to the existing job. Otherwise, the job will first be built, and then a job instance will be returned.

4.4.1 Job Already Built Check

As shown in chapter 3, jobs are built using the `.build()` function, like this

```
def main(urd):
    urd.build('themethod', parameter1=..., ...)
```

Prior to building a method, the Accelerator checks if an equivalent job has been built in the past. 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 parameters and compared to all jobs already built. Only if the hash and input parameter combination is unique, i.e. it has not been seen before, will the method be executed. The `.build()`-function returns an instance of type `Job`. To the caller, it is not apparent if the job was just built or if it was built at an earlier time.

4.4.2 Depend on More Files: `depend_extra`

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

```
from . import my_python_module

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

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

If the Accelerator suspects that a `depend_extra`-statement is missing, it will suggest adding it by printing a message in the output log like this:

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

The point of this is to make the user aware that the method depends on additional files that are currently not taken into account in the build check hashing. The warning is removed by putting the `myfuncs` file in a `depend_extra` list of the `test` method.

4.4.3 Avoiding Rebuild: `equivalent_hashes`

A change to a method's source code will cause a new job to be built upon running `.build()`, but sometimes it is desirable to modify the source code *without* causing a re-build. This happens, for example, when new functionality is added to an existing method, and re-computing all jobs is not an option. If functionality remains the same, existing jobs strictly do not need to be re-built. For this situation, there is an `equivalent_hashes` dictionary that can be used to *manually* specify which versions of the source code that are equivalent. The Accelerator helps creating this dictionary, if needed. This is how it works.

1. Find the hash `<old_hash>` of the existing job in that job's `setup.json`.
2. Add the following line to the updated method's source code

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

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

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

4. Copy `<current_hash>` into the `equivalent_hashes`:

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

This line now tells that `current_hash` is equivalent to `old_hash`, so if a job with the old hash exists, the method will not be built again. Note that the right part of the assignment is actually a list, so there could be any number of equivalent versions of the source code. From time to time, this has been used during development of the Accelerator's `standard_methods`, when new features have been added that do not interfere with existing use.

4.5 Method Execution

Methods are executed using the `build()` call, either from a `build script`, or from another method as a `subjob`. Methods typically takes input parameters, and they may generate return values and produce output files as well as output to `stdout` and `stderr`.

4.5.1 Input Parameters

There are three kinds of method input parameters assigned by the `build()` call: `jobs`, `datasets`, and `options`. These parameters are stated early in the method source code and are *global*, meaning that they do not need to be included as parameters to the functions in a method. Here is an example parameter set

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

The input parameters are populated by the builder when the `run` command is executed. Section 4.8 and 4.10 provide detailed descriptions of all parameters.

4.5.2 Functions Reserved for Execution Flow

During execution, methods are not run from top to bottom. Instead, there are three reserved functions that are called by the method dispatcher controlling the execution flow. These functions are

`prepare()`,

`analysis()`, and

`synthesis()`.

4.5.3 Execution Order

The three functions `prepare()`, `analysis()`, and `synthesis()` are called one at a time in that order. `prepare()` and `synthesis()` execute 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. It is discouraged to use `prepare()` only.

4.5.4 Function Arguments

There are some optional arguments that can be passed into the executing functions `prepare()`, `analysis()`, and `synthesis()` at run time. All functions have access to these

- `job`, which is an instance of the current job, and
- `slices`, an integer holding the total number of slices.

while only `analysis()` has access to

- `sliceno`, which provides a unique number to each parallel `analysis()`-process. This is also a **mandatory** argument to `analysis()`.

The `job` instance contains information and helper functions regarding the current job. The object is of type `CurrentJob`, which is an extension of the `Job` class used for job instances that are not in the execution stage.

The `analysis()` function (and only the `analysis()` function) takes the mandatory 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 it is commonly used when accessing sliced datasets or doing other parallel processing tasks, see for example chapter 6 for its use in dataset iterators.

4.5.5 Parallel Processing: The `analysis()` Function and Slices

The number of parallel analysis processes is set by the `slices` parameter in the Accelerator's configuration file. The input parameter `sliceno` to the `analysis()` function is the unique identifier for each parallel function call, and its value is in the range from zero to the number of slices minus one. When processing Accelerator *datasets*, the idea is that each dataset slice should have exactly one corresponding `analysis()` process, so that all the slices in the dataset can be processed in parallel.

4.5.6 Return Values

Return values may be passed from one function to any function that will execute later. To be specific, what is returned from prepare is called `prepare_res`, and can be used as input argument to `analysis()` and `synthesis()`. Furthermore, the return values from `analysis()` are 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 using the name "`result.pickle`". Here is an example of return value passing

```
options = dict(length=4)

def prepare():
    # options is a global variable
    return options.length * 2

def analysis(sliceno, prepare_res)
    return prepare_res + sliceno

def synthesis(analysis_res, prepare_res):
     return sum(analysis_res) + prepare_res
```

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. Creating persistent intermediate files is the topic of section B.1.17, however.

4.5.7 Merging Results from `analysis()`

It is common that results from different slices needs to be merged together. The Accelerator provides a relatively general function for merging sliced data structures. Consider this example

```
# create a set of all users
datasets = ('source',)

def analysis(sliceno):
    return(set(datasets.source.iterate(sliceno, 'user')))

def prepare(analysis_res):
    return analysis_res.merge_auto()
```

Here, each `analysis()` process creates a set of `users` seen in that *slice* of the `source` dataset. In order to create a set of all `users` in the dataset, all slice-sets have to be merged. Merging can be implemented using for example a `for`-loop, but the actual merging operation is dependent of the actual data type, and writing merging functions is error prone. (So we've been told!) Therefore, `analysis_res` has a function called `merge_auto()`, that is recommended for merging. This function can merge most data types, and even merge container variables in a recursive fashion. For example, a variable defined like this

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

(a `dict` of `dicts` of `sets`) is straightforward to merge using `merge_auto()`. The function works on many data types and is less error-prone than writing special mergers every time they are needed.

4.5.8 Standard Out and Standard Error

Anything sent to `stdout` or `stderr` during job execution will be sent *both* to the terminal in which the Accelerator server was started, *and* to a file in the current job directory. This covers, for example, anything output from Python's `print()`-function. Thus, the job directory contains a complete log of all printed output from the execution phase!

Output is collected in the job directory in a subdirectory named `OUTPUT`, and it is made available using the `.output()` function, see section 4.6.7. The `OUTPUT` directory is created *only* if anything was output from the job to `stdout` or `stderr`, otherwise it does not exist. Inside the directory there may be files like this

```
job-x/
   OUTPUT/
      prepare     # created if output in prepare()
      synthesis   #                      synthesis()
      0           #                      analysis() slice 0
      3           #                                       3
```

No empty files will be created.

4.6 The `Job` and `CurrentJob` Classes

The `Job` and `CurrentJob` classes provide functionality for easy access to data and datasets stored in a job directory. (Datasets will be covered in chapter 5). The `CurrentJob` is an extension of `Job` that adds special functions that are useful to a method during execution. This section provides a taste of the most common operations that are provided. See section B.1 for a complete list of the functionality.

Instances of these two classes are used extensively in Accelerator projects. In a build script every reference to a job, such as the return value of the `.build()` function or any job retrieval using the Urd database are of type `Job`. Any job passed as input parameter to a `.build()`-call will appear as a `Job` instance inside the running method. Instances of the `CurrentJob` class are provided when asking for a `job` input parameter in `prepare()`, `analysis()`, or `synthesis()`.

4.6.1 Writing and Reading Serialised Data

Data structures may be serialised and written to disk using `job.save()` and `job.json_save()`, with corresponding `.load()` and `.json_load()` functions, where the first writes a Python "pickle" file, and the latter uses JSON encoding. Here is an example of how to write files in a running job

```
def synthesis(job):
   job.save('a string to be written', 'stringfile')
   job.json_save(dict(key='value'), 'jsonfile')
```

The corresponding `job.load()` and `job.json_load()` functions can be called *both* in methods *and* build scripts. For example

```
jobs = ('anotherjob',)

def synthesis():
    jobs.anotherjob.load('stringfile')
```

will load a file from another job into the currently running method, while

```
def main(urd):
    job = urd.build('example')
    x = job.load('thefile')
```

will load data stored by the `example` method using the filename `thefile.pickle` into the build script.

4.6.2 Writing and Reading Serialised Data in Parallel

If data is read and written in the parallel `analysis()`-function, the argument `sliceno=` may be used to write one file for each slice. For example

```
def analysis(sliceno, job)
    data = ...
    job.save(data, 'filename', sliceno=sliceno)
```

Similarly, another job can then read one of these files per slice as follows

```
def analysis(sliceno):
    data = jobs.anotherjob.load('filename', sliceno=sliceno)
```

Writing "sliced" data results in n files on disk, where n is equal to the number of slices set in the configuration file. Each filename is extended with a human readable number that corresponds to the slice that the file's data belongs to.

4.6.3 General File Access

The `.open()` function corresponds to the built in `open()` with the addition that it cannot write to completed jobs, and that files written using it are book-kept in the job. It has to be used as a context manager, i.e. using the `with` statement, for example like this

```
def synthesis(job):
    with job.open('filename, 'wb') as fh:
        fh.write(...)
```

4.6.4 Accessing a Job's Return Value

The default behaviour of a job instance's `.load()` function is to read the return value from the job's `synthesis()` function, like this

```
def main(urd):
    job = urd.build('example')
    x = job.load()
```

This works both in build scripts and inside methods, and is a convenient way to access data generated by a job.

4.6.5 Accessing a Job's Datasets

Using the Job class, it is straightforward to access datasets in other jobs. For example

```python
def main(urd):
    job = urd.build(...)

    # This will print a list of all dataset instances in the job.
    print(job.datasets())

    # This will return a dataset instance of the job/training dataset.
    ds = job.dataset('training')
```

This works both in running methods and in build scripts

4.6.6 Accessing a Job's Options and Parameters

There are two sources of parameters to a running method,

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

parameters assigned by the Accelerator when the job starts building.

All these parameters are available using `job.params`. For example

```python
jobs = ('anotherjob',)

def synthesis():
    print(jobs.anotherjob.params.options)
```

will print the `options` dictionary that was fed to the `anotherjob` at build time, for example

```
{'message': 'Hello world!'}
```

A complete print of a job's `.params` may look like this

```json
{
    "starttime": 1602061144.081299,
    "endtime": 1602061147.2101562,
    "exectime": {
        "analysis": 0,
        "per_slice": [],
        "prepare": 0,
        "synthesis": 3.111,
        "total": 3.111
    },

    "caption": "",
    "hash": "9189b775e190826f3dc6ea85ea252a9e3d647185",
    "jobid": "beast-325",
    "method": "plot_walk_narrowbeams",
    "package": "dev",
    "seed": 3621427863964846452,
    "slices": 4,
    "version": 3,
    "versions": {
        "accelerator": "2020.10.3.dev1",
        "python": "3.5.2 (default, Jul 17 2020, 14:04:10) \n[GCC 5.4.0 20160609]",
        "python_path": "/home/eaenbrd/checkout/project_beast.acc/venv/bin/python3"
    },

    "options": {
        "gnbdir": 0.3839724354387525,
        "gnbpos": [
            57.71525125357654,
            12.83168
        ]
    },
    "datasets": {
        "source": "beast-222"
    },
    "jobs": {
        "background": "beast-4"
}
```

and a description of its keys

name	description
`package`	Python package for this method
`method`	name of this method
`jobid`	jobid of this job
`starttime`	start time in epoch format
`endtime`	end time in epoch format
`exectime`	various exec times
`caption`	a caption
`slices`	number of slices of current Accelerator configuration
`seed`	a random seed available for use[1]
`hash`	source code hash value
`versions`	Accelerator and Python versions
`options`	input parameter
`datasets`	input parameter
`jobs`	input parameter

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

Note how well covered input parameters and settings are, all the way down to the specific Python interpreter.

4.6.7 Accessing Job Output

Anything written to `stderr` or `stdout` during job execution is available using the `.output()` function. Here is an example

```python
def main(urd):
    job = urd.build('example')
    print(job.output())
```

With no argument, the `.output()` function returns all output. Particular parts of the output can be selected using the options, `'prepare'`, `'analysis'`, `'synthesis'`, or a digit specifying a particular slice.

4.6.8 Reading Post Data

The `.post` attribute contains information such as starttime, execution time (per function and slice), written files and subjobs for a job. For example

```python
def main(urd):
    job = job.build('example')
    print(job.post.exectime)
```

4.7 Converting Between Jobs and Datasets

Sometimes it is necessary to find the job that created a particular dataset, or access one of the other datasets in a job given a certain dataset.

4.7.1 From Dataset to Job

```
  job = ds.job
```

4.7.2 From Job to Dataset

```
  ds = job.dataset("datasetname")
```

4.7.3 From Dataset to Dataset (in same Job)

```
  ds = ds.job.dataset("datasetname")
```

4.8 Method Input Parameters

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

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

The input parameters are populated by the builder using name matching when the job build is initiated, see chapter 7 for more information.

The `jobs` parameter list is used to input references to other jobs, while the `datasets` parameter list is used to input references to datasets. These parameters are populated by the build call.

The `options` dictionary, on the other hand, is used to input any other type of parameters to be used by the method at run time. Options does not necessarily have to be populated by the build call. Instead, "default values" may be used as "global constants" in the method. An option assigned by the build call will override the default assignment.

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

```
jobs.accumulated_cost
datasets.transaction_log
options.length
```

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

4.8.1 Input Jobs

The `jobs` parameter is a tuple of job references linking other jobs to this job. In a running job, each item in the `jobs` tuple is of type `Job`, and it is possible to used them directly as references to corresponding jobs. All items in the `jobs` tuple must be assigned by the builder to avoid run time errors.

If the number of input jobs is not constant or known beforehand, they can be represented as a list, like in this example

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

where `source` is a single job reference, whereas `alistofjobs` is a list of job references.

4.8.2 Input Datasets

The `datasets` parameter is a tuple of links to datasets. In a running job, each item in the `datasets` variable is of type `Dataset`. The `Dataset` class is further described in chapter 5. All items in the `datasets` tuple must be assigned by the builder to avoid run time errors.

Similar to `jobs`, one can represent an unknown number datasets using a list, like this

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

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

4.8.3 Input Options

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

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

Options are straightforward to use and quite flexible. A formal overview is presented in section 4.10.

4.8.4 A Specific File From Another Job: `JobWithFile`

Any specific file from an existing job can be input to a new job at build time using `job.withfile()`. Here is an example

```
def main(urd):
    job = urd.build('example4')
    urd.build('example5',
              firstfile=job.withfile('myfile1', sliced=True),
              secondfile=job.withfile('myfile2'))
```

Inside the method, the `option` part is defined like this

```
from accelerator import JobWithFile
options=dict(firstfile=JobWithFile, secondfile=JobWithFile)
```

The `.withfile()` function requires a filename and takes two optional 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. (Creating sliced files is described in section 4.6.2.)

Here's how to use the `JobWithFile` object in a running job. First, loading a file is done using `.load()`, like this

```
from accelerator import JobWithFile
options=dict(firstfile=JobWithFile, secondfile=JobWithFile)

def analysis(sliceno):
    print(options.firstfile.load(sliceno=sliceno))

def synthesis():
    print(options.secondfile.load())
```

There is also an `.json_load()` function for JSON-files. To get the full filename to the file, use `.filename()`

```
    print(options.firstfile.filename(sliceno=3))
    print(options.secondfile.filename())
```

And finally, there is also a wrapper around `open()`, so it is possible to do

```
    with(options.firstfile.open(), 'rb') as fh:
        data = fh.read()
```

4.9 Subjobs

Jobs may launch jobs, i.e. a running job may build other jobs in a recursive manner. As always, if the jobs have been built already, they will be linked in immediately. If the build of a subjob fails, the building job will be invalidated. Subjobs are built like in this example

```
from accelerator import subjobs

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

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

There are three catches, though.

1. Dataset instances to datasets created in subjobs will not be explicitly available to the "to level" build script. The workaround is to link the dataset to the building method like this

   ```
   from accelerator import subjobs
   def synthesis():
       job = subjobs.build('create_a_dataset')
       ds = job.dataset(<name>)
       ds.link_to_here(name=<anothername>)
   ```

   with the effect that the building job will act like a dataset, even though the dataset is actually created and stored in the subjob. The `name` argument is optional, the name `default` is used if left empty, since this is the default dataset name.

   It is also possible to override the dataset's *previous dataset* using the `override_previous` option, which takes a job reference (or *None*) to be the new `previous`.

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

   The `link_to_here` call returns a dataset instance.

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

   ```
   from accelerator import subjobs

   depend_extra = ('a_childjob.py',)

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

3. Subjobs are not visible in build scripts, and does not show up in for example `urd.joblist`.

There is a limit to the recursion depth of subjobs, to avoid creating unlimited number of jobs by accident. The limit can be tweaked by modifying the source code, if necessary.

4.10 Formal Option Rules

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

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

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

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

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

3. An input may be left unassigned, unless

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

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

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

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

6. Values are accepted if they are valid input to the type's constructor, i.e. 3 and '3' are valid input for an integer.

7. `None` is always a valid input unless

   - RequiredOptions() and not none_ok set
   - OptionEnum() and not none_ok set

   This means that for example something typed to `int` can be overridden by the builder by assigning it to `None`. Also, `None` is also accepted in typed containers, so a type defined as `[int]` will accept the input `[1, 2, None]`.

8. All containers can be specified as empty, for example `{}` which expects a `dict`.

9. Complex types (like `dicts`, `dicts` of `lists` of `dicts`, . . .) never enforce specific keys, only types. For example, `{'a': 'b'}` defines a dictionary from strings to strings, and for example `{'foo': 'bar'}` is a valid assignment.

10. Containers with a type in the values default to empty containers. Otherwise the specified values are the default contents. Example

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

The following sections will describe typing in more detail.

4.10.1 Options with no Type

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.

4.10.2 Scalar Options

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.

4.10.3 String Options

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

In some situations, an example string is convenient

```
from accelerator 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.

4.10.4 Enumerated Options

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 accelerator 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 accelerator 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 accelerator 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`".

4.10.5 List and Set Options

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.

4.10.6 Date and Time Options

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))
```

4.10.7 More Complex Stuff: Types 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.

4.11 Jobs - a Summary

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

1. Data and metadata relating to a job is stored in a job directory.
2. Job objects are wrappers around job directories, providing helper functions.

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

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

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

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

3. Unique jobs are only executed once.

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

4. Jobs may link to each other using job references.

Which means that jobs may share results and parameters with each other.

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.

7. 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.

8. Jobs may dispatch other jobs.

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

Chapter 5

Datasets

The *dataset* is the preferred way to store row-column data using the Accelerator. The dataset provides fast parallel streaming access to data, and strict typing with many supported types. Furthermore, datasets are lightweight – adding new columns to a dataset, or appending datasets to eachother are instantaneous operations. Using datasets in a program is simple, since all operations are avai through the `Dataset` class.

Datasets are created by methods, and are therefore located inside job directories. There can be any number of datasets in a job. The most obvious way to generate a dataset is using the `cvsimport` method that creates a dataset from an input file. But the fact that a job can create and hold many datasets opens up for a lot of possibilities. The `csvimport` method, for example, creates a dataset from the input data, and it can be instructed to put unparsable lines of input data into a second "bad" dataset.

For performance reasons, datasets are split into several slices, and each data row exists in exactly one of the slices. The actual slicing may be carried out in different ways, like round robin, or randomly, but an interesting approach is to slice according to the hash value of a certain column. Slicing according to a hashed column ensures that all rows with a certain column value always ends up in the same slice. Hash-based slicing often makes completely parallel processing of a dataset possible, since related data is not spread over different slices, and thus no merge or "reduce" stage is required.

5.1 Dataset Internals

On a high level, the dataset stores a *matrix* of rows and columns. Each column is represented by a column name, or *label*, and all columns have the same number of rows. Columns are typed, and there is a wide range of types available. Typing will be introduced in section 5.8.

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

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

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

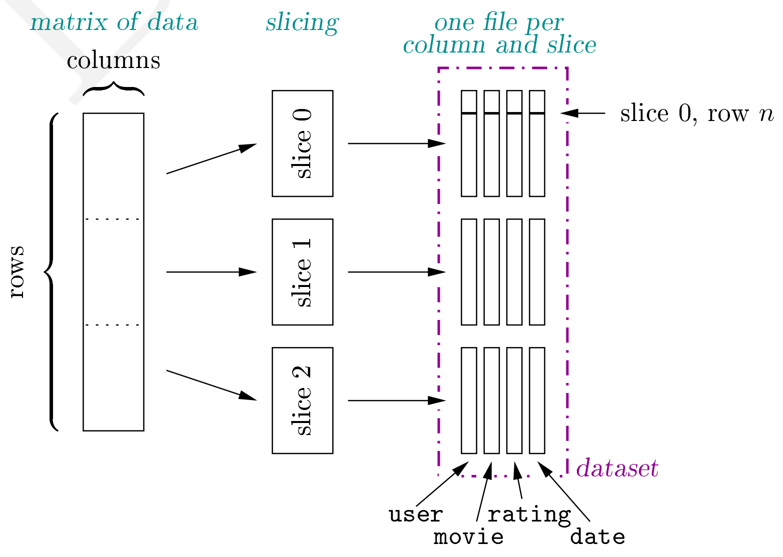

Figure 5.1: A "movie rating" dataset composed of four columns sliced into three slices.

5.2 Slicing and Hashing

As shown in the previous section and in figure 5.1, datasets are *sliced* into disjoint sets, called *slices*, that can be accessed independently. Typically, in a dataset, there is one file on disk for each slice *and* column. The main reason for doing this is performance. All files could be read in parallel, and only files relevant to the task at hand are read.

How to slice a dataset is not unique. There are many ways to carry out slicing. Perhaps the most simple way is to use round-robin, which cycles through the slices when writing, one data item at a time. Round-robin will balance the number of rows per slice as equal as possible, which is a good thing in many scenarios. In semi-mathematical terms, round robin would be

$$n \longrightarrow n \bmod N$$

Meaning that input data row n is stored in slice $n \bmod N$.

Another way is to slice by looking at the values of a fixed single column and put all rows with equal "property" 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, the "property" is a hash function, and the process is called *hash partitioning*. A dataset can be hashed on any single column, as long as its contents is hashable. Written as an equation, it will look like this

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

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

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

The hash function used by the Accelerator is a well-known function called siphash-2-4 that is available from the Accelerator's `gzutil` library

```
from accelerator.gzutil import siphash24

y = siphash24(x)
```

This function is normally only used "under the hood", there should be no need to call it explicitly.

5.3 Chaining

When a dataset is created, it is optional to attach 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. As will be discussed in detail later, this relates to dataset *iterators* (see chapter 6), that may continue iterating over the next dataset in the chain when the current dataset is exhausted. Here is a an example of how `csvimport` jobs can be chained

```
job1 = urd.build('csvimport', filename='file1.txt')
job2 = urd.build('csvimport', filename='file2.txt', previous=job1)
```

In order to maintain high speed when processing long chains, the Accelerator caches chain metadata every 64th dataset. This reduces seek times significantly on rotating disks.

5.4 Dataset as Job 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. See the following example

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

Note the comma used to indicate that this is in fact a Python `tuple` of dataset(s).
It is also possible to input a list or set of datasets, like this

```
datasets = (['source',])
def synthesis():
    for ds in datasets.source:
        print(ds.columns)
```

5.5 Datasets from Job Objects

Information about a job instance's datasets is provided using the `job.dataset()` function and the `job.datasets` attribute. To find all datasets in a job, use `job.datasets` like in this example

```
def main(urd):
    job = urd.build('create_datasets')
    for ds in job.datasets:
        print(ds.name)
```

To work on a specific dataset, just ask for it using its name as input parameter to `job.dataset()`,

```
    ds_first = job.dataset('first')
    ds_default = job.dataset()
```

Without an input parameter, the default dataset is returned. An error is issued if the dataset does not exist.

5.6 Dataset Properties

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

5.6.1 Dataset Name

The name of a dataset is accessible using the `.name` attribute, like this

```
    print(ds_first.name)
```

The Accelerator is designed to handle various string encodings with ease, and in most situations the naming rules are very liberal. The dataset name, however, should preferably be *limited to ASCII characters*, since a directory with the name of the dataset will be stored on disk. The Accelerator cannot guarantee that the file system in use handles any "special" characters. Newline is for example not allowed.

5.6.2 Column Names

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

```
datasets = ('source',)

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

The `.columns` attribute is actually a dictionary from column name to properties, as will be shown in the next section.
Not all column names are valid, see section 5.9.7 for more information.

5.6.3 Column Properties

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

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

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

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

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

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

5.6.4 Rows per Slice

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

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

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

5.6.5 Dataset Shape

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

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

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

5.6.6 Hashlabel

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

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

5.6.7 Filename and Caption

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

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

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

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

5.7 Operations on Chains

The .chain function is used to operate on dataset chains. It takes a dataset as input, some options that will be discussed next, and returns a DatasetChain object.

```
# return a chain object for dataset "ds"
chain = ds.chain()
```

The chain()-function takes the following optional arguments

name	default	description
length	-1	Number of datasets to include, default is -1, meaning all datasets in chain. Do not mix with stop_ds.
reverse	False	Return the datasets in reverse order. Default False.
stop_ds	None	Return datasets *from* stop_ds to current dataset. Do not mix with length.

The returned value, chain in the previous example, is of type DatasetChain, which supports the following operations.

name	description
min(<column>)	Minimum value of column, or None if column does not exist or is not sortable.
max(<column>)	Return minimum value of column, or None if column does not exist or is not sortable.
lines(sliceno=None)	Default is number of lines in chain. With option sliceno=x, number of lines in slice x.
column_count(<column>)	Number of datasets in chain containing column <column>
column_counts()	Counter of occurances per column per dataset in chain.
with_column(<column>)	A DatasetChain object containing only those datasets that has column <column>.
iterate(...)	Same arguments as Dataset.iterate(). Will iterate over the whole chain.

5.8 Column Data Types

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 supported types, including json and pickle, that may return rather complex datatypes.

By default, a typed column does not allow the storage of None values. This can be changed by setting the none_support Boolean when creating the column, see section B.6.1.

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

name	description
number	float or int

`float64`	64 bit (double) float
`float32`	32 bit float
`int64`	64 bit signed integer
`int32`	32 bit integer
`bits64`	64 bit bitmask
`bits32`	32 bit bitmask
`complex64`	64 bit complex number
`complex32`	32 bit complex number
`bool`	True or False
`date`	date
`time`	time
`datetime`	complete date and time object
`bytes`	raw data
`ascii`	ascii is faster in python2, otherwise use unicode
`unicode`	use for strings
`json`	a datastructure that is jsonable
`pickle`	a datastructure that is pickle-able!
`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:complex64`	int, float or string parsing into `complex64`
`parsed:complex32`	int, float or string parsing into `complex32`
`parsed:int64`	int, float or string parsing into `int64`
`parsed:int32`	int, float or string parsing into `int32`
`parsed:json`	string containing parseable json

5.8.1 Arbitrary precision numbers: `number`

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

5.8.2 Standard Fixed Size Numbers

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

5.8.3 Booleans

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

5.8.4 Types Relating to Time

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

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

5.8.5 String Types

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

5.8.6 Raw Data

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

5.8.7 Bitmasks

The bitmask types, `bits32` and `bits64`, are stored as 32 or 64 bits of data in a dataset, and is represented by unsigned integers in the Python code.

5.8.8 Complex Numbers

There are two complex number types, `complex32` and `complex64`.

5.8.9 JSON

The JSON type makes it possible to store and load more complex data structures in a dataset. Anything that is JSONable works as input. Conversion between JSON and Python data types is done by the writers and iterators, so the user can just work on the data and never has to see the actual JSON, for example

```
    dw = DatasetWriter(...)
    ...
    a = dict(x=3, y=dict(z=5, w=[1,2,3]))
    dw.write(a)
```

5.8.10 Pickle

The Python Pickle type is the most flexible type of all. It supports any pickle-able object, and encoding and decoding is done on-the-fly under the hood. This means that *it is possible to write almost anything into a dataset column*!

5.8.11 `parsed` Types

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

5.8.12 *`None`*-Handling

The value *`None`* is valid input for all types that support *`None`*, i.e. all types except the bitmask-types. For example, valid values for a `bool` type column are {*`True`*, *`False`* } without `none_support`, and {*`True`*, *`False`*, *`None`*} with `none_support`. See section B.6.1 for more information on `none_support`.

5.9 Creating a New Dataset

Datasets are created by methods using the `DatasetWriter` class. An instance of this class is available in a running method as `job.datasetwriter` like this

```
def prepare(job):
    dw = job.datasetwriter()
```

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

The most common arguments to `DatasetWriter` are

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

5.9.1 Create in `prepare()` + `analysis()`

The following example will use `DatasetWriter` to create a dataset with three columns of different types. The name of the dataset will be `firstset`. The writer will be initialised in `prepare()`, and data will be written to the dataset in `analysis()`. Note that the example actually creates a dataset *chain*, since it links the dataset under creation to the dataset named `previous` from the input parameters.

```
datasets = ('previous',)

def prepare(job):
    dw = job.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 function `dw.write()` is used to write data to the dataset. The order of the variables in the `.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

```
    # write a dict with keys corresponding to column names
    dw.write_dict({column: value})

    # write a list with items in same order as dw.add() calls
    dw.write_list([value, value, ...])

    # one parameter for each .add() call, in same order
    dw.write(value, value, ...)
```

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

5.9.2 Create in `synthesis()`

Since a dataset is sliced in multiple disjoint sets, and `synthesis()` is run only once, data has to be sliced during writing somehow. There are two possible ways to do this. One is to first set a slice number

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

before writing data into that slice. Note that this can only be done once per slice. The other is to use one of the `split_write` functions

```
    # use a dict-writer
    writer = dw.get_split_write_dict()
    writer({column: value})

    # use a list-writer
    writer = dw.get_split_write_list()
    writer([value, value, ...])

    # use a parameterised writer
    writer = dw.get_split_write()
    writer(value, value, ...)
```

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

5.9.3 Creating a Dataset Chain in one Job

It is possible to create a chain of datasets by inserting a writer object as previous to another writer, like this.

```
def prepare(job):
    dw1 = job.datasetwriter(name='ds1')
    dw2 = job.datasetwriter(name='ds2', previous=dw1)
```

5.9.4 Completing Dataset Creation

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

```
dw.finish()
```

The `finish()`-call returns a dataset object, so the just finished dataset could be put in use immediately, like this

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

5.9.5 Datasets Created by Subjobs

If a dataset is created in a subjob, it is not visible from the build scripts. This is solved by linking dataset meta-information to the job calling the subjob, using the `.link_to_here()` function. This is explained in detail in section 4.9, but the basic idea is as follows

```
from accelerator import subjobs
def synthesis():
    job = subjobs.build('dataset_creator')
    ds = job.dataset('ds1')
    ds.link_to_here('first_dataset')
```

Here, the subjob creates a dataset named `ds1`. This dataset resides in the subjob's job directory, but a link to it from the *current* job is created using the `link_to_here` function, which also allows the dataset to be renamed. The current job will now claim that it contains a dataset `first_dataset`.

In the same call, the subjob's dataset's previous can also be overridden:

```
    ds.link_to_here('first_dataset', override_previous=anotherds)
```

This might sound complicated, but is simply means that the current job can export a set of datasets created by its subjobs, and that it can manipulate these datasets so that they may appear to be chained in any way the user finds fit.

5.9.6 Creating Hash Partitioned Datasets

A hash partitioned dataset is created by setting the `hashlabel` argument to `job.datasetwriter`. For a hash partitioned dataset, only data fulfilling the hashing requirement for a slice may be written to that slice, and an exception will be rised if the data to be written does not belong to the current slice.

A simple way to filter the data to be written is to call

```
    dw.enable_hash_discard()
```

first in each slice or after each `.set_slice()`. Then, writes that belongs to another slice are silently ignored, while "correct" data gets written as expected.

It is possible to check before writing if the data is to be put into the current slice using the `dw.hashcheck()` function, like this

```
...
if dw.hashcheck(hashcoldata):
    # compute bulkdata here
    bulkdata = expensive_function(...)
    # and write to dataset
    dw.write(hashcoldata, bulkdata)
```

This is beneficial if it is expensive to compute the data to be stored. In the example above using `hashcheck()`, data is only computed if it is to be stored in the slice.

In general, the hash function for a particular column is available like this

```
dw.writers[colname].hash
```

This function can be used to manually check if the data belongs to a slice. For more details and alternatives, please see the documentation in the source file `dataset.py`.

5.9.7 Column Name Restrictions

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

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

5.9.8 More Advanced Dataset Creation

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

5.10 Appending New Columns to an Existing Dataset

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

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

5.10.1 Appending New Columns in Analysis

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

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

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

def analysis(sliceno, prepare_res):
    dw = prepare_res
    for data in datasets.source.iterate(sliceno, ... )
        ...
        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. Typically new columns are derived from existing ones, so this is usually not a problem.

5.10.2 Appending New Columns in Synthesis

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

```
def synthesis(job):

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

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

Note that the `for`-loop over all slices is controlling *both* the reading iterator *and* the dataset writer. Note also that `.set_slice()` can only be called once per slice.

Chapter 6

Iterators

The basic idea of the Accelerator's datasets is to make it easy to create parallel programs that can read and write large amounts of data at a very high speed. High speed data read access is implemented as a set of special Python *iterators*. Each iterator yields one `tuple` at a time containing elements from one or more specified data columns, one row at a time. In case of iterating over a single column, the output may optionally be a scalar instead of `tuple` for cleaner code and more efficient computing.

6.1 The Three Iterators

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

`iterate()`, for single dataset iteration,

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

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

And each of them will be discussed later in this chapter. For completeness, it should also be mentioned that the `DatasetChain` class also has an `.iterate()` function, and it works similar to `iterate_chain()`.

In many common use cases it is sufficient to provide only two arguments to the iterator: `sliceno`, which is mandatory, and `columns`. These, and all other arguments are presented in detail shortly. A typical use of an iterator looks like this

```
datasets = ('source',)

def analysis(sliceno):
    for m, u in dataset.source.iterate(sliceno, ('movie', 'user',)):
        # do something with m and u here...
```

Python's constructors can be used to create objects from iterators like in the following example, where the purpose is to compose a `dict`.

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

or this example

```
from collections import Counter
...
    c = Counter(dataset.source.iterate(sliceno, 'user')))
```

that will create a counter of how many times each `user` is present in the dataset.

6.1.1 Iterator Arguments

All three iterators share these arguments

name	default	description
`sliceno`	*mandatory*	Slice number (an integer) to iterate over, *None* to iterate over all slices sequentially, or `roundrobin` to take one value per slice in a round robin fashion.
`columns`	*None*	Tuple of column labels or a single name if iterating over one column. *None* selects all columns in alphabetical order.
`hashlabel`	*None*	Name of hash column. If the code relies on a dataset being hashed on a particular column, set this to make the iterator `assert` that this is actually the case. Execution will terminate if the hashlabel is incorrect.

`rehash`	*False*	Setting this to *True* will hash partition the dataset on-the-fly based on the `hashlabel` column. (Rehashing on-the-fly is slower, so ideally datasets should be rehashed using the `dataset_rehash` method 8.5.)
`status_reporting`	*True*	Give status when pressing `C-t`. Unless manually `zip`ing several iterators together, this should be set to default *True*. See `dataset.py` source code for full information.

In addition, `iterate_chain` takes these arguments too

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

and `iterate_list()` takes a `datasets` parameter

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

6.2 Basic Iteration

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

6.2.1 Parallel Iterator Invocation

For parallel iteration in `analysis()`, the iterator needs to know the number of the current slice. This information can be fed to the `analysis()` function in the `sliceno` variable. The following is an example of iteration that happens independently in each slice.

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

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

The program creates dictionaries mapping `users` to sets of `items` for the `source` dataset. Assuming that the dataset is hash partitioned (see 5.2), this operation is entirely parallel and there is no need to merge all the results from the analysis processes afterwards, since the different slices do not share any keys with each other.

6.2.2 Sequential Iterator Invocation

Setting the `sliceno` parameter to `None` will cause the iterator to run through all slices of the dataset, one slice 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)
```

Dataset slices will be iterated in increasing order.

6.2.3 Special Case, Round Robin Iteration

By default, the iterators stream slices of data. This is almost always exactly what is needed. But sometimes, for example when the order of rows imported by `csvimport` matters, there is a need for a row-wise iteration order. For maximum performance, the `csvimport` method writes datasets in a round robin fashion, so iterating over a csvimported dataset does not return the lines in the same order as they were written.

By setting the first parameter of any of the iterator functions to "`roundrobin`", the iterator will internally fetch all slice iterators and return one value at a time from each iterator in a round robin fashion. The resulting output is then in the same order as in the file imported by `csvimport`. In a dataset chain, round robin will happen *per dataset*. There is a performance penalty associated with this functionality, but it is handy for time-series-like data.

6.2.4 Special Cases, Iterating Over All or a Singe Column

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

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

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

If iterating over a single column, it makes little sense to keep the output values in a one-dimensional tuple. A scalar is cleaner and more efficient. 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)      # user is a scalar!
```

6.2.5 Iterate Over Chains

The `iterate_chain()` iterator is used to iterate over one or more datasets in a chain, starting at the "oldest" dataset. The following example will iterate over the last three datasets in the chain, oldest dataset first.

```
datasets = ('source',)

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

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

6.3 Halting Iteration

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

6.3.1 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.

6.3.2 Halting Using `stop_ds`

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

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

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

6.3.3 Halting Using Another Job's Input Parameters

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

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

6.4 Iterating Over a Data Range

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

```
for user, item in datasets.source.iterate_chain(
          sliceno, ('user', 'item',),
          range={timestamp, datetime(2016, 1, 1), datetime(2016, 3, 31),}):
```

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

```
for user, item in datasets.source.iterate_chain(
          sliceno, ('user', 'item',),
          sloppy_range={timestamp,
                        datetime(2016, 1, 1),
                        datetime(2016, 3, 31),}):
```

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

6.5 Iterating in the Reverse Direction

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

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

6.6 Hash Partitioned Datasets and on-the-fly Partitioning

Hash partitioning a dataset on a particular column, see section 5.2, may really simplify the parallel programming of methods using the dataset. However, the parallel code will not work properly if it turns out that the input data is in fact not hash partitioned in the expected way. For that reason, it is a good idea to *assert* the hashlabel by entering it into the iterator function, like this

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

so that execution will terminate if the `hashlabel` is not correct.

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

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

While this works, the preferred way to rehash is to use the `dataset_rehash` method 8.5, since it will store the rehashed dataset for later use, which in most scenarios will be more efficient.

6.7 Callbacks

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

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

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

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

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

The argument to the callback is the dataset instance corresponding to the dataset to be iterated next.

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

```
# callback once per slice
def prefun(dataset, sliceno):
    print(dataset.name, sliceno)

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

The `post_callback` function is defined similarly.

6.7.1 Skipping Datasets and Slices from Callbacks

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

- To skip the next dataset do

  ```
  raise SkipJob
  ```

- To have the iterator skip a slice, do a

  ```
  raise SkipSlice
  ```

- And to abort iterating completely

  ```
  raise StopIteration
  ```

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

Chapter 7

High Level Control: Urd

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

7.1 Introduction to Urd

Urd is a transaction log-based database and server used to group, tag, and keep track of jobs. Build scripts are capable of a large variety of complicated tasks, and on top of that Urd adds the capabilities of job organisation, storage, and retrieval. This makes it possible to handle much larger and more advanced projects while the process is fully transparent and reproducible.

Using Urd, a project can be separated into functionally independent parts, or *sessions*, each containing at least one job, and all dependencies between jobs inside as well as between these sessions are tracked by a *transaction log*. Sufficient information is stored so that any part or session of a project could theoretically be re-constructed the way it looked at any instance in time. More formally, Urd provides two things:

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

2. A searchable transaction log database of all sessions, together with their dependencies on other session. A timestamp, date, integer, or combination thereof can be used as key.

Due to its transaction log, there is no way using the provided API to accidentally modify or destroy data stored in the Urd database. The interesting transaction log database and many other aspects of Urd will be explained in this chapter.

7.2 A Simple Use Case

A simple use case is presented here to illustrate the basic Urd features. Assume a project where movie recommendation data is to be analysed. Every hour, new data corresponding to the last hour's activity is added to the project in the shape of a log file. The project is based on two build scripts, one *import* script, and one *analysis* script.

7.2.1 The Import Build Script

The import scrip is used to look for new data files, and import and chain them as they become available. For each new file that is imported, the build script will tell the Urd server the *timestamp* of the file as well as a *list of all jobs*, i.e. the Urd session, that have been built based on that file. Typically, this includes `csvimport`, `dataset_type`, perhaps a `dataset_hash_partition`, and similar data pre-processing jobs.

7.2.2 The Analysis Build Script

The analysis build scrip is used for the data analysis work, and is perhaps run less regularly. This script needs to know which jobs that correspond for the most recently imported file, and this is a straightforward thing to ask Urd. When all processing is done, all the analysis jobs are also stored in Urd together with a corresponding timestamp so that they can be retrieved later.

7.2.3 Urd Provides Separation

In the example, Urd is used to forward information about executed jobs from the first build script (the import script) to the second (the analysis script). In this sense, Urd provides *isolation* by "message passing" between build scripts. The import scripts can be modified or even removed, but as long as no workdirs are cleared and the Urd database is intact, it is still possible to fetch references to the existing import jobs from Urd. Thus, it is always possible to go back and see what things looked like at any previous point in time.

7.2.4 Transparency

Urd also provides *transparency*, since it can tell which input files that were used by any analysis job. In this way, it will never happen that there is uncertainty about which data that was used for a particular analysis.

To investigate the result of an analysis, Urd can be asked to return the jobs corresponding to, say, the most recent analysis jobs. The query response from Urd will contain these jobs, *as well as* the results from any Urd queries that were carried out for the jobs to run. This information will contain the import jobs, since these were queried in order to perform the analysis.

7.3 Local or Shared Urd Server

By default, Urd is run as a *local* server, which means that it is not externally accessible. A non-local Urd server that can share information between several users is straightforward to set up, see section A.5.

7.4 Urd Sessions and Lists

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

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

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

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

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

```
import/txn
```

and its *timestamp*

```
2018-05-03
```

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

7.5 A First Urd Query

The list created in the previous section is stored in the Urd database and therefore ready to be used by other build scripts. For example, here is a build script that does some processing on the previously imported file

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

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

    import_timestamp = import_session.timestamp
    import_job       = import_session.joblist['csvimport']

    urd.build('process', source=import_job)
    urd.finish('process/test', import_timestamp)
```

The first thing that happens is that all processing is covered in a session named `process/text`. The timestamp can be set in *either* `.begin()` or `.finish()`.

The script is then retrieving the most recently created urd session stored in list `import/txn`. Two things are extracted from this data, the *timestamp* and the *joblist*. The timestamp will be used for this session as well, to indicate that processing is based on data with that particular timestamp. A reference to the `csvimport` job is then extracted from the joblist and fed to the `process` job as an input dataset parameter. (For information about the `joblist` class, see section 3.2.)

7.6 The Contents of a Stored Session

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

```
{
    "user": "processing",
    "automata": "test",
    "timestamp": "2018-05-03",
    "caption": "",
    "joblist": [
        [
            "process",
            "TEST-37"
        ]
    ],
    "deps": {
        "import/txn": {
            "timestamp": "2018-05-03",
            "caption": "",
            "joblist": [
                [
                    "csvimport",
                    "TEST-34"
                ]
            ],
        }
    },
}
```

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

The most important keys are

name	**description**
`timestamp`	Timestamp of session
`caption`	A caption
`user/automata`	Name of Urd list
`joblist`	An object of type `joblist`, containing all jobs built in the session. For more information, see 3.2.

`deps`	A dictionary of dependencies from `user/automata` to urd sessions: `{'user/automata': session}`.

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

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

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

and the following applies

name	description
`urdlist`	is the name of the Urd list, and the same `urdlist` must be specified in both `begin()` and `finish()`. The `urdlist` is specified as `<user>/<list>`, where the `<user/>` part is optional. The `user` string is also for authentication, and must correspond to the current `URD_AUTH` settings, see section A.5.
`timestamp`	is *mandatory*, but could be set in either `begin()`, `finish()`, or both. `finish()` will override `begin()`.
`caption`	is *optional*, and can be set in either `begin()` or `finish()`. `finish()` will override `begin()`.
`update`	If set to `True`, the last item in the list may be updated. This option will be discussed in section 7.11.

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

7.7.1 What if a Build Script is Run Again?

The Urd database is based on transaction log files. There is no way to modify or remove information in the database using the Urd API. The first time a build script is run, it will cause jobs to be built, and if a session is set up using `begin()` and `finish`, this session will be stored in the Urd database. The second time the same script is run, the Accelerator will look up already built jobs and immediately return job references instead of building anything. As long as there are no changes, the session will look identical to the one stored in the Urd database, and it is possible to execute `finish()` without errors. But if there are any discrepancies, such as a job being rebuilt, Urd will refuse to store the differing session and instead cause an exception to notify the user.

Normally, a build script can be written in such a way that re-running it will be consistent will the Urd database and everything is fine. A mismatch with Urd is then an indication of some error. But there are cases when re-writing Urd history is the desired option, and this will be discussed in section 7.11.

7.7.2 Timestamp Definition and Resolution

The “timestamp” used to access items may be stated as either a `date`, `datetime`, `"datetime"`, `int`, `(datetime, int)`, or `"datetime+int"`. Here, `"datetime"` is a string of format

```
'%Y-%m-%d %H:%M:%S.%f'
```

(See Python's `datetime` module for explanation.) A specific timestamp could be shorter than the above specification in order to cover wider time ranges. The following examples cover all possible cases.

```
'2016-10-25'                 # day resolution
'2016-10-25 15'              # hour resolution
'2016-10-25 15:25'           # minute resolution
'2016-10-25 15:25:00'        # second resolution
'2016-10-25 15:25:00.123456' # microsecond resolution
```

Example of a timestamp with an `int`

```
'2016-10-25+3'
```

Note that

– `int`s sorts *first*,

– `datetime`s without `int` sorts *before* `datetime`s with `int`s,

– shorter `datetime` strings sorts *before* longer `datetime` strings, and

– a timestamp must be > 0.

7.8 Retrieving an Urd Session

A specific session can be retrieved from the Urd database using its *key* and *timestamp*. There are two sets of functions assigned for this, one that will *record* the lookup to the *ongoing* session, and one that will not. The reason for recording lookups is to provide the possibility for a session to contain references to all other sessions used to create it, and thereby ensuring transparency.

This section is devoted to the function calls that are recorded, and these are `get()`, `first()`, and `latest()`. For any of these calls to work, they have to be issued from *within* a session, i.e. after a `begin()` call. Otherwise Urd would not be able to record session dependencies. Section 7.9 describes the non-recording versions, and more ways to find sessions and inspect Urd database contents is described in section 7.12.

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

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

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

The timestamp must match exactly for an item to be returned. If there is no matching item, the `get()`-call will return an *empty session*, i.e. something like this

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

The strict matching behaviour can be relaxed by prefixing the timestamp with one of

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

For example

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

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

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

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

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

will match the latest timestamp starting with "`2018-05`" or less, such as "`2018-04-01`" or "`2018-05-31T23:59:59.999999`".

7.8.2 Finding the Latest Session: `latest()`

The `latest()` call will, for a given *key*, return the session with most recent timestamp. For example

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

will return a complete session, assuming that the list is exists and is populated. Otherwise, an *empty session* will be returned.

7.8.3 Finding the first item: `first()`

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

7.9 Retrieving an Urd Session without Recording

As pointed out in the previous section, dependencies will be recorded when the `get()`, `latest()`, and `first` functions are used. If, for some reason, the point is to just have a look at the database to see for example if something is already there and in that case what that is, it can be done using the "*peek*" functions: `peek()`, `peek_first()`, and `peek_latest()`, like this:

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

There are some good use cases for this, but it is not recommended for general use, they may cause a loss of continuity and visibility.

7.10 Aborting an Urd Session: `abort()`

There can only be one ongoing Urd session. A new session cannot be started until the current one has finished. There are three ways to end an urd session

- execute the `finish()` call and have the session recorded. (Or rejected, if a session with the same key and timestamp already exists but with different contents.)
- end the build script "prematurely" before the `finish()`-call. No data will be stored in Urd.
- issue an `abort()` call.

The `abort()` function is used like this

```
urd.begin('test')
urd.abort()
# execution continues here, a new session can be initiated
urd.begin('newtest')
```

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

7.11 Truncating and Updating

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

7.11.1 Updating the Last Item

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

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

If update is `True`, the entry in the test list at '2016-10-25' will be updated, unless the new information is equivalent. The `update()` call will simply add a new line to the Urd log database, and if the timstamp is the same as the previous entry, the new entry will be selected. Note that this only works for the *last* entry in the database, and that the previous version is not erased, it still exists in the log file.

7.11.2 Truncating a List

Sometimes there is a need to wipe all history or at least start over from an earlier time instance. Since data can not be erased from a transaction log, re-winding to an earlier timestamp is implemented using inserting a "truncation" marker in the log. Any data more recent than what is indicated by the marker is wiped from memory, and sessions with timestamps more recent than the marker can be stored again. The `truncate()` function is used to insert these markers, and it is used like this

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

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

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

that will erase all items from memory and cause the list to start over again. Again, remember that Urd is using plain text transaction log that can only be appended to. It is always possible to recover any old result or processing state from these logs.

7.11.3 Truncation Consequences: Ghosts

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

7.12 More Search Functions

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

7.12.1 Listing all Urd Lists: `list()`

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

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

may show something like

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

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

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

```
urd.since('ab/test', '2016-10-05')
```

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

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

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

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

7.13 Building Jobs: `build()`

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

```
job = urd.build(
    method,
    options={},     datasets={},     jobs={},
    name='',        caption='',
    workdir=None,
    **kw
)
```

If `options`, `datasets`, and `jobs` are uniquely defined in the method, they could be entered just as plain keyword arguments. If there are ambiguities, the full `options=` etc. must be used. Here is an explanation of `build` parameters:

name	**description**
`method`	Name of method to build. Enter `test` here if the method filename is `a_test.py`. The Accelerator will look for this file in all method directories specified in the Accelerator's configuration file.
`kw`	Use to specify any unique options, jobs, and datasets without the more elaborate dictionaries described next.
`options={}`	A `dict` of options to the method. This overrides options defined in the method itself, but adding options not prototyped in the method is *not* allowed.
`jobs={}`	A `dict` of jobs to the method. It is possible to specify a list of jobs like this `jobs=dict{alljobs=[job1, job2,...]}`
`datasets={}`	A `dict` of datasets to the method. Datasets may be lists too, just like `jobs` above.
`workdir=None`	If specified, the job will be built in this workdir, assuming the workdir is specified in the configuration file as *either* source or target.

`name`	A string associated with the job. Use it to distinguish several jobs created from the same method in joblists.
`caption`	A caption string. For decorative purposes only, this has no practical use.

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

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

If the source code is changed, a job rebuild can be prevented using the `equivalent_hashes` variable as explained in section 4.4.3.

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

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

```
def main(urd):
    # Import a list of files
    for filename, timestamp in listoffiles:
        urd.begin('import', timestamp)
        urd.latest('import')
        job = urd.build_chained('csvimport',
                                filename=filename,
                                ...
                                name='importing')
        job = urd.build_chained('dataset_type',
                                source=job,
                                ...
                                name='typing')
        urd.finish('import')
```

This example will build a chain of `csvimport` jobs, and one chain of `dataset_type` jobs. Each `dataset_type` job will have a corresponding `csvimport` dataset as source. The `build_chained()` function works, provided that

– the method to build has a dataset named `previous`,

– a unique `name=` is set in `.build_chained()`, and

– `urd.latest()` is called inside the Urd session.

The call to `urd.latest()` is necessary for the dependency-logic to work, but the output from the call can be discarded.

7.14 Changing workdir: `set_workdir()`

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

per job, using the `workdir=...` option to `urd.build` as shown in section 7.13, or

using `urd.set_workdir()`.

The latter,

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

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

7.15 Profiling a Build Script: `print_exectimes()`

The `JobList` object has a helper function that can be used to print profiling information for the joblist. The following example is self-explanatory

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

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

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

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

7.16 Passing Flags from the Command Line

Flags can be added to a build script at run time using a comma separated list like this

```
ax run [script] --flags=verbose,skiptest
```

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

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

7.17 The Urd HTTP-API

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

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

All calls return data in the JSON format. The Accelerator comes with a built in “`curl`” command that is designed for these calls, and it is used like this

```
% ax curl <endpoint>
```

But any standard tool such as the famous `curl` program works too, for example like this

```
% curl http://localhost:8123/list
```

7.17.1 The `list` endpoint

To show all stored lists issue

```
% ax curl list
["ab/test"]
```

All available lists are returned in a typed JSON list.

7.17.2 The `since` endpoint

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

```
% ax curl ab/test/since/2016-10-24
["2016-10-25"]
```

```
% ax curl ab/test/since/2016-10-26
[]
```

Results are in JSON list format.

7.17.3 The `first` and `latest` endpoints

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

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

And see the first stored job in the test list

```
% ax curl ab/test/first
```

works similarly. The returned data is an Urd item, described in section 7.6, in JSON format.

7.17.4 The `get` endpoint

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

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

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

7.18 Multi-Used Urd Consistency

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

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

Chapter 8

Standard Methods

The Accelerator is shipped with a set of common standard methods, including methods to import, type, export, and hash partition data. These methods are found in the method directory `./standard_methods`. All methods in `standard_methods` are designed and tested to work on both Python2 and Python3.

8.1 `csvimport` – Importing Data Files

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

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 "`input directory`" configuration parameter specified in the Accelerator's configuration file, see section A.4. A relative path makes it possible to relocate files to a different directory without trigging job remake.
`separator`	,	Field separator character. Accepts a single `iso-8859-1` character. Leave this empty to import each line of the input file into a single column.
`comment`	''	Lines beginning with this character are ignored. Accepts a single `iso-8859-1` character, or the empty string for no comments. Commented lines are stored in the `skipped` dataset.
`newline`	''	Newline character. Empty means `"\n"` or `"\r\n"`. Alternatively any single `iso-8859-1` character can be chosen.
`quotes`	''	Quote character. Empty or `False` means no quotes, `True` means both `'`, and `"`, any other character means itself.
`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',]`.
`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,}`.
`lineno_label`	''	If set, `lineno_label` becomes a column containing line numbers. Line numbers start at one (1), and corresponds to line numbers in the input file.
`discard`	`set()`	Labels in the discard set will not be stored in the dataset.
`allow_bad`	`False`	By default, this is set to `False` and an error will be asserted if there are problems parsing the input data, see section 8.1.3. Setting it to `True` will put all "bad" lines together with the corresponding line numbers into a separate dataset named `bad`. It is recommended to check the resulting datasets if enabling this option!

`skip_lines`	0	Skip this many lines at the start of the file. This is useful for data files that starts with a header, for example. Skipped lines will be stored in the `skipped` dataset.
`compression`	6	Compression level for the `gzip` compressor.
`allow_extra_empty`	*False*	Still consider a line good if it has extra empty fields at the end.
`skip_empty_lines`	*False*	Ignore empty lines.
`strip_labels`	*False*	Do .strip() on all labels (happens before rename).

8.1.2 Datasets

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

8.1.3 Bad Lines

A line is flagged as "bad" for one of two reasons

- there is a problem with quoting, or
- there is an incorrect number of separators.

for example

```
"a","b" "c"      # invalid assuming two or three comma separated columns
"a","b"" ""c"    # valid assuming two comma separated columns
```

8.1.4 Output

The result of the `csvimport` is a dataset named `default`. Lines marked as "bad" will be stored in the dataset `bad`, while skipped and commented lines will be stored in the dataset `skipped`. All columns will be of type `bytes`. Typically, the dataset from `csvimport` is fed to a `dataset_type` job for column typing.

8.1.5 Line Numbers

A column with line numbers is always attached to the `bad` and `skipped` datasets, and conditionally using `lineno_label` to the main dataset. Line numbers start at one (1), and always corresponds to the lines in the input dataset. For example, if there are labels on the first line of the input file, this line is number 1. Any line number can thus only appear in one of the main, `bad`, or `skipped` datasets.

8.1.6 Limitations

Each data value is limited to 16MB maximim. However, this is just a constant in the code that is by default set to a value that allows the Accelerator to run on low memory platforms. If you need to store, say, `bytes` values larger than 16MB, please update this constant to a larger value.

8.1.7 Example Invocation

An example invocation is the following

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

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

8.2 csvimport_zip – Importing zip Archives

The csvimport_zip method is a wrapper around csvimport that is used to import files stored in zip archives. One or more files in a zip archive can be imported by a call to this function, and each file will be imported to a separate dataset.

8.2.1 Options

All options to csvimport are available to this method as well, and the filename option is used to specify the name of the zip file.

name	default	description
inside_filenames	{}	Dictionary from filename in zipfile to dataset name, {'filename in zip': 'dataset name', ...}. If left empty, all files will be imported to datasets with cleaned up names. If there is only one file imported from the zip (whether specified explicitly or because the zip only contains one file) this will also end up as the default dataset.
chaining	on	Can be one of off, on, by_filename, or by_dsname. off – Don't chain the imports. on – Chain the imports in the order the files are in the zip file. by_filename – Chain in filename order. by_dsname – Chain in dataset name order. Since inside_filenames is a dict this is your only way of controlling its order.
include_re	”	Regexp of files to include, matches anywhere.
exclude_re	”	Regexp of files to exclude, takes priority over include_re.
strip_dirs	*False*	Strip directories from filename (a/b/c → c.)

If chaining is enabled, the last dataset will be the default dataset. Note that naming a non-last dataset "default" is an error. If strip_dirs is set, the filename (as used for both sorting and naming datasets, but not when matching regexes) will not include directories. The default is to include directories.

8.2.2 Example invocation

```
jid = urd.build("csvimport_zip",
    options=dict(
        filename="data_Q2_2019.zip",
        exclude_re=r"(__MACOSX|\.DS_Store)",
        chaining="by_filename",
        strip_dirs=True,
    ),
)
```

8.3 `dataset_type` – Typing Datasets

The `dataset_type` method will read a source dataset or dataset chain and type its columns. This method is primarily used for typing datasets created by `csvimport`, but it can type any column of type `bytes`, `ascii`, or `unicode` to any other type.

The method will also hash partition the output dataset if the `hashlabel` input paramter is set, causing a new dataset to be created. For additional information about hash partitioning, see the `dateset_hashpart` method in section 8.5.

The default behaviour is to append new columns with typed data to the existing source dataset. 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` option, typed columns can be assigned a name that differs from the original name, so that both typed and untyped data are available simultaneously. This brings transparency to the typing process. (However, even if the untyped data is "inaccessible" in the typed dataset, it is still available if referenced as the input dataset.)

In order to type the data, the input data is subject to parsing. Some datasets may contain data that is incorrect in the sense that it causes parsing errors when typing. Unparseable data can either be replaced by a default value or removed from the dataset. Since the Accelerator's dataset type does not permit removal of rows, i.e. datasets can not shrink, `dataset_type` will in this situation create a new dataset containing only the rows containing typeable data.

If typing a dataset chain, any columns that do not have the same type over all the typed datasets will be discarded.

8.3.1 Datasets

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

8.3.2 Options

name	**default**	**description**
`column2type`	`{}`	A dictionary from column label to type, for example `{'movie': 'unicode:UTF-8',}`.
`timezone`	`None`	Set input timezone for `datetime` columns. Output will be in UTC. See text.
`hashlabel`	`None`	Hash partition dataset based on this column. Leave as `None` to inherit hashlabel, set to " to not have a hashlabel. Hashing causes a new dataset to be created.
`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`		Optional caption. A reasonable caption is created automatically if left blank
`discard_untyped`	`None`	If set to `True`, force creation of new dataset and make untyped columns inaccessible. If set to `False`, an error is generated if any columns were not preservable.
`filter_bad`	`False`	If `False`, fail when a value fails to convert and there is no default. If `True`, filter out the line with the unconvertable value. This will create a new dataset.

numeric_comma	*False*	If *True*, write decimal number as "3,14" instead of default "3.14".
length	-1	Go back at most this many datasets. The default is -1, which goes until previous.source if it exists, or first dataset in chain otherwise.
as_chain	*False*	If hash partitioning, avoid re-writing at the end by doing one dataset per slice.
compression	6	gzip compression level.

The timezone option is used to specify which input timezone a datetime column is in. It does not work for date or time columns. It can be set to anything accepted by the system's $TZ. Note

- It does not work for %s (which is always in UTC).
- No error checking can be done on this (tzset(3) can not return failure).
- On most 32bit systems this will break dates outside about 1970 - 2037.
- Setting this will mask most bad dates (mktime(3) "fixes" them).
- Do not set this to 'UTC', leaving it as None is faster and safer.

8.3.3 Example Invocation

An example invocation is the following

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

8.3.4 Typing

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

Numbers

The number type is integer or floating point.

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

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

Floating Point Numbers

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

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

Integers

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

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

Integers Stored as Floats

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

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

Convert to Boolean

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

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

Time and Date

There are three types relating to time available, `date`, `time`, and `datetime`. Each of these has a corresponding version that ignores trailing garbage characters. All time types require a format specification (represented here by an asterisk) 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). `strip` will happen before `ignore`.

ASCII is similar, there are two types

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

where the argument is one of

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

Using `strict` will cause errors if unparsable. `strip` will happen before `ignore`.

8.4 csvexport – Exporting Text Files

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

Options

name	default	description
`filename`	*mandatory*	Name of output file. File will by default be stored in the job's job directory. The filename has to end with “`.csv`” for plain text files, and “`.gz`” for gzipped output.
`separator`	','	Column separator.
`labelsonfirstline`	`True`	If `True`, write column names on first row.
`chain_source`	`False`	If `True`, read a dataset chain from `datasets.source` back to `jobs.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.
`compression`	6	gzip level
`line_separator`	`'\n '`	Line separator.

Datasets

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

Jobs

name	default	description
`previous`	`None`	Job reference to previous `csvexport` if chained.

8.4.1 Example Invocation

An example invocation is the following

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

8.5 `dataset_hashpart` – Hash Partition a Dataset

The `dataset_hashpart` method will create a new dataset based on its `source` dataset. The new dataset will be hash partitioned 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 hash partition
`previous`	`None`	Previous dataset to chain to.

8.5.1 Example Invocation

An example invocation is the following

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

8.5.2 Hashing Details

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

```
from accelerator.gzutil import siphash24

target_sliceno = siphash24(cols[hashlabel]) % params.slices
```

8.5.3 Notes on Chains

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

2. Internally, `dataset_hashpart` 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.

8.6 `dataset_filter_columns` – Removing Columns from a Dataset

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

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

Options

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

Only one of `keep_columns` and `discard_columns` may be populated.

Datasets

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

8.7 `dataset_sort` – Sorting a Dataset

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

Options

name	default	description
`sort_columns`	*mandatory*	A column or a list of columns. If a list is specified, sorting will be carried out from left to right.
`sort_order`	`ascending`	Could be reversed by specifying `descending`
`sort_across_slices`	*`False`*	If *`False`*, only sort within slices. Otherwise sort across slices.
`trigger_column`	”	If `sort_across_slices` is set, use this to delay the slice switches to the next line where the value in this column changes.

Datasets

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

8.7.1 Sorting *`None`* and `NaN` values

The special values *`None`* and `NaN` follow these rules

- `NaN` will sort same as `+Inf`, i.e. *last*.
- *`None`* sorts as `-Inf`, i.e. *first* in `float` columns. Intermingled *`None`* and `-Inf` will keep their original order due to the stable sorting algorithm.
- *`None`* sorts *last* in `date`, `time`, and `datetime` columns.
- For all other types, *`None`* sorts *first*.

8.7.2 Spreading of Left Over Values

If the number of rows in a dataset is not even divisible by the number of slices, some slices will have one more row than others. Instead of putting this data in, say, the first slices, `dataset_sort` attempts to even out any bias by selecting the slices that get the additional data row in a pseudo-random manner. In order to have the sorting stable, selection of slices is based on the first values of the sorting column. It is not perfect, if the data is already sorted the first slices will be picked, but it is stable, which is the most important thing.

8.7.3 The Trigger Column

If sorting across slices, i.e. the full dataset will be sorted from the first to the last slice, the `trigger_column` could be used to delay the slice switching so that it does not appear unless the value in the column is changing. This is beneficial in parallel processing tasks, because it localises values to single slices.

8.7.4 A Practical Limitation

Internally, the method works by reading the columns to sort by, and create an indexing column that stipulates the sorting order. Each column is then read in turn and sorted according to the sorting column. Therefore, the method has limited sorting capability. Internally, it sorts one column at a time, and it needs to hold that complete column plus an indexing column in memory simultaneously. Still, a standard computer can sort very large datasets without trouble.

8.8 dataset_checksum, dataset_checksum_chain

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

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

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

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

Note also that `pickle`-columns cannot be used for checksumming, since the order of the pickled data stricture is not defined.

The checksum will be printed to `stdout` as well as written to `result.pickle`.

Options

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

Datasets

name	**default**	**description**
`source`	*mandatory*	A dataset to checksum.
`stop`	`None`	Stop hashing at this dataset. Only for `dataset_checksum_chain`.

8.9 `dataset_merge` – Merge Several Datasets into One

Merge two or more datasets. The datasets must have the same number of lines in each slice and if they do not have a common ancestor you must set `allow_unrelated=True`. Columns from later datasets override columns of the same name from earlier datasets. Note that merging is a very fast constant time operation.

Options

name	default	description
`allow_unrelated`	`False`	Must be `True` to join datasets that do not share a common ancestor.

Datasets

name	default	description
`[source,]`	*mandatory*	A list of datasets to merge.
`previous`	`None`	Previous for the merged dataset.

8.10 `dataset_unroundrobin` – Transpose a Dataset

This method will transpose the dataset so that first row from slices 0, 1, 2, ... will end up in slice 0 rows 0, 1, 2 .... It always creates a new dataset. If `trigger_column` is set to the name of a column, slice switching is delayed until the data in the specified column changes. Otherwise, the new dataset will have the same number of rows per slice as the source dataset.

The `trigger_column` will ensure that no value is spread over more than one slice, assuming that all occurances of the same value are grouped together, which is helpful in many parallel processing cases.

Options

name	**default**	**description**
`trigger_column`	*None*	Set to delay slice switching based on content.

Datasets

name	**default**	**description**
`source`	*mandatory*	An input dataset.
`previous`	*None*	Previous for the created dataset.

Chapter 9

Running the Accelerator

The Accelerator is controlled using the ax shell command. In order to run any command, ax needs to have access to a configuration file (see section A.4). The ax command will look for this file first in the current directory, and then recursively in directories above it.

It is assumed that the Accelerator server and build script run commands are executed from the same directory. This will work out of the box. But if set up correctly, they could be run from different directories or even from different computers if necessary.

Asking for help is always an option. To begin,

```
  ax --help
```

will print something like

```
usage: ax [--config CONFIG_FILE] command

positional arguments:
  command

optional arguments:
  --config CONFIG_FILE  Configuration file

commands:

    curl  http request (with curl) to urd or the server
  server  run the main server
    grep  search for a pattern in one or more datasets
      ds  display information about datasets
    init  create a project directory
     run  run a build script
     urd  run the urd server

Use ax <command> --help for <command> usage.
```

each command will be introduced next.

9.1 Initialisation

In order to start a new project, a few things need to be set up, in particular

identify existing / create new *workdirs*,

identify existing / create new *method packages*, and

write a *configuration file*.

This can be done manually, but a simple way to start from scratch is to use the init command

```
  ax init
```

with the following options

```
  ax init --help
```

```
usage: ax init [-h] [--slices SLICES] [--name NAME] [--input INPUT] [--force]
               [DIR]

Creates an accelerator project directory. Defaults to the current directory.
Creates accelerator.conf, a method dir, a workdir and result dir. Both the
method directory and workdir will be named <NAME>, "dev" by default.

positional arguments:
  DIR              project directory to create. default "."
```

```
optional arguments:
  -h, --help       show this help message and exit
  --slices SLICES  override slice count detection
  --name NAME      name of method dir and workdir, default "dev"
  --input INPUT    input directory
  --force          go ahead even though directory is not empty, or workdir
                   exists with incompatible slice count
```

9.2 Accelerator Server

The Accelerator server, or daemon, needs to be running in order to execute any commands.

9.2.1 Invocation

```
% ax server
```

will start the Accelerator server, assuming that a configuration file that makes sense is in place.

```
% ax server --help
```

```
usage: ax server [-h] [--debug]

optional arguments:
  -h, --help  show this help message and exit
  --debug
```

Communication with the Accelerator server will take place over an UNIX socket by default. There is no need for any additional configuration to make that happen. It is possible, however, to communicate over a TCP port instead if specified in the Accelerator's configuration file.

9.3 Running Build Scripts

9.3.1 Invocation

Build scripts are executed using

```
ax run <script>
```

```
ax run --help
```

```
usage: ax run [options] [script]

positional arguments:
  script                build script to run. default "build".
                        searches under all method directories in alphabetical
                        order if it does not contain a dot.
                        prefixes build_ to last element unless specified.
                        package name suffixes are ok.
                        so for example "test_methods.tests" expands to
                        "accelerator.test_methods.build_tests".

optional arguments:
  -h, --help            show this help message and exit
  -f FLAGS, --flags FLAGS
                        comma separated list of flags
```

```
-A, --abort           abort (fail) currently running job(s)
-q, --quick           skip method updates and checking workdirs for new jobs
-w, --just_wait       just wait for running job, don't run any build script
-F, --fullpath        print full path to jobdirs
--verbose VERBOSE     verbosity style {no, status, dots, log}
--quiet               same as --verbose=no
--horizon HORIZON     time horizon - dates after this are not visible in
                      urd.latest
```

When the `run` command starts, it will 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 server). Thereafter, it will execute the build script.

9.4 Dataset Information

The `ds` command gives a compact, but easy to read, overview of either

a dataset,

a chain of datasets, or

available datasets in a job directory.

it provides information about column names and types, max and min values, number of rows, and balance of rows between slices.

9.4.1 Invocation

```
usage: ax ds [options] ds [ds [...]]

positional arguments:
dataset

optional arguments:
-h, --help              show this help message and exit
-c, --chain             list all datasets in a chain
-C, --non_empty_chain   list all non-empty datasets in a chain
-l, --list              list all datasets in a job with number of rows
-L, --chainedlist       list all datasets in a job with number of chained rows
-m, --suppress_minmax   do not print min/max column values
-n, --suppress_columns  do not print columns
-q, --suppress_errors   silently ignores bad input datasets/jobids
-s, --slices            list relative number of lines per slice in sorted
order
-S, --chainedslices     same as -s but for full chain
```

The `dataset` option is either a *dataset*, when used with the `-s`, `-S`, and `-c` options, or a *jobid* when used with `-l` option. Datasets or jobids could be either names or absolute paths. Examples of valid datasets are `test-2`, `test-2/default`, and `/home/wdirs/test/test2/dsx`. Of these, only `test-2` is a valid jobid. Here are all options

`-h` `--help`	show help message and exit.
`-q` `--quiet`	Silently ignore any error.

When `ds` is fed with dataset(s)

`-c` `--chain`	Print name and number of lines for all datasets in the chain.
`-s` `--slices`	Print absolute and relative number of lines per slice for the input dataset.
`-S` `--chain`	Same as `-s`, but data is for the whole chain of datasets.

When `ds` is fed with jobid(s)

`-l` `--list`	Print the name and number of lines of all datasets in the input jobid.

Example invocation 1

```
ax ds test-20 -S
```

or

```
ax ds test-20/badlines
```

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.

Example invocation 2

Combining `ds` will shell features can be an elegant way to extract information from a workdir. For example

```
ax ds -l -q test-{0..99}
```

will scan for datasets in the 100 first jobs of `test`. Adding the `-q` option will make `ds` suppress the warning messages for those jobs that do not contain any datasets.

Example invocation 3

Find all datasets in `test-20`

```
ax ds -l test-20
```

Example Output

```
import-2340/default
    Previous: import-2245/default
    Hashlabel: serial_number
    Columns:
          capacity_bytes        int64   [        -1, 14000519643136]
          date                  date    [2019-04-01, 2019-06-30]
          failure               bool    [     False,       True]
          model                 ascii
        * serial_number         ascii
    5 columns
    9,831,138 lines
    Chain length 17, from import-269 to import-2340
    Balance, lines per slice, full chain:
          1:   4.44% (6,486,330)   20:   4.35% (6,365,896)    3:   4.32% (6,323,701)
          0:   4.43% (6,472,949)   17:   4.35% (6,363,206)   19:   4.31% (6,309,295)
         11:   4.39% (6,423,397)   21:   4.35% (6,360,759)   12:   4.31% (6,306,181)
          4:   4.39% (6,421,043)   15:   4.35% (6,358,757)    8:   4.31% (6,304,311)
          6:   4.39% (6,418,617)    7:   4.34% (6,352,750)   13:   4.31% (6,303,637)
         14:   4.39% (6,417,911)   18:   4.34% (6,348,128)    5:   4.28% (6,252,735)
          2:   4.36% (6,376,139)   10:   4.34% (6,347,694)   16:   4.26% (6,230,543)
         22:   4.35% (6,366,040)    9:   4.33% (6,325,241)
    Max to average ratio: 1.020
    146,235,260 total lines in chain
```

The `max to average ratio` shows the ratio between the slice with most rows and the average number of rows. This can be interpreted as the execution time overhead for a dataset where all slices are not of the same size.

9.5 Look at Data in a Dataset

The `grep` command is a parallel grep for datasets that is used to look at data stored in datasets or dataset chains.

9.5.1 Invocation

```
  usage: ax grep [options] pattern ds [ds [...]] [column [column [...]]

    positional arguments:
    pattern
    dataset
    columns

    optional arguments:
    -h, --help            show this help message and exit
    -c, --chain           follow dataset chains
    -i, --ignore-case     case insensitive pattern
    -s SLICE, --slice SLICE
    grep this slice only, can be specified multiple times
```

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

```
ax grep Alice test-0 test-1/special name
```

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

`-h` `--help`	show help message and exit
`-c` `--chain`	follow dataset chains
`-i` `--ignore-case`	Case insensitive pattern
`-s N` `--slice N`	Grep in slice `N` only

Strings and columns with special characters have to be quoted.

9.5.2 Abuse grep to Show Datasets

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

```
ax grep . test-0 | less
```

(The regexp `.` will match any string that is at least one character long.)

9.5.3 Information about Methods

Use `ax method` to print a list of available methods.

```
ax method
```

The method description is shown by specifying a method name

```
ax method csvimport
```

Descriptions are added using the `description="some string"` statement in a method.

9.6 The Urd Job Database Server

By default, a local Urd server is running when the Accelerator server is running. Read more about this in section 7.

Start Urd by

```
ax urd
```

These are the options

```
ax urd --help
```

```
usage: urd [-h] [--port PORT] [--path PATH]

optional arguments:
-h, --help   show this help message and exit
--port PORT  server port (default: 8080)
--path PATH  database directory (can be relative to project directory)
(default: ./urd.db)
```

Remember to set matching values in the Accelerator's configuration file so that it can find the Urd server.

9.6.1 Authorization to Urd

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

```
% URD_AUTH=user:passwd ax run [script]
```

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

Appendix A

Setup and Installation

This chapter covers how to install the Accelerator, how to configure it, and how to set up a new project.

A.1 Install the Accelerator

A.1.1 Using the pip command

The easiest way to install the Accelerator is by fetching it from the PyPi repository

```
pip install accelerator
```

Some prefer to install to a virtual environment and do something in line with the following

```
python3 -m venv accvenv
source accvenv/bin/activate
pip install accelerator
```

This will install the Accelerator to the `accvenv` virtual environment. Now, use for example the following command

```
ax --help
```

to check that the installation worked. The next step is to set up a project.

A.2 Set up a New Project

In order to run, the Accelerator needs to have these things in place

at least one *workdir* to store data in,

most likely a new *method package* directory to store new code in, and

a *configuration file* to set things up.

This is all taken care of by the `init` command.

A typical project setup will look like this

```
myproject/
    accelerator.conf
    dev/
        methods.conf
        a_method.py
        build.py
```

where methods are stored in the `dev` directory.

A.3 Run the Tests

A rather extensive test suite is included in the Accerator installation. To run this, enable the test package in the configuration file:

```
method packages:
    ...
    accelerator.test_methods
```

start the server using

```
ax server
```

and in another shell start the test

```
ax run tests
```

Since the tests include testing of different character encodings, you may end up with a

```
Exception: Failed to enable numeric_comma, please install at least one
of the following locales: da_DK nb_NO nn_NO sv_SE fi_FI en_ZA es_ES
es_MX fr_FR ru_RU de_DE nl_NL it_IT
```

On a Debian-based machine, locales can be configured using

```
dpkg-reconfigure locales
```

which has to be run with root privileges.

A.4 Server Configuration File

The configuration file specifies which method packages and workdirs that are available for a project. A template configuration file can be generated using the `init` command as described in section 9.1. Below is an example of a configuration file.

```
# The configuration is a collection of key value pairs.
#
# Values are specified as
# key: value
# or for several values
# key:
#       value 1
#       value 2
#       ...
# (any leading whitespace is ok)
#
# Use ${VAR} or ${VAR=DEFAULT} to use environment variables.

slices: 23
workdirs:
        test /zbd/workdirs/test
        import ${HOME}/workdirs/import
        live wdirs/live

# Target workdir defaults to the first workdir, but you can override it.
# target workdir: dev
# (this is where jobs without a workdir override are built)

method packages:
        dev
        accelerator.standard_methods
#       accelerator.test_methods

urd: http://localhost:9000

result directory: ${HOME}/accelerator/results
input directory: /zbd/data/backblaze

# If you want to run methods on different python interpreters you can
# specify names for other interpreters here, and put that name after
# the method in methods.conf.
# You automatically get four names for the interpreter that started
# the server: DEFAULT, 3, 3.7 and 3.7.3 (adjusted to the actual
# version used). You can override these here, except DEFAULT.
# interpreters:
#       2.7 /path/to/python2.7
#       test /path/to/beta/python
```

The configuration file above specifies 23 slices and three workdirs, called `test`, `import`, and `live`. The `test` workdir is specified using an absolute path, the `import` workdir is specified relative to the user's home directory using the shell environment variable `$HOME`, and the `live` workdir is specified using a path relative to the location of the configuration file itself.

The workdir that is specified first is the *target workdir*, where jobs are written to by default. All other specified workdirs will by default only be used for reading. Any of the workdirs specified could be written to, though, using the `set_workdir=` option to the `build` command, as described on page 7.14.

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

name	**description**
`slices`	Number of slices used for the project.
`workdirs`	A list of paths to workdir directories. At least one workdir needs to be defined. All workdirs that are used together must have the same number of slices. It is possible to use shell environment variables such as `${HOME}` when specifying workdirs. Path starting with a slash (`/`) are absolute paths, all other paths are relative to the location of the configuration file itself. Unless overridden by the `target workdir`, the first workdir in the list will be the default *target workdir* that is used for all writing. Other specified workdirs will only be read from, unless overrided by the `build` call as described on page 7.14.
`target workdir`	Name of the *target workdir*. If specified this overrides the first item in the `workdirs` list.
`method packages`	A list of directories containing methods. These will be the only directories where the Accelerator can “see” methods. `standard_methods` is bundled with the Accelerator and is commonly used.
`urd`	If present, an URL to the Urd server.
`result directory`	A common path that is available to all jobs. Use the `job.link_result()`-function to create symbolic links from files in job directories to this directory. Just like workdirs, this path is either absolute or relative to the location of the configuration file.
`input directory`	Default root path for `csvimport`. This is to avoid rebuilds of imports if input files are moved to another directory. (This typically happens when setting up a similar system on another physical machine.) See section A.9.1 on how to get access to `input_directory` from any method. Just like workdirs, this path is either absolute or relative to the location of the configuration file.
`interpreters`	Name and path to python executables. These are used in `methods.conf` to specify specific Python versions (or virtual environments) for individual methods. If unspecified, methods will be executed using the same binary that runs the Accelerator’s server process.

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.5 Setting up a Standalone Urd-server

The main server program will start a local Urd server by default. This server is for local use only. If the Urd server should be used to share information between users, a standalone server needs to be set up.

To run a standalone Urd server, two things are needed

a directory where it can put its database, and

a `passwd` file to store user-password pairs in.

The `passwd` file is stored in the urd database directory. The default name of the database directory is `urd.db`, so

```
mkdir urd.db
cd urd.db
<editor> passwd
```

where `<editor>` should be replaced by the editor of choice.

A.5.1 Starting Urd

Urd is running as a daemon. It is started like this

```
ax urd --port=<port> --path=<path>
```

A.5.2 The Urd Database

The Urd database has the following structure

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

A.5.3 The `passwd` file

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

```
user:password
```

For example, if the file contains the following line

```
ab:secret
```

A build script run like this

```
URD_AUTH=ab:secret ax run script
```

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

A.6 Workdirs

Jobs are stored in *workdirs*. Workdirs are defined in the Accelerator's configuration file, where at least one workdir must be specified.

By default, the only workdir that is written to is the target workdir, while all other defined workdirs are for reading. It is possible to override this, however, by setting the `workdir=` option in the `urd.build()` call, see section 7.13.

Jobdirs are stored in the workdir by the server, 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/
    .slices.conf
    test-0/
    test-1/
    test-2/
    test-LATEST -> test-2
```

The `.slices.conf` file contains the number of slices used for the workdir. The link `<workdir>-LATEST` is always pointing to the last jobdir created. This is useful for example when iteratively testing a method and accessing its data for example for plotting purposes. Each new build of the (modified) method will create a new job, and the link will always point to the most recent version.

A.6.1 Creating a Workdir

If a workdir defined in the configuration file does not exist on disk at the stated location, the server will exit and print an error stating that a directory is missing. The first time the server encounters a new directory it will initialise it in accordance with the configuration file. So, new workdirs are created by adding them to the configuration file *and* creating the corresponding directories. The Accelerator will then initiate these directories on the next startup.

The initiation process creates a file named `.slices.conf` that indicates that the directory is now a workdir. This file contains the number of slices that is used for the workdir.

A.7 Status (Progress) Reporting

During job building, it is possible to press `C-t`, i.e. `Ctrl` + `t` simultaneously, in the `run` shell to get status information. The built in status will report the processing state, if it is in `prepare()`, `analysis()`, or `synthesis()`. Iterators report which dataset (perhaps in a chain) that is currently being iterated, and the `blob` functions report status of file pickling.

The `status` module makes it possible to insert status reporting into any method. For more information about status reporting, see section A.8.

A.8 Generating Progress Messages: the `status` Module

The status module is used by the Accelerator to report processing state. It is also used by various functions to report iterator and file access progress. Status messages are presented in the `run` shell by pressing `C-t`, i.e. the `Ctrl` and `t` keys simultaneously.

The status module can be used to write progress and status messages for any function. Here is an example of how to use the status module

```python
from accelerator import status
...
def analysis(sliceno):
    msg = "reached line %d already!"
    with status(msg % (0,) as update:
        for ix, data in enumerate(datasets.source.iterate(sliceno, 'data')):
            if ix % 1000000 == 0:
```

```
        update(msg % (ix,))
```

In the example above, the status message will be updated once every million iteration. By pressing `C-t` during its execution, the user will get a message telling how many lines the iterator has reached.

A.9 Working with Relative Paths

In some situations, like importing data from files, it is convenient to store the absolute path of the files as a configuration parameter and then work only with relative paths in the source code. This has two advantages.

> First, it makes it possible to move input files around without forcing a re-build of the import jobs, and
>
> second, absolute paths will not be stored in the source code.

In order to make use of relative paths, store the "system dependent" left part of the path in the Accelerator's configuration file. There are two variables in the configuration file that can be used for this, and they have different purposes. The `input_directory` variable is intended for reading input files, and the `result_directory` is intended for writing output. See the following subsections for details.

A.9.1 The `input_directory`

Files stored in the `input_directory` can be accessed using the `CurrentJob` object either like this

```
def synthesis(job)
    fname = job.input_filename('afile.csv')
```

or like this

```
def synthesis(job)
    with job.open_input('afile.csv', 'rt') as fh:
        for line in fh:
            ...
```

Since the `input_directory` is defined in the Accelerator's configuration file, the input directory can be moved and re-defined without having to re-execute any built jobs.

A.9.2 The `result_directory`

Storing files in job directories is great for transparency, but in some cases it is convenient to keep a reference to result files in a common place. This is the purpose of the `result_directory`. However, storing files in this directory directly would void the connection to the job that created it. A better way is to keep the file in the job directory and create a symbolic (soft) link to it in the result directory. This functionality is implemented in the `job.link_result()` function that is available to *build scripts*, and used like this

```
def main(urd):
    job = urd.build('somejob')
    job.link_result()   # links the job's "result.pickle" to result_directory
    job.link_result('result.txt')
    job.link_result('result.txt', linkname='result_from_somejob.txt')
```

Note that this only works in build scripts. The `job.link_result(filename)` call will create a symbolic link named `filename` in the `result_directory` defined in the Accelerator's configuration file. The link will point to the original file in the job directory, and will be silently replaced if it already exists.

Appendix B

Classes

The Accelerator is programed using an object oriented approach. This chapter outlines the most common classes and its member functions.

B.1 The `Job` and `CurrentJob` Classes

The `Job` and `CurrentJob` classes are similar, but used in different contexts:

> The `Job` class is used to represent and operate on *existing* jobs. An object of this class is returned from job `build()` calls as well as when retreiving jobs from Urd or a `JobList` object.
>
> The `CurrentJob` class is an extension that provides mechanisms for operations performed while a job is *executing*, such as saving files to the job's jobdir.

The classes are derived from the `str` class, and objects of these classes decay to (unicode) strings when pickled. The following attributes are available on both the `Job` and `CurrentJob` classes:

name	**description**
`chain()`	Job chain
`dataset()`	Return a named dataset from the job.
`datasets`	List of datasets in job.
`filename()`	Return absolute path to a file in a job.
`files()`	Return list of files created by job.
`json_load()`	Load a json file from the job's directory.
`link_result()`	Create a soft link from a file in a job's directory to `result_directory`.
`load()`	Load a pickle file from the job's directory.
`method`	The job's method. This can be overriden by `name=` if job instance is output from Urd or a `build()` call.
`number`	The job number as an `int`.
`open()`	Similar to standard `open`, use to open files.
`output()`	Return what the job printed to `stdout` and `stderr`.
`params`	Return a dict corresponding to the file `setup.json` for this job.
`path`	The filesystem directory where the job is stored.
`post`	Return a dict corresponding to the job's `post.json`.
`withfile()`	A `JobWithFile` with this job.
`workdir`	The workdir name (the part before `-number` in the jobid).

In addition, the `CurrentJob` class has these unique attributes:

name	**description**
`datasetwriter()`	Returns a DatasetWriter object. See documentation for `Dataset.DatasetWriter()`, section B.6.
`input_filename()`	Full filename of file in `input_directory`.
`json_save()`	Store a json file in the current job directory.
`open()`	Added extra `temp` argument.
`open_input()`	Open a file in `input_directory`.
`register_file()`	Record a file produced by this job.
`save()`	Store a pickle file in the current job directory.

Detailed description of the functions, where neccessary, follows.

B.1.1 `Job.chain()`

This works like `dataset.chain()`, but for chains of `jobs`.

name	default	description
`length`	`-1`	Length of chain.
`reverse`	*`False`*	Reverse direction of chain.
`stop_job`	*`None`*	Chain to here.

B.1.2 `Job.dataset()`

name	default	description
`name`	`default`	

Get a dataset instance from a job.

B.1.3 `Job.files()`

name	default	description
`pattern`	”	Return only files matching `pattern`

This method returns a list of all filenames corresponding to files created by the job using the functions `CurrentJob.open()`, `CurrentJob.save()`, or `CurrentJob.json_save()`. The list can be filtered using the `pattern` option. Filtering is based on Python's `fnmatch`-functionality.

B.1.4 `Job.filename()`

name	default	description
`filename`	*Mandatory*	Name of file in job directory.
`sliceno`	*`None`*	Set to current slice number if sliced, otherwise *`None`*.

Return the absolute (full path) filename to a file stored in the job. If the file is sliced, a particular slice file can be retrieved using the `sliceno` parameter. Sliced files are described in section 4.6.2.

B.1.5 `Job.input_filename()`

Return the full filename of a file stored in the `input_directory`.

name	default	description
`filename`	*Mandatory*	Name of file.

B.1.6 `Job.open_input()`

Open a file in the `input_directory`.

name	default	description
`filename`	*Mandatory*	Name of file.
`mode`	`r`	Open file in this mode, see Python's `open()`
`encoding`	*`None`*	Same as Python's `open()`
`errors`	*`None`*	Same as Python's `open()`

B.1.7 `Job.json_load()`

name	default	description
`filename`	`result.json`	Name of file.
`sliceno`	*`None`*	

Load a file from a job, in JSON format.

B.1.8 `Job.load()`

name	default	description
`filename`	`result.pickle`	Name of file.
`sliceno`	*`None`*	
`encoding`	`bytes`	

Load a file from a job in Python's pickle format.

B.1.9 `Job.open()`

name	default	description
`filename`	*Mandatory*	Name of file.
`mode`	`r`	Open file in this mode, see Python's `open()`
`sliceno`	*`None`*	Read or write sliced files.
`encoding`	*`None`*	Same as Python's `open()`
`errors`	*`None`*	Same as Python's `open()`
`temp`	*`None`*	Control file persistence. See text.

This is a wrapper around the standard `open` function with some extra features. Note that

– `Job.open()` can only read files, not write them, and therefore "`r`" flag must be set.

– `CurrentJob.open()` can both read and write.

– `CurrentJob.open()` must be used as a context manager, like this

```
with job.open(...) as fh:
    ....
```

– `CurrentJob.open()` can use the `temp` flag to modify the persistence of written files.

– `CurrentJob.open()` will register the file to the current `job`.

The `temp` argument is used to control the persistence of files written using `.open()`. This is useful mainly for debug purposes, and explained in section B.1.17. Sliced files are described in section B.1.16.

B.1.10 `Job.output()`

name	default	description
`what`	*`None`*	Which functions to return output from.

Get everything a job has printed to `stdout` and `stderr` in a string variable. The parameter `what` can be set to

`None`, which returns everything,

`prepare`, which returns everything from `prepare`,

`analysis`, which returns everything from `analysis`,

`synthesis`, which returns everything from `synthesis`, or

a number, which returns output from the corresponding `analysis` slice.

B.1.11 `Job.register_file()`

Register a file to the running `job`. This happens automatically when using `Job.open()`, but if the file was produced in a way where that is not practical this can be used to manually register the file.

name	default	description
`filename`	*Mandatory*	Name of file.

B.1.12 `Job.withfile()`

name	default	description
`filename`	*Mandatory*	Name of file.
`sliced`	*`False`*	Boolean indicating if the file is sliced or not.
`extra`	*`None`*	Any additional information to the job to be built.

The `.withfile()` is used to highlight a specific file in a job and feed it to another job `build()`. The file could be sliced.

B.1.13 `Currentjob.link_result()`

name	default	description
`filename`	*Mandatory*	Name of file in job directory.
`linkname`	*`None`*	Name of link if set.

Use to create a soft link from a file in a job directory to the `result_directory`.

NOTE: This only works for `Job` instances, and not `CurrentJob` instances. This is for reproducibility reasons. Links in `result_directory` cannot be recreated if created in a job, since jobs can only be executed once.

B.1.14 `CurrentJob.json_save()`

name	default	description
obj	*Mandatory*	
filename	result.json	
sliceno	*None*	
sort_keys	*True*	
temp	*None*	

For `CurrentJob` instances only. Save data into the current job's directory in JSON format. The `temp` argument is used to control the persistence of files written using `.json_save()`. This is useful mainly for debug purposes, and explained in section B.1.17.

B.1.15 `CurrentJob.save()`

name	default	description
obj	*Mandatory*	
filename	result.pickle	
sliceno	*None*	
temp	*None*	

For `CurrentJob` instances only. Save data into the current job's directory in Python's pickle format. The `temp` argument is used to control the persistence of files written using `.save()`. This is useful mainly for debug purposes, and explained in section B.1.17.

B.1.16 Sliced Files

A *sliced* file is actually a set of files used to store data independently in each `analysis()` process using a common name. The functions that operate on files, such as for example `.open()` and `.load()`, can switch to sliced files using the `sliceno` parameter. From a user's perspective, they always appear to work on single files. For example

```
def analysis(sliceno, job):
    data = ...
    job.date(data, "mydata", sliceno=sliceno, temp=False)
```

will create a set of files `mydata.%d`, where `%d` is replaced by the slice number. In this way, data can be passed "in parallel" between different jobs.

B.1.17 File Persistence

The `temp` argument controls persistence of files stored using `.open()`, `.save()`, or `.json_save()`. By default it is being set to *False*, which implies that the stored file is *not* temporary. But setting it to *True*, like in the following

```
job.save(data, filename, temp=True)
```

will cause the stored file to be deleted upon job completion. The functionality can be combined with the *debug* mode, see below.

temp	"normal" mode	debug mode
False	stored	stored
True	stored and removed	stored

Debug mode is active if the Accelerator server is started with the `--debug` flag.

B.2 The JobWithFile Class

The `JobWithFile` class is used to create a job input parameter from a file stored in a job.

name	**description**
`filename()`	Return filename.
`load()`	Load file contents. Takes an `encoding=`-parameter.
`json_load()`	Load JSON file contents.
`open()`	Returns a filehandle to the open file. Takes `encoding=`- and `errors=`-parameters.

All above functions take the argument `sliceno`, which default is set to *`None`*, indicating that it is actually a single file on disk. If `sliceno` is set, it is assumed that the file is sliced, see section B.1.16, and the function will look up that slice of the file only.

B.3 The `JobList` Class

Objects of the `JobList` class are returned by member functions to the `Urd` class. They are used to group sessions of jobs together.

name	description
`find()`	Return a new `JobList` with only jobs with that method or name in it.
`get()`	Return the latest `Job` with that method or name.
`[<method>]`	Same as `.get` but error if no job with that method or name is in the list.
`as_tuples`	The `JobList` represented as `(method, jid) tuples`.
`pretty`	Return a prettified string version of the `JobList`.
`exectime`	Execution times in total as well as per method.
`print_exectimes()`	Print execution time information to `stdout`.

Detailed description of the functions, where neccessary, follows.

B.3.1 `JobList.find()`

name	default	description
`method`	*Mandatory*	Method or name to find.

Return a new `Joblist` will all jobs in the current `JobList` matching the `method` argument. The matching part is either the unique name of the method's source code, or the name optionally given at build time using the `name=` argument.

B.3.2 `JobList.get()`

name	default	description
`method`	*Mandatory*	Method or name to find.
`default`	*`None`*	Return the latest matching job.

Return the latest job that matches the `method` argument. The matching part is either the unique name of the method's source code, or the name optionally given at build time using the `name=` argument. If no matches are found, it will return the `default` argument.

B.3.3 `JobList.print_exectimes()`

name	default	description
`verbose`	*`True`*	In addition to total time, print execution time for each method in list.

Print total execution time for the `Joblist`, and, conditionally, execution time for each job in the list, to `stdout`.

B.4 The `Dataset` Class

The `Dataset` class is used to operate on small or large datasets stored on disk. It decays to a (unicode) string when pickled.

name	**description**
`columns`	A `dict` from column to properties, such as type, min, and max values.
`previous`	The dataset's previous dataset, if it exists, *None* otherwise.
`parent`	The dataset's parent dataset, if it exists, *None* otherwise.
`filename`	The dataset's filename, if it exists. (`csvimport` sets this.)
`hashlabel`	Column used for hash partitioning, or *None*.
`caption`	The dataset's caption.
`lines`	A `list` with number of lines per slice.
`shape`	A tuple containing number of columns and number of lines in dataset.
`link_to_here()`	Used to associate a subjob's dataset with the current job, see section 4.9.
`merge()`	Merge this dataset with another dataset, see section B.4.2.
`chain()`	A `DatasetChain` object, see section B.5
`iterate_chain()`	Iterator over chains, see chapter 6.
`iterate()`	Iterator over dataset see chapter 6.
`iterate_list()`	Iterator over a list of datasets, see chapter 6.

Detailed description of the functions, where neccessary, follows.

B.4.1 `Dataset.link_to_here()`

name	**default**	**description**
`name`	`default`	The new name of the dataset.
`column_filter`	*None*	Iterable of columns to include, or *None* to get all.
`override_previous`	`_no_override`	Set this to the new previous.

Use this to expose a subjob as a dataset in your job, like in this example:

```
def synthesis():
    job = build('ex')
    job.dataset().link_to_here(name='new')
```

The current job will now appear to have a dataset named `new`, that is actually a link to the subjob's `default` dataset. It is possible to filter which columns should be visible in the link using `column_filter`. For chaining purposes, it is possible for the link to expose a parent dataset of choice, set using the `override_previous` parameter.

B.4.2 `Dataset.merge()`

name	**default**	**description**

`other`	*Mandatory*	Merge with this dataset.
`name`	“`default`”	Name of new dataset
`previous`	*`None`*	The new dataset’s `previous` dataset.
`allow_unrelated`	*`False`*	Set this if the datasets do not share a common ancestor.

Merge this and other dataset. Columns from the other dataset take priority. If datasets do not have a common ancestor you get an error unless `allow_unrelated` is set. The new dataset always has the previous specified here (even if *`None`*). Returns the new dataset.

B.4.3 `Dataset.chain()`

name	**default**	**description**
`length`	`-1`	Number of datasets in chain. The default value of `-1` will include all datasets in chain.
`reverse`	*`False`*	Reverse order of chain.
`stop_ds`	*`None`*	If set, chain will start at the dataset after `stop_ds`.

This function will return a `DatasetChain` object, see section B.5.

B.5 The `DatasetChain` Class

These are lists of datasets returned from Dataset.chain. They exist to provide some convenience methods on chains.

name	description
`min()`	Min value for a specified column over the whole chain.
`max()`	Max value for a specified column over the whole chain.
`lines()`	Number of rows in this chain, optionally for a specific slice.
`column_counts()`	The number of datasets each column appears in.
`column_count()`	Number of datasets in this chain that contain a specified column.
`with_column()`	Return a new chain without any datasets that don't contain a specified column.
`none_support()`	Return *`True`* if any dataset in the chain has None-support for this column.
`iterate(...)`	Same arguments as `Dataset.iterate()`. Will iterate over the whole chain.

Detailed description of the functions, where neccessary, follows.

B.5.1 `DatasetChain.min()`, `DatasetChain.max()`

name	default	description
`column`	*Mandatory*	Min/max value of column, see text.

Minimum or maximum value for column over the whole chain. Will be *`None`* if no dataset in the chain contains `column`, if all datasets are empty or if `column` has a type without min/max tracking.

B.5.2 `DatasetChain.lines()`

name	default	description
`sliceno`	*`None`*	If set, return number of lines in speficied slice.

Number of rows in this chain, optionally for a specific slice.

B.5.3 `DatasetChain.column_counts()`

Return a Python `Counter`,`{colname: occurances}`, holding the number of datasets each column appears in. Takes no options.

B.5.4 `DatasetChain.column_count()`

name	default	description
`column`	*Mandatory*	A column name.

Number of datasets in this chain that contain a specified column.

B.5.5 `DatasetChain.with_column()`

name	**default**	**description**
`column`	*Mandatory*	A column name.

Return a new `DatasetChain` with all datasets in this chain containing a speficied column.

B.6 The DatasetWriter Class

The DatasetWriter class is used to create datasets. Datasets could be stand-alone, part of a chain, or an extension (new columns) to an existing dataset.

The class has a number of member functions, described below, that may be used for dataset creation. Alternatively, the new dataset could be set up using the DatasetWriter *constructor*. The constructor approach is currently only documented in the source code, see dataset.py.

name	description
add()	Add a new column to the dataset under creation.
hashcheck()	Check if value belongs in current slice.
set_slice()	Set which slice that will receive the next write.
enable_hash_discard()	Make the write functions silently discard data that does not hash to the current slice.
get_split_write()	Get a writer object, see section 5.9.2.
get_split_write_list()	Get a writer object, see section 5.9.2.
get_split_write_dict()	Get a writer object, see section 5.9.2.
discard()	Discard the dataset under creation.
finish()	Call this if dataset is to be used before creating job finishes, e.g. if the dataset under creation is input to a subjob.

Detailed description of the functions, where neccessary, follows.

B.6.1 DatasetWriter.add()

name	default	description
colname	*Mandatory*	Name of new column.
coltype	*Mandatory*	Type of new column.
none_support	*False*	Set to *True* to allow storing *None*s.

Add a new column to a dataset in creation. This example will create an age column of type number, where the values could also be *None*.

```
dw.add('age', 'number', none_support=True)
```

All dataset types are described in chapter 5.

B.6.2 DatasetWriter.hashcheck()

name	default	description
value	*Mandatory*	Some data/

Check if a value belongs to the current slice. Return *True* if value belongs to the current slice, *False* otherwise.

B.6.3 DatasetWriter.set_slice()

name	default	description
sliceno	*Mandatory*	Slice number to use for writing.

Specify which slice that will receive the next write(s). Use this if writing data in `prepare()` or `synthesis()`.

B.6.4 `DatasetWriter.enable_hash_discard()`

Takes no options. Set this in each slice or after each `set_slice()` to make the writer discard values that do not belong to the current slice.

B.7 The `Urd` Class

name	**description**
`get()`	Get an Urd item from a specified list and timetamp.
`latest()`	Get the latest Urd item for a specified list.
`first()`	Get the first Urd item for a specified list.
`peek()`	Get an Urd item from a specified list and timetamp without recording.
`peek_latest()`	Get the latest Urd item for a specified list without recording.
`peek_first()`	Get the first Urd item for a specified list without recording.
`since()`	Get all timestamps later than a specified timestamp for a specified list.
`list()`	List all Urd lists
`begin()`	Start a new Urd session.
`abort()`	Abort a running Urd session.
`finish()`	Finish a running Urd session and store its contents.
`truncate()`	Discard all Urd items later than a specified timestamp for a specified list.
`set_workdir()`	Set the target workdir.
`build()`	Build a job.
`build_chained()`	Build a job with chaining.
`warn()`	Add a warning message to be displayed at the end of the build.

Detailed description of the functions, where neccessary, follows.

B.7.1 `Urd.get()`

name	**default**	**description**
`path`	*Mandatory*	
`timestamp`	*Mandatory*	

Get an Urd item with specified list and timestamp. The operation is recorded in the current Urd session.

B.7.2 `Urd.latest()`

name	**default**	**description**
`path`	*Mandatory*	

Get the latest job in a specified Urd list. The operation is recorded in the current Urd session.

B.7.3 `Urd.first()`

name	**default**	**description**
`path`	*Mandatory*	

Get the first job in a specified Urd list. The operation is recorded in the current Urd session.

B.7.4 `Urd.peek()`

name	default	description
`path`	*Mandatory*	
`timestamp`	*Mandatory*	

Same as `.get()`, but without recording the dependency.

B.7.5 `Urd.peek_latest()`

name	default	description
`path`	*Mandatory*	

Same as `.latest()`, but without recording the dependency.

B.7.6 `Urd.peek_first()`

name	default	description
`path`	*Mandatory*	

Same as `.first()`, but without recording the dependency.

B.7.7 `Urd.since()`

name	default	description
`path`	*Mandatory*	
`timestamp`	*Mandatory*	

Return a list of all timestamps more recent than the input `timestamp` for a specified Urd list.

B.7.8 `Urd.list()`

Return a list of all available Urd lists.

B.7.9 `Urd.begin()`

name	default	description
`path`	*Mandatory*	
`timestamp`	*Mandatory*	
`caption`	`None`	
`update`	`False`	

Start a new Urd session.

B.7.10 `Urd.abort()`

Abort the current Urd session, discard its contents.

B.7.11 `Urd.finish()`

name	default	description
`path`	*Mandatory*	
`timestamp`	*Mandatory*	
`caption`	`None`	

Finish the current Urd session and store it in the Urd database.

B.7.12 `Urd.truncate()`

name	default	description
`path`	*Mandatory*	
`timestamp`	*Mandatory*	

Discard everything later than `timestamp` for the specified Urd list.

B.7.13 `Urd.set_workdir()`

name	default	description
`workdir`	*Mandatory*	

Set target workdir. It can be set to any workdir present in the Accelerator's configuration file.

B.7.14 `Urd.build()`

name	default	description
`method`	*Mandatory*	Method to build.
`options`	`{}`	Input options.
`datasets`	`{}`	Input datasets.
`jobs`	`{}`	Input jobs.
`name`	`None`	Record job using this name instead of method name.
`caption`	`None`	Optional caption
`workdir`	`None`	Store job in this workdir.

Build a job. If an Urd session is running, the job and its dependencies will be recorded.

B.7.15 `Urd.build_chained()`

Build a chained job. Same options as `.build()`. See chapter 5 for more information.

B.7.16 `Urd.warn()`

Print a string to `stdout` when the build script ends with no errors.

name	default	description
`line`	*Mandatory*	Some string