

Expertmaker Accelerator

User's Reference



Glossary

automata	a script run by urd dispatching or recalling jobs
chain	a one-directional list of datasets
dataset	efficient storage of data
job	a running or completed method
jobid	a link to a successfully completed method
method	main source file of program to be executed by framework
package	a location where methods are stored
urd	dispatcher and job database
workdir	a location where job data and metadata is stored

Chapter 1

Introduction

The Accelerator is a very efficient environment for running and scheduling big data tasks. It has been in continuous development since 2012, and has been used in seven different projects, of which three has been live to customers, and the other four was data analysis and insight projects.

Due to its minimalistic implementation, the Accelerator is capable of working at high speed with terabytes of data with billions of lines on a single computer. Two things stand out: first, all jobs are bookkeeping in a novel way; and second, data is represented and accessed with close to zero overhead. The end result is a globally optimised data processing machine with a wide spectrum of uses.

Some Accelerator features

- **Data integration** - The Accelerator has been used in several projects with different customers and data, and has thereby adopted to a number of formats and cases.
- **Efficient Data Access** - Data is streamed through jobs using low level operating- and filesystem primitives.
- **Simple job tracking** - Easy to find source data, easy to use results from other users.
- **Transparency** - Total context for all jobs ever run is stored and straightforward to retrieve. Any job could be retrieved or replayed.

Chapter 2

Overview

2.1 Introduction

The Accelerator platform is a complete environment designed for high performance parallel computing on large sets of data. On a high level, the platform is composed of two parts: a server providing a set of features for efficient parallel processing of large amounts of data, and a client that dispatches jobs on the server and provides a lookup of all data in a project.

A user of the platform may use or write two kinds of programs, denoted *methods* and *automatas*. Methods are parallel programs executed on the server, and automatas are high level scripts used to control flow and parameters to and from the methods running on the server. Programs are written in the Python programming language¹. In theory, any language may be used, and critical parts of the internals of the Accelerator are written in the C programming language.

Methods that are running or have completed running are called jobs. The server stores results and parameters from all jobs in a database that is partitioned into what is called workdirs. The client, on the other hand, keeps a log, called the *Urd log*, which is bookkeeping all dispatched jobs and associated parameters and dependencies. Interestingly, data is also stored in jobs, so retrieving data is no different from retrieving jobs.

Here is a very simple example

```
# a_helloworld.py:
def analysis(sliceno):
    print("Hello, world: %d" % (sliceno,))

# automata.py:
def main(urd):
    urd.begin('test')
    urd.build('helloworld')
```

The program is executed by calling

```
% automatarunner.py
```

This example will print the string Hello, world: plus a digit for each slice a number of times to standard out. In more detail, the automata is dispatching the

2.2 Dispatch, Dependencies, and Jobids

The framework is basically a job dispatcher and dependency tracker. Input to a job is parameters and source code, called method. When a job is run, a directory is created in the workdir. A pointer to this directory, called a jobid, is returned upon finished execution. Inside the directory, everything needed to run the job is stored, for example input parameters, as well as all results created during execution.

The framework keeps information for each job that has been successfully completed for dependency tracking. Dependency tracking is used for two things: first, it avoids re-execution of jobs that have been run before; and second, it permits jobs to depend on already finished jobs.

Re-execution is avoided if a job is issued with exactly the same source code and input options. If source code is not equal, or if any input to the job is different, the job will be executed again. All successful runs of a job will be kept.

A job may have reference to other jobs as input. If these references are valid, the running job will have access to all information from the reference jobs.

2.3 Datasets and Chaining

A particularly efficient way to store data is by using the dataset. Datasets reside inside jobs, abstracting databases to jobs containing data. Data is stored so that it can be retrieved row by row at a very high pace and requesting from one to any number of columns.

Data in a dataset is separated into slices, where on a single server, the number of slices is close to the number of actual CPU cores for best performance.

A dataset may be hashed on one of its columns. Hashing means that each slice contains a mutually exclusive set of column data.

¹Both Python 2 and Python 3 are supported.

Typically, a dataset is created for each file that is imported. When several files are imported, the jobids may be chained, that is, each new jobid contains a pointer to the previous job. Using the previous-pointers, datasets could be iterated one after another without interruption.

2.4 Job Bookkeeping

A prominent feature of both the client and the server is the ability to immediately recall instead of recompute previously executed jobs. Any job that has been successfully executed may be recalled, provided that the job is submitted again with exactly the same input and source code. The framework uses a hash of the method's source code to decide if it has changed or not.

The combination of bookkeeping and the client/server architecture provides a very powerful environment for accessing and depending on previous result. Not only can a user's own results be recalled, but also those from another user or from the "production" user in a live environment. In a typical application, this dramatically increases performance while minimizing the probability of errors. It also allows for a high level of transparency. Past jobs could be replayed in a controlled manner, and the complete context with respect to input arguments of any job could be retrieved and used to answer "what-if"-scenarios.

2.5 Workdirs

A workdir is where results are stored. One can think of it as a directory populated with jobid-directories. Workdirs are specified with absolute paths in the server filesystem. A workdir has a fixed slicing, and all workdirs in a project need to have the same number of slices.

2.6 Method

Code is executed on the framework is written in methods. Methods allow for parallel execution and more.

2.7 Urd

Urd provides a scripting environment for running and recalling jobs. It has a transaction-log database that stores information about jobs that have been run and their inter-dependencies. Since data is also stored in jobs, another of Urd's purposes is to provide lookups to all data in a project.

2.8 Automata

An automata is a script run by urd, in which jobs and data are recalled, and new jobs may be dispatched.

Chapter 3

High Level Control: Urd

3.1 Introduction to Urd

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

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

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

3.2 Urd sessions

Here is an example of a minimal urd session

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

Every job that is dispatched between `begin` and `finish` will be appended to the `test` Urd list with timestamp 20161025. The Urd list will be updated as long as the `finish` function is called, since it is responsible for updating the urd transaction log. Before `finish`, nothing is stored, and it is perfectly ok to omit `finish` during development work.

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

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

and the following rules apply to these options:

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

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

Timestamp resolution

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

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

and it can be truncated as shown in the following examples covering all possible cases.

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

Aborting an Urd Session

When a session is initiated, a new session cannot start until the current has finished. A session may be aborted, however,

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

and aborted sessions are not stored in the urd transaction log.

3.3 Building Jobs

Jobs are dispatched in sessions using the build function. The syntax is as follows

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

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

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

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

3.3.1 Handling Consecutive Jobs

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

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

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

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

3.3.2 Building Chained Jobs

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

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

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

3.3.3 Debug: Why build

By specifying the flag why_build to the automatarunner it is possible to see the reasons for building a job. See for listing 3.3.3.

A more machine-friendly output is possible by specifying why_build=True in the build-request (and not specifying it to the automatarunner).

```

print(urd.build('method3', jobids=dict(firstjob=jid2, secondjob=jid2), name='knut'))
# Would have built from:
# =====
# {
#   "caption": "fsm_method3",
#   "method": "method3",
#   "params": {
#     "method3": {
#       "datasets": {},
#       "jobids": {
#         "firstjob": "test-59",
#         "secondjob": "test-59"
#       },
#       "options": {}
#     }
#   },
#   "why_build": "on_build"
# }
# Could have avoided build if:
# =====
# {
#   "method3": {
#     "test-60": {
#       "jobids": {
#         "firstjob": "test-56"
#       }
#     }
#   }
# }

```

Listing 1: Example output from why_build.

3.4 Sessions with dependency

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

Here is an example. Assume that a method, method1, uses some kind of imported transaction logs, called tlog. When dispatching method1, it should be using the latest available tlog. The example shows how the function latest is used for this

```

def main(urd):
    urd.begin('test')
    latest_tlog = urd.latest('tlog').joblist.jobid
    urd.build('method1', datasets=dict(tlog=latest_tlog))
    urd.finish('test', '20161025')

```

Two things have happened here. First, urd has provided a jobid link to the latest available tlog. Second, the dependency of exactly this version of tlog to method1 is recorded in the urd list test for timestamp 20161025. So, if there is a question in the future which version of the tlog database that was used on that date for the method1 function, it is immediately available from urd.

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

```

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

```

And here is an example of running method1 on tlog data from previous month

```
def main(urd):  
    urd.begin('test')  
    tlog = urd.get('tlog', '20160925').joblist.jobid  
    urd.build('method1', datasets=dict(tlog=tlog))  
    urd.finish('test', '20161025')
```

3.5 Avoiding recording dependency

Dependency-recording will be activated on use of the `get`, `latest`, and `first` functions. If, for some reason, the point is to just have a look at the database to see what is in there, it can be done using the `peek`-versions, as presented below:

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

3.6 More on Finding Items in Urd

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

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

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

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

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

```
print(urd.since('0'))  
# ["20160101", "20161004", "20161005", "20161006", "20161007", "20161008"]  
print(urd.since('2016'))  
# ["20160101", "20161004", "20161005", "20161006", "20161007", "20161008"]  
print(urd.since('20161'))  
# ["20161004", "20161005", "20161006", "20161007", "20161008"]  
  
print(urd.since('2016105'))  
# ["20161006", "20161007", "20161008"]  
...  
print(urd.since('2016105 000000'))  
# ["20161006", "20161007", "20161008"]
```

3.7 Truncating and Updating

Urd will always keep a consistent history of all events taken place. Sometimes, however, it makes sense to re-run events from the past. There are two means to achieve this, update and truncate.

update

The `begin`-function takes an optional argument `update`

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

If `update` is `True`, the entry in the `test` list at '20161025' will be updated, unless there has been no change.

Truncate

The `truncate()` class member is used to rollback an arbitrary amount of an urd list.

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

This will rollback everything that has happened in the `test` list back to '20160930'. Internally, urd stores the complete history, however.

3.8 More on Joblist and Jobtuple

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

Jobtuple

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

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

>>> jt
('imprt', 'jid-0')
as expected, and
>>> jt.method
'imprt'

>>> jt.jobid
'jid-0'
but note that
>>> print(jt) # str and encode return jobid only
jid-0
```

JobList

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

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

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

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

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

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

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

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

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

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

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

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

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

Looking up by index returns JobTuple.

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

These conveniences are also supported

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

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

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

3.9 Talking directly to Urd: The Urd HTTP-API

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

Show all stored lists like this

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

Looking up the latest stored job in the test list

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

And see the first stored job in the test list

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

See what is inside the test list stored at 20161025

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

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

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

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


Chapter 4

Method

4.1 Methods

Methods are source files executed by the framework. The nomenclature is that methods turn into jobs when they are assigned input arguments and get executed by the framework. A method may be dispatched by urd, or by another method as a subjob.

4.2 Packages and Source code

Methods are stored in directories, called packages, in the main framework directory. A package needs to be present in the main configuration file, and the package directory has to contain the file `__init__.py` for Python to see it.

There are two limitations, that apply to methods, designed to avoid accidental execution of the wrong source code. For a method file to be accepted by the framework, the filename has to start with the prefix “a_”. Furthermore, the method name, without this prefix must be present on a separate line in the `methods.conf` file for the package.

4.3 Recall and Method Hashing

The framework is applying a hash function to the method source code to determine if a job may be recalled or re-executed. Changing the method source code in an unique way causes a new job to be built from the method code.

A method may use code located in other files, and in that case it is possible to ensure that jobs are dispatched if any of those files change. This is specified in the method using the `depend_extra` list, as for example:

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

You can specify a module object or a filename relative to the module source.

A different scenario is when the method source code needs to be modified, but the change does not alter past behavior, i.e. remake of jobs should be actively avoided. There is an `equivalent_hashes` dict for that, as shown

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

The verifier string is a hash that depends on everything in the method except the `equivalent_hashes` block. It is there to avoid the scenario where you forget the method has `equivalent_hashes` and make changes. When the verifier is wrong (as it will be after editing the method) you will be informed what the correct verifier would be, so you can use that or discard the `equivalent_hashes` block, as appropriate. The tuple on the right can contain any number of old hashes.

4.4 Method Inputs

When a method is dispatched as a job, it will be provided with input from the dispatcher. There are three kinds of input to a method: `jobids`, `datasets`, and `options`. These are specified early in the method source code, such as for example

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

The `jobids` argument is used to input pointers to other finished jobs, `datasets` is used to input other datasets, and `options` is used to input any other type of parameters specifying running behavior of the method. Note that `jobids` and `datasets` are tuples (and a single entry has to be followed by a comma), while `options` is a dictionary. Each will be described in more detail next

4.4.1 Input Jobids

The `jobids` argument is a tuple of jobids linking this job to other jobs. Inside the running method, a jobid from the `jobids` tuple is simply a string that can be passed to various helper functions in order to use results from the corresponding jobs.

4.4.2 Input Datasets

The `datasets` argument is the way to communicate datasets from automata to job. In the running method, the `datasets` variable is a tuple of dataset objects ready to use. The dataset class is described in a dedicated chapter.

4.4.3 Input Options

The `options` argument is of type `dict` and used to pass various information from the dispatcher (typically `Urd` or a method) to a method. Information could be integers, strings, enumerations, sets, lists, and dicts in a recursive fashion. Options may have a default value.

Options are defined in a method like this

```
options = dict( ... ) # or
options = { ... }
```

Options are easiest described by examples, and such will be presented in the following sections. The remaining of this section is dedicated to describe the formal rules for option typing and assignment.

- an input value is required to be of the correct type.
- An input may be left unassigned, unless
 - typing is `RequiredOptions()`, or
 - `OptionEnum` without default
- 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`).
- If typing is specified as a value, this is the default value if left unspecified.
- If typing is specified as a class name, default is `None`.
- Values are accepted if they are valid input to the type's constructor, i.e. `3` and `'3'` are valid input for an integer.
- `None` is a valid input unless
 - `RequiredOptions()` and not `none_ok` set
 - `OptionEnum()` and not `none_ok` set
- All containers can be specified as empty (?)
- Complex types (like dicts, dicts of lists of dicts, ...) never enforce specific keys, only types. (`{'a': 'b'}` is a valid value for `{'foo': 'bar'}`) (?)
- containers with a type in the values default to empty containers (otherwise the specified values are the default contents)

Typing and default values are presented in the following sections. Default values are assigned if there is no input. Note that definition of type is that the constructor must accept the value provided, for example the string `'3'` is a valid input for the `int` constructor.

Unspecifieds

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

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

Here, `length` could be set to anything.

Scalars

Scalars are either explicitly typed, as

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

or implicitly with default value like

```
options = dict(length=3)
# Requires an intable value, default is 3 if left unassigned
```

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

Strings

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

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

A default value may be specified as follows

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

And a string required to be specified and non-empty as

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

In some situations, an example string is convenient

```
# Requires string, provides example (NOT default value)
options = dict(name=OptionString('bar'))
```

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

Enums

Enumerations are convenient in a number of situations. They are typed as follows

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

A default value may be specified like this

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

or, like this, accepting None

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

The none_ok flag may be combined with a default value too.

Furthermore, the asterisk-wildcard may be used too

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

Lists and Sets

Lists and sets are specified like this

```
# Requires list of intable, defaults to empty list
options=dict(foo=[int])
```

and

```
# Requires set of intable, defaults to empty set
options=dict(foo={int})
```

More complex stuff

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}})
```

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

Date and Time

The following date and time related types are supported: datetime, date, time, and timedelta. A typical usecase is as follows

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

and with a default assignment

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

JobWithFile

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

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

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

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

In the method, the JobWithFile object has these members

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

Where the full filename of the file is available through

```
from extras import full_filename
print full_filename(filename, '')
```

The mandatory string (which is empty in this case, is a filename extension. **Will be changed!**)

4.5 Code Flow: prepare, analysis, and synthesis

There are three pre-defined functions in a method, prepare, analysis, and synthesis, and they are always run in that order. prepare and synthesis are single threaded, while analysis provides parallel execution.

The following combinations are valid and are of practical use

- synthesis-only, used for a single threaded program
- analysis-only, used for completely parallel tasks
- prepare + analysis, setup and run a parallel job
- analysis + synthesis, run a parallel job and combine outputs
- prepare + analysis + synthesis, setup, run, and combine a parallel task.

All the three functions take options, jobids, and datasets as optional arguments. The analysis function takes a required argument sliceno, which is an integer between zero and the total number of slices minus one. This is the unique identity indicator for the analysis process.

Return values may be passed from one function to another. What is returned from prepare is called prepare_res, and may be used as input argument to analysis and synthesis. The return values from analysis is available as analysis_res in synthesis. The return value from synthesis is stored permanently in the jobdir. A complete example may look like

```
options = dict(length=4)
datasets = ('transaction_log')

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

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

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

4.5.1 The params Variable

The params variable contain all input and initialization parameters for a job, such as the caption

```
def synthesis(params):
    print params.caption
# urd_dispatched_method1
```

The params variable contains lots of information that is typically not required in a regular method. For a method developer, the most important members are

params.package	# which package the method source code is stored in
params.slices	# number of slices for the workdirs in use
params.caption	# may be specified when building a job
params.seed	# seed to initialize a random number generator
params.starttime	# execution start in epoch time

but params does also contain all options, jobids, and datasets inputs.

4.5.2 Accessing Another Job's params using job_params

The params data structure contains all input and initialization data for a job. To access another job's params variable, just feed the job's jobid into the job_params function.

```
from extras import job_params

jobids = ('anotherjob',)

def synthesis():
    print jobids.anotherjob
    # will print something like 'jid-0_0'
    print job_params(jobids.anotherjob).options
    # will print something like {length: 3}
```

4.6 More on Intermediate and Result Files

4.6.1 Sharing Data Inside a Job

Data is easily shared between `prepare`, `analysis`, and `synthesis` as follows.

- what is returned in `prepare` is available as `prepare_res` in `analysis` and `synthesis`
- what is returned in `analysis` is available as `analysis_res` in `synthesis`. `analysis_res` is an iterator.
- what is returned in `synthesis` is a persistent file in the job catalog referenced by `result`.

The blob module

The simplest way to share intra job data is using the blob module.

```
import blob
def synthesis()
    blob.load('filename')
```

saving data is as follows

```
blob.save(data, filename, sliceno=None, temp=None)
```

`sliceno` and `temp` are optional. If `sliceno` is set, data is stored in a sliced file. This is typically used in `analysis`, where each thread will save its own file. The argument `temp` is used for file persistence. By default, files are stored permanently when a job terminates successfully. Setting `temp` to `True` removes them upon completion of the job. Temporary files are useful when communicating data between the functions in the method (and not using the res-files) or in debugging.

4.6.2 debug help

There is also a more advanced debug functionality relating to `temp`.

```
from extras import Temp
def analysis(sliceno):
    ...
    blob.save(data, filename, sliceno=sliceno, temp=Temp.DEBUG)
    # or
    blob.save(data, filename, sliceno=sliceno, temp=Temp.DEBUGTEMP)
```

where the first only stores when `-debug` is specified, and the other always stores but removes unless `-debug` is set.

4.7 Subjobs

Jobs may launch subjobs. If the jobs are already built, they will be immediately linked in. The syntax is as follows, assuming we build the jobs in prepare:

```
import subjobs

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

The `subjobs.build` uses the same syntax as `urd.build` described elsewhere, so `options`, `datasets`, `jobids`, and `caption` are available. Similarly, subjob `jobid` is returned.

If there are datasets built in a subjob, these will not be explicitly available to Urd. Instead, the dataset definition may be copied to the launching method like this

```
from Dataset import dataset

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

the `name` argument is optional, the `name` default is used if left empty, corresponding to the default dataset.

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 `callin` method, like this

```
import subjobs

depend_extra = ('a_childjob.py',)

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

Chapter 5

Dataset

5.1 Introduction

The dataset is the preferred way to store large amounts of data. The dataset is the container for fast and simple access to data. Data in a dataset are stored as a matrix in rows and columns. Using the dataset, data is simple to access at very high speed.

Datasets are created in methods, and any job may contain zero, one, or more datasets. The most obvious use of a dataset is the `csvimport` method that creates a dataset from an input file. But a job may contain more than one dataset. This allows for efficient storage and access of data in some common practical situations. For example, a filtering job may split the input dataset in two or more output datasets that can be accessed independently.

For performance reasons, datasets are typically split into several slices, where each data row exists in exactly one of the slices. The actual slicing may be carried out in any fashion, like round robin, or even random, but an interesting approach is to slice according to the hash value of a certain column. Slicing according to a hashed column ensures that all rows with a certain hash column value ends up in the same slice.

This chapter covers ... chaining, where datasets grows in number of lines, and column appending, where datasets grow in number of columns.

5.2 Dataset access

A method's dataset inputs are specified in the `datasets` tuple. In a running job, the datasets in the input tuple are objects of the `Dataset` class. For example

```
datasets = ('tlog',)

def synthesis():
    print(datasets.tlog.filename)
```

In this example, the `tlog` input dataset is an object of class `Dataset` when the method is running, so printing its filename is straightforward. Most common dataset operations are available as member functions to the `Dataset` class, so this automatic instantiation is very convenient.

A dataset object can also be instantiated from a jobid, like this

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

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

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

where the latter approach is a bit more flexible in that it uses parameters for both jobid and name.

5.3 Dataset Properties

The dataset class has a number of member functions and attributes that is intended to make it simple to work with datasets.

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())
    # ['GTIN', 'date', 'locale', 'subsource']
```

Column Properties

For each column, the name, type, and if applicable, the min and max values are accessible like this

```
# each key, i.e. column, has a number of properties, of which the
# most important ones are shown below
print(datasets.source.columns['locale'].type)
# ascii
print(datasets.source.columns['locale'].name)
# locale
print(datasets.source.columns['locale'].min)
# 3
print(datasets.source.columns['locale'].max)
# 9
```

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. From an efficiency perspective, the variance of the numbers in this list should be low, so that all slices will contain about the same amount of data.

Dataset Shape (i.e. Number of Columns and Total Number of Lines)

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

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

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

Hashlabel

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

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

Filename and Caption

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

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

It is possible to set a caption when creating a dataset. The caption is entirely user-defined and has no function in the Accelerator. The caption is accessible like this

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

Chains

When a dataset is created, it is optional to input a link to another dataset using the parameter `previous`. This has effect on the dataset iterators, which may continue to iterate over dataset boundaries when one dataset is exhausted and continue to the next. This will be described in another section in more detail. The previous dataset is available as an attribute

```
print(datasets.source.previous)
# neu4-4893/default
```

5.4 Column Typing

The dataset columns are typed. All types are shown in table 5.1. Some of the types are explained in more detail in the following.

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

Table 5.1: Available dataset column types.

5.4.1 Arbitrary precision numbers: number

The type `number` is integer when possible and float otherwise. It can handle very large numbers, up to $\pm(2^{1007} - 1)$. Integers are forced using `number:int`, and it accepts trailing decimal zeroes like 7.0, 4.000 etc. This is useful when typing datafiles where numbers are integers but have trailing zero decimals.

The number type occupies a minimum of nine bytes, where eight is for the number and the additional byte is a marker.

5.4.2 Standard fixed size numbers

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

5.4.3 Types relating to time

The `date`, `time`, and `datetime` are compatible with Python's corresponding classes. `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'))
```

5.4.4 String types

To be completed.

5.5 Create New Dataset

Datasets are either created in prepare + analysis, or in just synthesis.

5.5.1 Create in prepare + analysis

A simple example writing three columns to the default dataset is presented next. Note that the example creates a dataset chain, linking the currently created dataset to the dataset that is input with the name previous.

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

def prepare():
    dw = DatasetWriter(
        hashlabel = 'X',
        previous = datasets.previous,
    )
    dw.add('X', number)
    dw.add('Y', number)
    dw.add('Z', number)
    return dw

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

Note that the order of the variables in the `dw.write` function call is the same as the order of the `add` calls in `prepare`¹.

`DatasetWriter` takes a number of optional arguments such as `caption` and `filename`. The argument “name” specifies the name of the dataset, which is set to be “default” when unassigned. Several datasets can be created in the same method using more than one `datasetwriter` instance with different “name”s.

There is some flexibility in the way the `write` function may be called

```
dw.write_dict({column: value})
dw.write_list([value, value, ...])
dw.write(value, value, ...)
# or even
dw.writers[name].write(value) # return True if hashed to correct slice
```

5.5.2 Creating Hashed datasets

If `hashlabel` is set, one can use `dw.hashcheck(value)` to check if `value` belongs to the slice. It is also possible to just call the writer since it will discard anything not belonging to the correct slice.

5.5.3 Create in synthesis

There are two options if the dataset is to be created in synthesis. One in to set the slice number first

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

while the other is to use one of these functions

```
dw.get_split_write_dict()({column: value})
dw.get_split_write_list()([value, value, ...])
dw.get_split_write()(value, value, ...)
```

that will assign the data to the correct slice automatically.

5.5.4 Placeholder: Creating datasets more manually

¹in case `write` is called with a dict, the order is unknown, but then names are looked up using the dict keys.

5.6 Appending new columns to an existing Dataset

With minimal overhead, the dataset supports adding new columns to an existing dataset. This is implemented by storing the new column data together with a pointer to the dataset

Appending new columns work exactly the same way as creating a dataset, with the exception that a link to a dataset that is to be appended to is input to the writer constructor. The following example appends one column to an existing dataset while maintaining the chain. Note that appending does only apply to one dataset, and not to the complete chain.

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

def prepare():
    dw = dataset.DatasetWriter(
        parent=datasets.source,
        previous=datasets.previous,
        caption="flattening_attempt_1"
    )
    dw.add(name, type)
    return dw

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

Note that an error is issued if the total number of appended lines does not match the number of lines in the parent dataset.

Chapter 6

Iterator

6.1 Dataset Iterators

Iterators are used to stream dataset data one line at a time into a method. Data may be streamed from a single dataset, or from a chain of datasets, one dataset at a time.

6.1.1 Basic Iteration

The iterator iterates over specified data columns. The output will be one line at a time, formatted as a tuple.

A Simple Parallel Iterator Invocation

Read variables X and Y in all slices in parallel

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

data is iterated over all rows and all datasets in the chain, running in increasing jobid direction.

Iterate over all data in synthesis

It is possible to iterate over all slices in a single loop by specifying sliceno=None as follows

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

slices will be iterated one at a time in increasing order.

6.1.2 Special cases, iterating over all or a single column

If column names are omitted, the iterator will produce a tuple of data including all columns.

```
for items in iterate_datasets(sliceno, None, jobids):
    print(items) # is a tuple of all columns
```

Also, if only one column is specified, it is possible to have the iterator produce a scalar directly

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

# alternative 2, target is tuple
for user, in iterate_datasets(sliceno, ('USER',), jobids):
    userset.add(user)

# alternative 3, input is string, not list
for user in iterate_datasets(sliceno, 'USER', jobids):
    userset.add(user)
```

Both styles are supported by filters and translators.

6.1.3 Halting Iteration

Iteration over a dataset chain will continue until all data is exhausted. Sometimes, one wish to limit the number of iterations. There are several mechanisms for that, and they may be combined in the same expression. If so, iteration will be over the shortest range of the conditions.

Halting using length

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

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

Halting using stop_jobid

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

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

Halting using another job's input parameters

```
for user, item in datasets.source.iterate_chain(
    sliceno,
    columns=('user', 'item',),
    stop_jobid = {jobids.preprocess: 'source',}):
```

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

It is also possible to iterate over a specified data range. This works for datasets that are sorted on the column of choice.

Iterating over a data range

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

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

6.1.4 Iterating in the reverse direction

It is possible to iterate the dataset chain in the backwards direction by specifying reverse=True, but note that iteration is always in the forward direction within each dataset.

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

@@@ HASHLABEL FOR AVOIDING MISTAKES!!!

6.1.5 iterate_list

PLACEHOLDER

6.1.6 Translators

Translators transform data values. A translator is either a callable or a dict. Translators are similar to filters, and always executed before filtering.

The idea behind translators and filters is that they provide a way to modify code behavior by supplying functions as iterator options. In this way, it is possible to write re-useable simple methods that still could be altered significantly in behavior.

Callable Translator

A translator function is called with a tuple and is expected to return a tuple of the same length. Using translator functions it is possible to mix different columns with each other before sending them to the iterator output. Here is an example

```
def merger(user, item):
    return "%s:%s" % (user, item), None
for merge, _ in iterate_datasetchain(None, ('user', 'item',), tip_jobid=jobid,
                                     translator=merger):
    ...
```

The purpose of this translator is to convert each (user, item) tuple to a string user:item. This is the first output of the translator and iterator and is stored in the merge variable. The second output variable is not used in this application, but a variable still has to be assigned.

Translator dict

One or more columns may be translated independently using a translator dictionary, specified as {name: translation}. A translation may be either a dict or a callable. Examples of both kinds are shown below.

First an example illustrating the use of a translation dict. Here, integers are translated into more comprehensible strings.

```
mapper = {2: 'HUMANLIKE', 4: 'LABRADOR', 5: 'STARFISH',}
for animal in iterate_datasetchain(None, 'NUM_LEGS', tip_jobid=jobid,
                                  translator={'NUM_LEGS': mapper,}):
    ...
```

Items missing in a translation-dict yield None.

Second, an example of a translation callable. In this example, each user string is output from the iterator reads backwards.

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

6.1.7 Filters

Filters are run *after* translators.

Filters decide which rows to pass to the iterator output. A filter is either a callable or a dict.

Callable Filter

Callable filters receive the iterator tuple as input. The output must be True for the tuple to be output from the iterator, otherwise the iterator continues reading the next line.

The following two examples, of which the first uses a callable filter, are functionally equivalent

```
# Ex1. filter users that are bearded and longhaired
filter=lambda line: line[1] and line[2]
for user, beard, hair in iterate_datasetchain(None, ('user', 'bearded', 'longhaired'),
                                             tip_jobid=jobid, filters=filter):
    ...

# Ex2. filter users that are bearded and longhaired
for user, beard, hair in iterate_datasetchain(None, ('user', 'bearded', 'longhaired'),
                                             tip_jobid=jobid):
    if beard and hair:
        ...
```

Filter Dict

It is possible to filter on one or more columns independently using a dict. All filters must be True for a line to be output from the iterator. Here are two examples. The first example will remove all lines except the ones with valid users. The second example will only keep lines with movie items. (The fact that book is false is redundant — missing keys will result in discarded lines.)

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

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

Filter by Column Values

Column values could also be used directly, i.e. the values get evaluated by Python. For example, assume there is a column hasbeard with values being boolean integers, i.e. 1 or 0. Bearded users may be collected by

```
for user, beard in iterate_datasetchain(None, ('user', 'bearded',), tip_jobid=jobid,
                                         filters={'bearded': None}):
    bearded.add(user)
```

This may seem strange at first, but it works because the key for the bearded column exists, and the value is None, and not a callable nor a dict.

6.1.8 Callback

It is also possible to inject callback functions, either before or after each dataset is loaded. If sliceno=None, i.e. iteration runs over all slices of all datasets, it is even possible to have callback between slice change.

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

```
# pre_callback once per dataset
def prefun(jobid):
    print(jobid)
for user, item in iterate_datasetchain(sliceno, ('user', 'item',), tip_jobid=jobid,
                                       pre_callback=prefun):
    ...
```

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

```
# callback once per slice
def prefun(jobid, sliceno):
    print(sliceno, jobid)
for user, item in iterate_datasetchain(None, ('user', 'item',), tip_jobid=jobid,
                                       pre_callback=prefun):
    ...
```

Post-callback is defined similarly.

Skipping Datasets and Slices from Callbacks

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

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

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

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

Chapter 7

standard methods

To cover: `csvimport`, `csvexport`, `dataset__type`, `dataset__sort`, `dataset__rehash`, and more.

7.1 csvimport

This method is used to import text files in tabular format. It reads plain text files as well as gzip compressed files.

7.1.1 Arguments

Options

```
filename
separator
labelsonfirstline
labels
hashlabel
quote_support
rename
discard
allow_bad
```

Input `filename` is mandatory and may either be a plain text file or a gzipped file. A path may be specified as part of the filename, and this path is relative to the configuration option `source_directory`. This is to make input files relocatable so that reading files from a different directory will not trigger remaking the import jobs.

The `separator` option may contain any character to be used for separating the columns in the input file. The default separator is comma.

Labels may be used if they appear on the first line. This is the default. If `labelsonfirstline` is set to `False`, labels must be entered using the `label` list option.

An optional `hashlabel` may be assigned, but bear in mind that hashing will occur on the input text data. Thus, data is to be hashed on for example the integer value of a column, rehashing has to take place after typing to integer.

The option `quote_support` makes it possible to read files with quoted values.

It is possible to rename columns while importing using the `rename` option. If used, a dictionary with `source_name` to `dest_name` must be provided. Columns may also be omitted from importing using the `discard` option. If used, a set containing columns to omit must be provided.

Finally, there is an `allow_bad` option that relaxes the input file strictness. If set to `False`, which is the default, `csvimport` will assert an error if there are format errors in the input data. Setting it to `True` makes importing silently ignoring any format errors. Using this option calls for checking the result.

Datasets

```
previous
```

The only input dataset is `previous` which is set to the previous job when creating a chain.

7.1.2 Output

The result of the `csvimport` is a dataset. There is also a `report.txt` stored in the `jobdir` which contains information on number of lines read and more (CHECK THIS!)

7.2 dataset_type

A dataset imported with `csvimport` may be typed using `dataset_type`. This method parses the input data...

7.2.1 Arguments

Datasets

source
previous

Options

column2type
defaults
rename
caption
discard_untyped
filter_bad
numeric_comma

7.2.2 Example Invocation

An example invocation is the following

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

7.2.3 Typing

This section describes all typing options in detail.

Numbers

The *number* type is int or float.

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

Float 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	<i>ignore</i> , will discard trailing garbage
float32e	float64e	<i>exact</i> , error if parsed number does not fit in type
float32s	float64s	<i>saturate</i> , saturate to min/max if number does not fit in type
float32ei	float64ei	<i>exact</i> + <i>ignore</i>
float32si	float64si	<i>saturate</i> + <i>ignore</i>

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	<i>auto</i> , avoid and use a deterministic type if possible
int32_0i	int64_0i	<i>auto</i> , ignore trailing garbage
int32_8	int64_8	<i>octal</i>
int32_8i	int64_8i	<i>octal</i> , ignore trailing garbage
int32_10	int64_10	<i>decimal</i>
int32_10i	int64_10i	<i>decimal</i> , ignore trailing garbage
int32_16	int64_16	<i>hexadecimal</i>
int32_16i	int64_16i	<i>hexadecimal</i> , ignore trailing garbage

7.2.4 Integers Stored as Floats

There are also 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	<i>exact</i> , error if parsed number does not fit in type
floatint32s	floatint64s	<i>saturate</i> , saturate to min/max if number does not fit in type
floatint32ei	floatint64ei	<i>exact</i> + <i>ignore</i>
floatint32si	floatint64si	<i>saturate</i> + <i>ignore</i>

7.2.5 Convert to Boolean

It is common that a column holds values that are to be interpreted as either False or True. There are two types that is useful in this context. The first one is `strbool`, that works like this

`strbool` False if value in ('false', '0', 'f', 'no', 'off', 'nil', 'null', '')
True otherwise

The other is `floatbool` that is True when the float has bits set to one and zero otherwise. There is also a `floatbooli` that ignores trailing garbage characters.

`floatbool` True if float has bits set to one, zero otherwise
`floatbooli` same + *ignore*

7.2.6 Time and Date

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

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

The format is standard Python time formats, see for example

```
# 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'
```

7.2.7 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. Here are the basic types, and variations and options will be described below.

<code>bytes</code>	list of bytes
<code>bytesstrip</code>	list of bytes, strip characters 8-13,32 from start and end
<code>ascii</code>	list of ascii characters
<code>asciistrip</code>	list of ascii characters, strip characters 8-13,32 from start and end
<code>unicode:*</code>	list of unicode characters

The following also takes argument

<code>unicodestrip:*</code>	list of unicode characters, strip characters 8-13,32 from start and end
-----------------------------	---

where the argument is one of `replace`, `ignore`, and `strict`. `strict` will cause error, `ignore` will erase illegal characters.

<code>ascii:*</code>	list of ascii characters
<code>asciistrip:*</code>	list of ascii characters, strip characters 8-13,32 from start and end

where the argument is one of `replace`, `encode`, and `strict`. `strict` will cause error, `ignore` will erase illegal characters, and `encode` is reversible encoding.

7.3 dataset__csvexport

The `dataset_export` method is used to export datasets to text files. It may write chains of datasets, it may write one dataset per slice and more.

7.3.1 Input Options, Datasets, and Jobids

Options

```
filename
separator
labelsonfirstline
chain_source
quote_fields
labels
sliced
```

Datasets

```
datasets
```

jobids

```
jobids
```

The `dataset_export` may write a single output file or one file per slice. The output format is either text or gzipped text, with separators between columns. Filename is specified by the `filename` option, and if `sliced` is set to `True` (`False` is default), the filename should contain a `%03d` or similar for the slice number to appear in the output files.

Any separator may be specified by the `separator` option. Values may be quoted by setting a quote character in the `quote_fields` option. Valid quotes are WHAT?

Default behaviour is to write the names of the columns on the first line. This is suppressed by setting `labelsonfirstline` to `False`. Individual columns may be selected for export using the `labels` list, which defaults to all labels in the first dataset in a chain.

How a chain of datasets is to be exported is controlled by the `chain_source` option. Internally, this is input as `stop_jobid` to the iterator creating.....By default, it exports everything in a chain back to the previous dataset.WHAT ELSE?

7.4 dataset_rehash

Dataset rehash will create a new dataset based on its source dataset. The new dataset will be hashed on the column specified by the `hashlabel` input option.

7.4.1 Input Datasets and Options

Datasets

<code>source</code>	source dataset to rehash
<code>previous</code>	previous dataset that will be chained to

Options

<code>hashlabel</code>	column for hashing, required.
<code>length</code>	Go back at most this many datasets. Default is -1, which goes back to <code>previous.source</code>
<code>caption</code>	Optional caption. A reasonable caption is created automatically if left blank
<code>as_chain</code>	True generates one dataset per slice, False generates one dataset. Default <code>False</code> .

Note that columns typed as `json`, `list`, and `set` cannot be used for hashing.

7.4.2 Example Invocation

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

7.4.3 Hashing Details

Hashing on a column means that each row will be copied to a slice that is determined from the value of the `hashlabel` column. In particular, the value of the row's `hashlabel` column will be fed through a hashing function, and the output modulo the number of slices will be the target slice number. In meta-language, it is something like this

```
from gzutil import siphash
target_sliceno = siphash(cols[hashlabel])
```

7.4.4 Note on Chains

1.

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

2.

Internally, `dataset_rehash` always generates one dataset per slice in a chain. This is also what is returned if `as_chain == True`. Otherwise, all datasets are 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.

7.4.5 Internal Operation

Figure 7.1 shows how rehashing happens...

Each `analysis` reads one complete slice of the source dataset and hashes it into a new dataset. This means that there is one new dataset created per slice. Each of these datasets are chained and named using an integer counter starting at zero. The last slice, however, is named `default`.

If the `as_chain` is set, `dataset_rehash` is finished. Otherwise, in `synthesis`, each of the datasets are read in turn and concatenated into a single dataset. When this is finished, all datasets except the concatenated one is removed from disk.

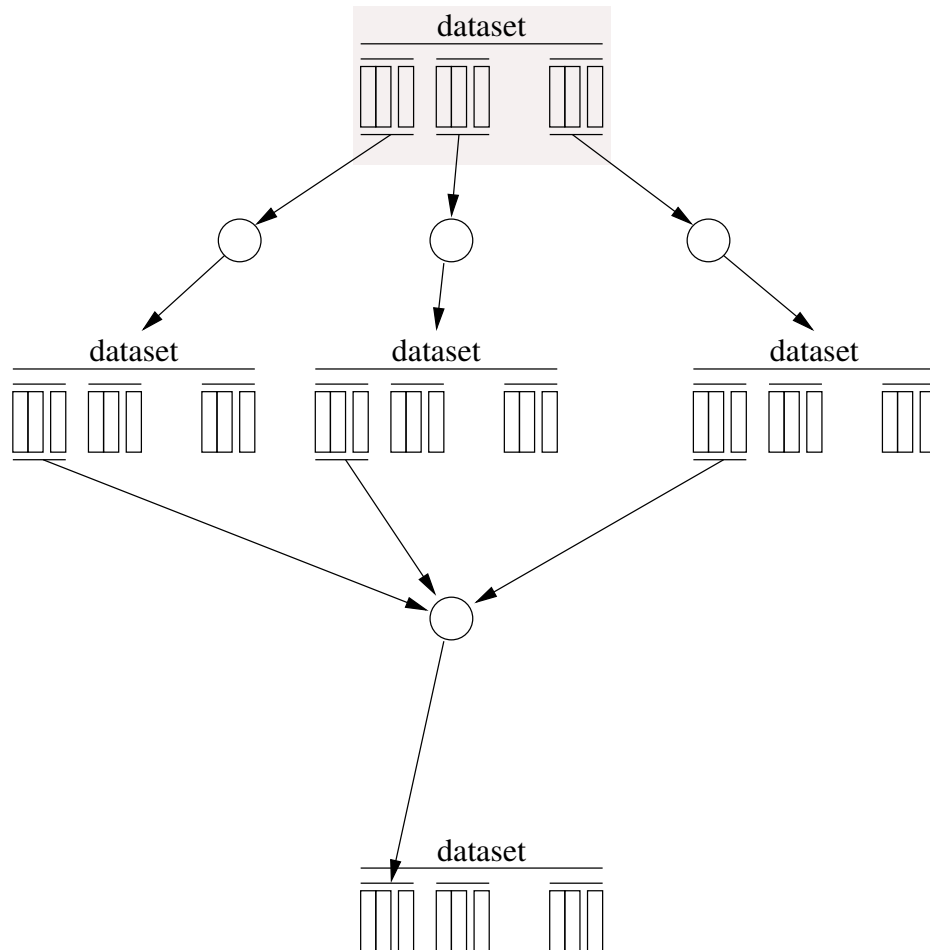


Figure 7.1: `dataset_rehash` data flow

7.5 dataset_filter_columns

The method `dataset_filter_columns` is used to remove columns from a dataset. This method is typically run before methods that operate on all columns of a dataset when only a subset of the columns are required. A typical example is `dataset_rehash` that operates on all columns of a dataset. If a subset such as only one of the columns is to be rehashed, they can be cut out using `dataset_filter_columns`.

Note that no data is copied using this method. It only updates soft links. So execution is typically significantly below a second.

7.6 dataset__sort

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 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.

The method has limited sorting capability. Internally, it sorts one column at a time, and it requires to hold that complete column plus an indexing column in memory simultaneously.

7.6.1 Inputs

Options

```
sort_columns'  
sort_order  
sort_across_slices
```

Datasets

```
source  
previous
```

Data is sorted based on the columns specified in the list `sort_columns`. Data is first sorted on the first column, then the second, and so on.

7.7 dataset__datesplit, dataset__datesplit__discarded

7.8 dataset_checksum, dataset_checksum_chain

Chapter 8

Command Line Tools

8.1 Facts about datasets: dsinfo

The `dsinfo` is a command line tool that gives a compact