1 Introduction
  1.1 Overview .................................................. 3
  1.2 Use Cases ................................................ 3

2 Existing Deployments
  2.1 Digital Earth Australia ................................... 5
  2.2 Digital Earth Africa ...................................... 5
  2.3 Swiss Data Cube ........................................... 5
  2.4 Vietnam Open Data Cube ................................... 5

3 Jupyter Notebooks ........................................... 7

4 Code Recipes ................................................ 9
  4.1 Display an RGB Image ..................................... 9
  4.2 Multi-Product Time Series ................................. 10
  4.3 Polygon Drill ............................................... 10
  4.4 Line Transect .............................................. 11

5 Overview .................................................... 13
  5.1 Overview .................................................. 13

6 Types of Datasets .......................................... 15
  6.1 Indexed Datasets .......................................... 15
  6.2 Ingested Datasets .......................................... 15
  6.3 Provenance ................................................ 16

7 Installation .................................................. 17
  7.1 Miniconda (recommended) ................................. 17
  7.2 Microsoft Windows ........................................ 18
  7.3 Ubuntu ..................................................... 21
  7.4 Mac OS X .................................................. 22

8 Database Setup ............................................. 25
  8.1 Install PostgreSQL ......................................... 25
  8.2 Create Database ........................................... 26
  8.3 Create Configuration File ................................ 26
  8.4 Initialise the Database Schema .............................. 26
19 Virtual Products .......................................................... 111
  19.1 Introduction ................................................................ 111
  19.2 Design ........................................................................ 111
  19.3 Grammar ..................................................................... 112
  19.4 Using virtual products .................................................. 114
  19.5 User-defined transformations ....................................... 114

20 Architecture Guide .......................................................... 117
  20.1 Introduction ................................................................ 117
  20.2 High Level Architecture ............................................... 117
  20.3 Data Model .................................................................. 122
  20.4 Metadata Index ........................................................... 125
  20.5 Data Loading ............................................................... 126
  20.6 Extending Datacube ...................................................... 131

21 Release Process .............................................................. 137
  21.1 Build a version ............................................................ 137
  21.2 Marking it stable ......................................................... 137

22 What’s New ...................................................................... 139
  22.1 v1.7dev ...................................................................... 139
  22.2 v1.6.1 (27 August 2018) ............................................... 139
  22.3 v1.6.0 (23 August 2018) ............................................... 139
  22.4 v1.6rc2 (29 June 2018) ................................................ 140
  22.5 v1.6rc1 Easter Bilby (10 April 2018) .............................. 141
  22.6 v1.5.4 Dingley Dahu (13th December 2017) ................. 143
  22.7 v1.5.3 Purpler Unicorn with Starlight (16 October 2017) . 143
  22.8 v1.5.2 Purpler Unicorn with Stars (28 August 2017) ...... 143
  22.9 v1.5.4 Dingley Dahu (13th December 2017) ................. 143
  22.10 v1.5.3 Purpler Unicorn with Starlight (16 October 2017) . 144
  22.11 v1.5.2 Purpler Unicorn with Stars (28 August 2017) ...... 144
  22.12 v1.5.1 Purpler Unicorn (13 July 2017) ........................ 144
  22.13 v1.5.0 Purple Unicorn (9 July 2017) .......................... 144
  22.14 v1.4.1 (25 May 2017) ............................................... 145
  22.15 v1.4.0 (17 May 2017) ............................................... 145
  22.16 v1.3.1 (20 April 2017) ............................................... 145
  22.17 v1.3.0 ................................................................. 145
  22.18 v1.2.2 ................................................................. 146
  22.19 v1.2.0 Boring as Batman (15 February 2017) ................. 146
  22.20 v1.1.18 Mushroom Milkshake (9 February 2017) .......... 146
  22.21 v1.1.17 Happy Festivus Continues (12 January 2017) .... 146
  22.22 v1.1.16 Happy Festivus (6 January 2017) .................... 146
  22.23 v1.1.15 Minion Party Hangover (1 December 2016) ...... 147
  22.24 v1.1.14 Minion Party (30 November 2016) ................. 147
  22.25 v1.1.13 Black Goat (15 November 2016) .................... 147
  22.26 v1.1.12 Unnamed Unknown (1 November 2016) ......... 147
  22.27 v1.1.11 Unnamed Unknown (19 October 2016) ........... 147
  22.28 v1.1.10 Rabid Rabbit (5 October 2016) ....................... 148
  22.29 v1.1.9 Pest Hippo (20 September 2016) ................. 148
  22.30 v1.1.8 Last Mammoth (5 September 2016) ............... 148
  22.31 v1.1.7 Bit Shift (22 August 2016) ............................ 149
  22.32 v1.1.6 Lightning Roll (8 August 2016) ...................... 149
  22.33 v1.1.5 Untranslatable Sign (26 July 2016) ............... 149
  22.34 v1.1.4 Imperfect Inspiration (12 July 2016) ............ 149
The Open Data Cube provides an integrated gridded data analysis environment for decades of analysis ready earth observation satellite and related data from multiple satellite and other acquisition systems.
CHAPTER 1

Introduction

1.1 Overview

The Open Data Cube is a collection of software designed to:

- Catalogue large amounts of Earth Observation data
- Provide a Python based API for high performance querying and data access
- Give scientists and other users easy ability to perform Exploratory Data Analysis
- Allow scalable continent scale processing of the stored data
- Track the provenance of all the contained data to allow for quality control and updates

The Open Data Cube software is based around the datacube-core library. datacube-core is an open source Python library, released under the Apache 2.0 license.

1.2 Use Cases

1.2.1 Large-scale workflows on HPC

Continent or global-scale processing of data on a High Performance Computing supercomputer cluster.

1.2.2 Exploratory Data Analysis

Allows interactive analysis of data, such as through a Jupyter Notebook.
1.2.3 Cloud-based Services

Using ODC (Open Data Cube) to serve WMS (Web Map Service), WCS (Web Coverage Service), or custom tools (such as polygon drill time series analysis).

1.2.4 Standalone Applications

Running environmental analysis applications on a laptop, suitable for field work, or outreach to a developing region.
CHAPTER 2

Existing Deployments

2.1 Digital Earth Australia

If you are using Digital Earth Australia at the NCI, see the Digital Earth Australia User Guide.

2.2 Digital Earth Africa


2.3 Swiss Data Cube

https://www.swissdatacube.org/

2.4 Vietnam Open Data Cube

http://datacube.vn/

Note:  This is not a comprehensive list.

http://tinyurl.com/datacubeui
CHAPTER 3

Jupyter Notebooks

For interactively exploring data in a Data Cube, we recommend using Jupyter Notebooks.

Several GitHub repositories of example Open Data Cube notebooks are available, showing how to access data through ODC, along with example algorithms and visualisations.

- https://github.com/ceos-seo/data_cube_notebooks
- https://github.com/GeoscienceAustralia/dea-notebooks/
4.1 Display an RGB Image


```python
import datacube
from datacube.storage.masking import mask_invalid_data

query = {
    'time': ('1990-01-01', '1991-01-01'),
    'lat': (-35.2, -35.4),
    'lon': (149.0, 149.2),
}

dc = datacube.Datacube(app='plot-rgb-recipe')
data = dc.load(product='ls5_nbar_albers', measurements=['red', 'green', 'blue'], **query)
data = mask_invalid_data(data)
fake_saturation = 4000
rgb = data.to_array(dim='color')
rgb = rgb.transpose(*rgb.dims[1:]+rgb.dims[:1]))  # make 'color' the last dimension
rgb = rgb.where((rgb <= fake_saturation).all(dim='color'))  # mask out pixels where any band is 'saturated'
rgb /= fake_saturation  # scale to [0, 1] range for imshow

rgb.plot.imshow(x=data.crs.dimensions[1], y=data.crs.dimensions[0],
    col='time', col_wrap=5, add_colorbar=False)
```
4.2 Multi-Product Time Series


```python
import numpy
import xarray
import datacube

query = {
    'time': ('2013-01-01', '2014-01-01'),
    'lat': (-35.2, -35.4),
    'lon': (149.0, 149.2),
}

products = ['ls7_nbar_albers', 'ls8_nbar_albers']

dc = datacube.Datacube(app='multi-prod-recipe')

# find similarly named measurements
measurements = set(dc.index.products.get_by_name(products[0]).measurements.keys())

for prod in products[1:]:
    measurements.intersection(dc.index.products.get_by_name(products[0]).measurements.
    keys())

datasets = []

for prod in products:
    ds = dc.load(product=prod, measurements=measurements, **query)
    ds['product'] = ('time', numpy.repeat(prod, ds.time.size))
    datasets.append(ds)

combined = xarray.concat(datasets, dim='time')
combined = combined.sortby('time')  # sort along time dim
```

4.3 Polygon Drill

Uses `datacube.Datacube` `xarray.DataArray` `datacube.model.CRS` `datacube.Datacube.load()` `xarray.Dataset.where()`

```python
import fiona
import rasterio.features

import datacube
from datacube.utils import geometry

def geometry_mask(geoms, geobox, all_touched=False, invert=False):
    """
    Create a mask from shapes.

    By default, mask is intended for use as a numpy mask, where pixels that overlap shapes are False.
    :param list[Geometry] geoms: geometries to be rasterized
    :param datacube.utils.GeoBox geobox:
    """

    # continue on next page
```
:param bool all_touched: If True, all pixels touched by geometries will be burned in. If false, only pixels whose center is within the polygon or that are selected by Bresenham’s line algorithm will be burned in.

:param bool invert: If True, mask will be True for pixels that overlap shapes.

.. code::

   return rasterio.features.geometry_mask([geom.to_crs(geobox.crs) for geom in geoms],
   out_shape=geobox.shape,
   transform=geobox.affine,
   all_touched=all_touched,
   invert=invert)

.. code::

    def main():
        shape_file = 'my_shape_file.shp'
        with fiona.open(shape_file) as shapes:
            crs = geometry.CRS(shapes.crs_wkt)
            first_geometry = next(iter(shapes))['geometry']
            geom = geometry.Geometry(first_geometry, crs=crs)

            query = {
                'time': ('1990-01-01', '1991-01-01'),
                'geopolygons': geom
            }  

            dc = datacube.Datacube(app='poly-drill-recipe')
            data = dc.load(product='ls5_nbar_albers', measurements=['red'], **query)

            mask = geometry_mask([geom], data.geobox, invert=True)
            data = data.where(mask)
            data.red.plot.imshow(col='time', col_wrap=5)

4.4 Line Transect

Uses datacube.Datacube xarray.DataArray datacube.model.CRS datacube.Datacube.load() xarray.Dataset.sel_points()
```python
:param datacube.utils.Geometry line: line along which to extract the transect
:param float resolution: interval used to extract points along the line (in data
    → CRS units)
:param str method: see xarray.Dataset.sel_points
:param float tolerance: see xarray.Dataset.sel_points

""
assert line.type == 'LineString'
line = line.to_crs(data.crs)

dist = numpy.arange(0, line.length, resolution)
points = [line.interpolate(d).coords[0] for d in dist]
indexers = {
    data.crs.dimensions[0]: [p[1] for p in points],
    data.crs.dimensions[1]: [p[0] for p in points]
}
return data.sel_points(xarray.DataArray(dist, name='distance', dims=['distance']),
    method=method,
    tolerance=tolerance,
    **indexers)

def main():
    with fiona.open('line.shp') as shapes:
        crs = geometry.CRS(shapes.crs_wkt)
        first_geometry = next(shapes)['geometry']
        line = geometry.Geometry(first_geometry, crs=crs)

        query = {
            'time': ('1990-01-01', '1991-01-01'),
            'geopolygon': line
        }

        dc = datacube.Datacube(app='line-trans-recipe')
data = dc.load(product='ls5_nbar_albers', measurements=['red'], **query)
trans = transect(data, line, abs(data.affine.a))
trans.red.plot(x='distance', y='time')
```

Chapter 4. Code Recipes
This section contains information on setting up and running an Open Data Cube. Generally, users will not be required to know most of the information contained here.

5.1 Overview

Follow the steps below to install and configure a new Data Cube instance.
Optional data ingestion (Performance optimisation)
Optional dataset preparation required for 3rd party datasets

Install Data Cube Package and Dependencies
Create a Database to hold the Index

Write Prepare Script
Run Prepare Script

Initialise Database
Define Product Types

Index Datasets

Write ingest config
Run ingestion

Finished
Ready to analyse data
When using the Data Cube, it will contain records about 2 different types of products and datasets.

<table>
<thead>
<tr>
<th>Type of dataset</th>
<th>In Index</th>
<th>Typical data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed</td>
<td>Yes</td>
<td>Created externally</td>
</tr>
<tr>
<td>Ingested</td>
<td>Yes</td>
<td>Created within the Data Cube</td>
</tr>
</tbody>
</table>

### 6.1 Indexed Datasets

Data is available (has a file location or uri), with associated metadata available in a format understood by the Data Cube.

Example:

- USGS Landsat Scenes with prepared `agdc-metadata.yaml`
- GA Landsat Scenes

### 6.2 Ingested Datasets

Data has been created by/and is managed by the Data Cube. The data has typically been copied, compressed, tiled and possibly re-projected into a shape suitable for analysis, and stored in NetCDF4 files.

Example:

- Tiled GA Landsat Data, ingested into Australian Albers Equal Area Projection (EPSG:3577) and stored in 100km tiles in NetCDF4
6.3 Provenance

![Diagram showing the provenance of datasets: RAW Telemetry → Ortho Rectified → NBAR Scene → NBAR Albers Tile]
These installation instructions build a Data Cube environment that can be used to ingest data (with config files from github) and run analytics processes.

The Data Cube is a set of python code with dependencies including:

- Python 3.5+ (3.6 recommended)
- GDAL
- PostgreSQL database

These dependencies along with the target operating system environment should be considered when deciding how to install Data Cube to meet your system requirements.

The recommended method for installing the Data Cube is to use a container and package manager. The instructions below are for Miniconda and PostgreSQL.

Other methods to build and install the Data Cube are maintained by the community and are available at https://github.com/opendatacube/documentation. These may include docker recipes and operating system specific deployments.

### 7.1 Miniconda (recommended)

#### 7.1.1 Install Miniconda

Follow conda installation guide for your platform: https://conda.io/docs/install/quick.html

#### 7.1.2 Configure Miniconda

Add conda-forge channel

```
conda config --add channels conda-forge
```
conda-forge channel provides multitude of community maintained packages. Find out more about it here https://conda-forge.org/

### 7.1.3 Create the environment

```bash
conda create --name cubeenv python=3.6 datacube
```

Activate the environment on **Linux** and **OS X**

```bash
source activate cubeenv
```

Activate the environment on **Windows**

```bash
activate cubeenv
```

Find out more about managing virtual environments here https://conda.io/docs/using/envs.html

### 7.1.4 Install other packages

```bash
conda install jupyter matplotlib scipy
```

Find out more about managing packages here https://conda.io/docs/using/pkgs.html

### 7.2 Microsoft Windows

#### 7.2.1 Miniconda


2. Open the Anaconda Prompt from the Start Menu to execute the following commands.

3. **Add the conda-forge channel**

```bash
conda config --add channels conda-forge
```

The conda-forge channel provides multitude of community maintained packages. Find out more about it here https://conda-forge.org/

4. **Create a virtual environment in conda**

```bash
conda create --name cubeenv python=3.5 datacube
```

5. **Activate the virtual environment**

```bash
source activate cubeenv
```

Find out more about managing virtual environments here https://conda.io/docs/using/envs.html

6. **Install other packages**

```bash
conda install jupyter matplotlib scipy
```

Find out more about managing packages here https://conda.io/docs/using/pkgs.html
Datacube is now installed and can be used in the Anaconda Prompt by activating the `cubeenv` environment.

## 7.2.2 Manual Installation (Fallback)

Only follow these steps if the Miniconda installation does not suit your needs.

### Python 3 environment


   **Note:** If in a restricted environment with no local administrator access, python can be installed by running:
   
   ```
   msiexec /a python-3.6.4.msi TARGETDIR=C:\Python3
   ```

   Or by launching the version 3.6 installer and selecting **not** to install for all users (only single user install).

2. Ensure `pip` is installed:
   
   ```
   cd C:\Python3
   python -m ensurepip
   ```

3. Upgrade and Install python virtualenv:
   
   ```
   python -m pip install --upgrade pip setuptools virtualenv
   ```

4. Create an AGDC virtualenv:
   
   ```
   mkdir C:\envs
   Scripts\virtualenv C:\envs\agdcv2
   ```

   **Note:** 3.5 only workaround: Copy `vcruntime140.dll` from Python install dir into virtualenv Scripts\folder.

5. Activate virtualenv:
   
   ```
   C:\envs\agdcv2\Scripts\activate
   ```

The python virtual environment isolates this python installation from other python installations (which may be in use for other application software) to prevent conflicts between different python module versions.

### Python modules

On windows systems by default there are no ready configured compilers, and so libraries needed for some python modules must be obtained in precompiled (binary) form.

Download and install binary wheels from [https://www.lfd.uci.edu/~gohlke/pythonlibs/](https://www.lfd.uci.edu/~gohlke/pythonlibs/)

You will need to download at least:

- GDAL
- rasterio
• numpy
• netCDF4
• psycopg2
• numexpr
• scipy
• pandas
• matplotlib

The following may also be useful:
• lxml
• pyzmq
• udunits2

Install these packages by running in your Downloads directory:

```bash
pip install *.whl
```

**Note:** It may be necessary to manually replace `*.whl` with the full filenames for each .whl file (unless using a unix-like shell instead of the standard windows command line console).

---

**Note:** For 3.5 only

If there are problems loading libraries. Try:

```bash
copy site-packages/matplotlib/msvcp140.dll site-packages/osgeo/
```

Also, install the python notebook interface for working with datacube example notebooks:

```bash
pip install jupyter
```

---

**Datacube installation**

Obtain a current copy of the datacube source code from GitHub. A simple way is to extract [https://github.com/opendatacube/datacube-core/archive/develop.zip](https://github.com/opendatacube/datacube-core/archive/develop.zip) into a subdirectory of the python environment.

Install the datacube module by running:

```bash
cd datacube-core-develop
python setup.py install
```

---

**Extra instructions for installing Compliance Checker**

```bash
pip install cf_units
```

• Download and install udunits2 from gohlke
• Edit `site-packages/cf_units/etc/site.cfg` with path to udunits2.dll which should be `venv/share/udunits/udunits2.dll`
7.3 Ubuntu

7.3.1 Miniconda

1. Download and install Miniconda using the following instructions https://conda.io/docs/user-guide/install/linux.html
2. Open your favourite terminal to execute the following commands.
3. Add the conda-forge channel
   
   ```
   conda config --add channels conda-forge
   ```

   The conda-forge channel provides multitude of community maintained packages. Find out more about it here https://conda-forge.org/
4. Create a virtual environment in conda
   
   ```
   conda create --name cubeenv python=3.5 datacube
   ```

5. Activate the virtual environment
   
   ```
   source activate cubeenv
   ```

   Find out more about managing virtual environments here https://conda.io/docs/using/envs.html
6. Install other packages
   
   ```
   conda install jupyter matplotlib scipy
   ```

   Find out more about managing packages here https://conda.io/docs/using/pkgs.html

Datacube is now installed and can be used in a terminal by activating the `cubeenv` environment.

7.3.2 Manual Installation (Fallback)

Only follow these steps if the Miniconda installation does not suit your needs.

**Required software**

HDF5, and netCDF4:

```
apt-get install libhdf5-serial-dev libnetcdf-dev
```

GDAL:

```
apt-get install libgdal-dev
```

Optional packages (useful utilities, docs):

```
apt-get install postgresql-doc-9.5 libhdf5-doc netcdf-doc libgdal1-doc hdf5-tools netcdf-bin gdal-bin pgadmin3
```
Python and packages

Python 3.5+ is required. Python 3.6 is recommended.

Download the latest version of the software from the repository and install it:

```
git clone https://github.com/opendatacube/datacube-core
cd datacube-core
git checkout develop
python setup.py install
```

It may be useful to use conda to install binary packages:

```
conda install psycopg2 gdal libgdal hdf5 rasterio netcdf4 libnetcdf pandas
```

**Note:** Usage of virtual environments is recommended

### 7.4 Mac OS X

#### 7.4.1 Miniconda

1. Download and install Miniconda using the following instructions [https://conda.io/docs/user-guide/install/macos.html](https://conda.io/docs/user-guide/install/macos.html)

2. Open Terminal to execute the following commands.

3. **Add the conda-forge channel**

   ```
   conda config --add channels conda-forge
   ```

   The conda-forge channel provides multitude of community maintained packages. Find out more about it here [https://conda-forge.org/](https://conda-forge.org/)

4. **Create a virtual environment in conda**

   ```
   conda create --name cubeenv python=3.5 datacube
   ```

5. **Activate the virtual environment**

   ```
   source activate cubeenv
   ```

   Find out more about managing virtual environments here [https://conda.io/docs/using/envs.html](https://conda.io/docs/using/envs.html)

6. **Install other packages**

   ```
   conda install jupyter matplotlib scipy
   ```

   Find out more about managing packages here [https://conda.io/docs/using/pkgs.html](https://conda.io/docs/using/pkgs.html)

Datacube is now installed and can be used in Terminal by activating the `cubeenv` environment.
7.4.2 Manual Installation (Fallback)

Only follow these steps if the Miniconda installation does not suit your needs.

**Note:** This section was typed up from memory. Verification and input would be appreciated. See also Mac and Homebrew install instructions.

**Required software**

Homebrew:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/\n  →master/install)"
```

HDF5, netCDF4, and GDAL:

```
brew install hdf5 netcdf gdal
```

**Python and packages**

Python 3.5+ is required (3.6 is recommended)

Download the latest version of the software from the repository and install it:

```
  git clone https://github.com/opendatacube/datacube-core
cd datacube-core
git checkout develop
python setup.py install
```

It may be useful to use conda to install binary packages:

```
conda install psycopg2 gdal libgdal hdf5 rasterio netcdf4 libnetcdf pandas
```

**Note:** Usage of virtual environments is recommended
**Attention:** You must have a properly configured Postgres installation for this to work. If you have a fresh install of Postgres on Ubuntu then you may want to configure the `postgres` user password to complete the `postgres` setup.

### 8.1 Install PostgreSQL

Data Cube is using PostgreSQL.

#### 8.1.1 Ubuntu

Ubuntu 16.04 includes packages for PostgreSQL 9.5. On earlier versions of Ubuntu you can use the postgresql.org repo as described on their download page.

Install postgres using `apt`:

```
sudo apt install postgresql-9.5 postgresql-client-9.5 postgresql-contrib-9.5
```

Configure the `postgres` user password to complete the `postgres` setup.

#### 8.1.2 Windows

An easy to install version of PostgreSQL can be downloaded from [https://sourceforge.net/projects/postgresqlportable/](https://sourceforge.net/projects/postgresqlportable/). It can install and run as an unprivileged windows user.

After installing, launch `PostgreSQLPortable.exe` (and place a shortcut in the windows Startup menu).

To prepare the database for first use, enter the following commands in the PostgrSQL Portable window, substituting “u12345” with your windows login user-ID:
create role u12345 superuser login;
create database datacube;

8.1.3 MacOS

Install Postgres.app from http://postgresapp.com/

8.2 Create Database

If you have existing Postgres authentication:

```
createdb datacube
```

or specify connection details manually:

```
createdb -h <hostname> -U <username> datacube
```

Note: You can also delete the database by running `dropdb datacube`. This step is not reversible.

8.3 Create Configuration File

Datacube looks for a configuration file in `~/.datacube.conf` or in the location specified by the `DATACUBE_CONFIG_PATH` environment variable. The file has this format:

```
[datacube]
db_database: datacube

# A blank host will use a local socket. Specify a hostname (such as localhost) to use
# TCP.
db_hostname:

# Credentials are optional: you might have other Postgres authentication configured.
# The default username otherwise is the current user id.
# db_username:
# db_password:
```

Uncomment and fill in lines as required.

See also Runtime Config

8.4 Initialise the Database Schema

The `datacube system init` tool can create and populate the Data Cube database schema

```
datacube -v system init
```
Environment Configuration

It is possible to connect to multiple Data Cube indexes from within the one python process. When initialising a Datacube instance, it will load configuration options from one or more config files. These configuration options define which indexes are available, and any parameters required to connect to them.

9.1 Types of Indexes

At the moment, there are two types of indexes supported, but in the future we expect to support more. The two indexes currently are the standard PostgreSQL backed index, and the other is an extension to the standard index, with additional support for data stored in the S3 AIO format.

The type of index driver to use is defined by the index_driver option in each section of the user config file.

9.2 Runtime Config

The runtime config specifies configuration options for the current user, such as available Data Cube instances and which to use by default.

This is loaded from the following locations in order, if they exist, with properties from latter files overriding those in earlier ones:

- /etc/datacube.conf
- $DATACUBE_CONFIG_PATH
- ~/.datacube.conf
- datacube.conf

Example:
Open Data Cube Documentation, Release 1.5.1+528.g06440e47

```
[default]

db_database: datacube

# A blank host will use a local socket. Specify a hostname (such as localhost) to use TCP.
db_hostname:

# Credentials are optional: you might have other Postgres authentication configured. # The default username is the current user id
# db_username:
# A blank password will fall back to default postgres driver authentication, such as reading your ~/.pgpass file.
# db_password:
index_driver: pg

## Development environment ##

[dev]

# These fields are all the defaults, so they could be omitted, but are here for reference

db_database: datacube

# A blank host will use a local socket. Specify a hostname (such as localhost) to use TCP.
db_hostname:

# Credentials are optional: you might have other Postgres authentication configured. # The default username is the current user id
# db_username:
# A blank password will fall back to default postgres driver authentication, such as reading your ~/.pgpass file.
# db_password:

## Staging environment ##

[staging]

db_hostname: staging.dea.ga.gov.au

[s3_test]

db_hostname: staging.dea.ga.gov.au
index_driver: s3aio_index

Note that the staging environment only specifies the hostname, all other fields will use default values (dbname datacube, current username, password loaded from ~/.pgpass)

When using the datacube, it will use your default environment unless you specify one explicitly eg.

```
with Datacube(env='staging') as dc:
...
```

or for cli commands -E <name>:

```bash
datacube -E staging system check
```
Once you have the Data Cube software installed and connected to a database, you can start to load in some data. This step is performed using the `datacube` command line tool.

When you load data into the Data Cube, all you are doing is recording the existence of and detailed metadata about the data into the index. None of the data itself is copied, moved or transformed. This is therefore a relatively safe and fast process.

### 10.1 Prerequisites for Indexing Data

- A working Data Cube setup
- Some data to load (Links to some freely available *Sample Earth Observation Data.*)

### 10.2 Steps to Adding Data

- **Create a new product** Before the data itself can be added, a product describing the data must be created. Requires creation of a `Product Definition` document (yaml)
- **Index the data** After this step the data is accessible through the datacube. Requires datacube friendly `Dataset Documents` for data which is to be indexed
- **Ingest the data(optional)** After indexing the data you can choose to ingest. This provides the ability to tile the original data into a faster storage format or a new projection system. Requires creation of an ingestion configuration file (yaml).

### 10.3 Product Definition

The Data Cube can handle many different types of data, and requires a bit of information up front to know what to do with them. This is the task of a Product Definition.
A Product Definition provides a short name, a description, some basic source metadata and (optionally) a list of measurements describing the type of data that will be contained in the Datasets of its type. In Landsat Surface Reflectance, for example, the measurements are the list of bands.

The measurements is an ordered list of data, which specify a name and some aliases, a data type or dtype, and some options extras including what type of units the measurement is in, a nodata value, and even a way of specifying bit level descriptions or the spectral response in the case of reflectance data.

A product definition example:

```python
name: ls8_level1_scene
description: Landsat 8 Level 1 Collection-1
metadata_type: eo

metadata:
    platform:
        code: LANDSAT_8
    instrument:
        name: OLI_TIRS
    product_type: level1
    format:
        name: GeoTIFF

measurements:
    - name: 'coastal_aerosol'
      aliases: [band_1, coastal_aerosol]
      dtype: int16
      nodata: -999
      units: '1'

    - name: 'blue'
      aliases: [band_2, blue]
      dtype: int16
      nodata: -999
      units: '1'

    - name: 'green'
      aliases: [band_3, green]
      dtype: int16
      nodata: -999
      units: '1'

    - name: 'red'
      aliases: [band_4, red]
      dtype: int16
      nodata: -999
      units: '1'

    - name: 'nir'
      aliases: [band_5, nir]
      dtype: int16
      nodata: -999
      units: '1'

    - name: 'swir1'
      aliases: [band_6, swir1]
      dtype: int16
      nodata: -999
      units: '1'
```

(continues on next page)
More detailed information on the structure of a product definition document can be found here

A set of Product definitions are supplied here to cover some common Geoscience Australia and other Earth Observation Data.

### 10.4 Loading Product Definitions

To load Products into your Data Cube run:

```
datacube product add <path-to-product-definition-yml>
```

### 10.5 Dataset Documents

Every dataset requires a metadata document describing what the data represents and where it has come from, as well as what format it is stored in. At a minimum, you need the dimensions or fields your want to search by, such as lat, lon and time, but you can include any information you deem useful.

It is typically stored in YAML documents, but JSON is also supported. It is stored in the index for searching, querying and accessing the data.

The data from Geoscience Australia already comes with relevant files (named `ga-metadata.yaml`), so no further steps are required for indexing them.

For third party datasets, see Dataset Preparation Scripts.

A Dataset Documents is required to accompany the dataset for it to be recognised by the Data Cube. It defines critical metadata of the dataset such as:

- measurements
- platform and sensor names
- geospatial extents and projection
- acquisition time

**Note:** Some metadata requires cleanup before they are ready to be loaded.

For more information see Dataset Documents.

### 10.6 Adding Data - Indexing

Everything is now ready, and we can use the `datacube` tool to add one or more datasets into our Cube by running:
10.6.1 Sample Earth Observation Data

The U.S. Geological Survey provides many freely available, Analysis Ready, earth observation data products. The following are a good place to start looking.

- **Landsat**
  - USGS Landsat Surface Reflectance - LEDAPS 30m

- **MODIS**
  - MCD43A1 - BRDF-Albedo Model Parameters 16-Day L3 Global 500m
  - MCD43A2 - BRDF-Albedo Quality 16-Day L3 Global 500m
  - MCD43A3 - Albedo 16-Day L3 Global 500m
  - MCD43A4 - Nadir BRDF-Adjusted Reflectance 16-Day L3 Global 500m

Once you have downloaded some data, it will need *metadata preparation* before use in the Data Cube.

10.7 Indexing Data on Amazon(AWS S3)

Cloud storage is a sought after feature for most storage platforms. Options currently exist that allow for a users to store, index, and retrieve data from Amazon S3 buckets using the open data cube. The following sections outline this process.

10.7.1 Configuring AWS CLI Credentials

Install the AWS CLI package and configure it with your Amazon AWS credentials. For a more detailed tutorial on AWS CLI configurations, visit the official AWS docs. The only two fields required to be configured are the *Access Key* and *Secret Access Key*. These keys can be found on your AWS login security page. Try not to lose your *Secret Access Key* as you will not be able to view it again and you will have to request a new one.

```bash
pip install boto3 ruamel.yaml
sudo apt-get install awscli -y
aws configure
```

Add the ca-certificates requisite for S3 indexing and export them to the environment variable the data cube will look for. If you forget this step you will see an error upon attempting to load the indexed dataset.

```bash
sudo apt-get install ca-certificates
export CURL_CA_BUNDLE=/etc/ssl/certs/ca-certificates.crt
```

You may want to add the line `export CURL_CA_BUNDLE=/etc/ssl/certs/ca-certificates.crt` to your `.bashrc` file to make these changes permanent.

10.7.2 Download Indexing Scripts

In order to utilize the convenience of S3 indexing, we must retrieve scripts necessary for S3 indexing. The direct links are provided below since, at the time of this document, they are not all included in the latest release (1.6.1).
Once the necessary scripts have been gathered, it is time to install the AWS CLI package and configure it with your Amazon AWS credentials. The only two fields required to be configured are the Access Key and Secret Access Key. These keys can be found on your AWS login security page. Try not to lose your Secret Access Key as you will not be able to view it again and you will have to request a new one.

```
pip install boto3 ruamel.yaml
sudo apt-get install awscli -y
aws configure
```

Add the ca-certificates requisite for S3 indexing and export them to the environment variable the data cube will look for. If you forget this step you will see an error upon attempting to load the indexed dataset.

```
sudo apt-get install ca-certificates
export CURL_CA_BUNDLE=/etc/ssl/certs/ca-certificates.crt
```

You may want to add the line `export CURL_CA_BUNDLE=/etc/ssl/certs/ca-certificates.crt` to your `.bashrc` file to make these changes permanent.

### 10.7.3 S3 Indexing Example

For this example we will be indexing from Amazon AWS’ `landsat-pds`. This dataset is constantly updated and is free for use. It contains an incredible amount of Landsat 8 data downloaded directly from USGS and hosted on their public S3 bucket. More information can be found here: https://registry.opendata.aws/landsat-8/.

Add a product that matches the metadata for the data found on the S3 bucket. If using a different dataset, you may have to use or create a `yaml` product definition file if an exact match is not readily available.

```
datacube product add ~/Datacube/S3_scripts/ls_usgs.yaml
```

This is an example of indexing an S3 dataset from AWS’ landsat-pds. Notice how `MTL.txt` is the file that is parsed to index the dataset. `-p` is the option for the path of the directory from the landsat-pds main directory. `--suffix` refers to the suffix of the metadata file to process, it will not always be an `MTL.txt` but for landsat-pds, it will be.

```
cd ~/Datacube/S3_scripts
python3 index_from_s3_bucket.py landsat-pds -p c1/L8/139/045/ --suffix="MTL.txt"
```

This is an example that works with the command above to illustrate the Python usage. The `dc.load` would just use bounds defined within the data that was indexed. `output_crs` and `resolution` will be required for this command to work. These commands will need to be entered into a notebook or in a Python console, accessed with the command python

```
import datacube
dc = datacube.Datacube()
```

(continues on next page)
ds = dc.load("ls8_levell_usgs", output_crs="EPSG:4326", resolution=(-30, 30),
latitude=(21, 21.2), longitude=(86.7, 86.9))
Congratulations, you’re ready to ingest some data. If you’ve made it this far you should already have some data indexed, and want to tile it up into a faster storage format or projection system.

11.1 Ingestion Configuration

An Ingestion Configuration file defines a mapping from the way one set of Datasets is stored, into a new storage scheme for the data. This will be recorded in the index as a new Product, and the data will be manipulated and written out to disk in the new format.

An Ingestion Config is written in YAML and contains the following:

- which measurements are stored
- what projection the data is stored in
- what resolution the data is stored in
- how data is tiled
- where the data is stored
- how the data should be resampled and compressed

Multiple ingestion configurations can be kept around to ingest datasets into different projections, resolutions, etc.

11.2 Ingestion Config

An ingestion config is a document which defines the way data should be prepared for high performance access. This can include slicing the data into regular chunks, reprojecting into to the desired projection and compressing the data.

An Ingestion Config is written in YAML and contains the following:

- Source Product name - source_type
• Output Product name - `output_type`

• Output file location and file name template

• Global metadata attributes

• Storage format, specifying:
  – Driver
  – CRS
  – Resolution
  – Tile size
  – Tile Origin

• Details about **measurements**:
  – Output measurement name
  – Source measurement name
  – Resampling method
  – Data type
  – Compression options

**output_type** Name of the output Product. It’s used as a human-readable identifier. Must be unique and consist of alphanumeric characters and/or underscores.

**description (optional)** A human-readable description of the output Product.

**location** Directory to write the output storage units.

**file_path_template** File path pattern defining the name of the storage unit files. **TODO:** list available substitutions

**global_attributes** File level (NetCDF) attributes

**storage**

  **driver** Storage type format. Currently only ‘NetCDF CF’ is supported

  **crs** Definition of the output coordinate reference system for the data to be stored in. May be specified as an EPSG code or WKT.

  **tile_size** Size of the tiles for the data to be stored in specified in projection units. Use **latitude** and **longitude** if the projection is geographic, otherwise use **x** and **y**

  **origin** Coordinates of the bottom-left or top-left corner of the (0,0) tile specified in projection units. If coordinates are for top-left corner, ensure that the **latitude** or **y** dimension of **tile_size** is negative so tile indexes count downward. Use **latitude** and **longitude** if the projection is geographic, otherwise use **x** and **y**

  **resolution** Resolution for the data to be stored in specified in projection units. Negative values flip the axis. Use **latitude** and **longitude** if the projection is geographic, otherwise use **x** and **y**

  **chunking** Size of the internal NetCDF chunks in ‘pixels’.

  **dimension_order** Order of the dimensions for the data to be stored in. Use **latitude** and **longitude** if the projection is geographic, otherwise use **x** and **y**. **TODO:** currently ignored. Is it really needed?

**measurements** Mapping of the input measurement names as specified in the **Dataset Documents** to the per-measurement ingestion parameters

  **dtype** Data type to store the data in. One of (u)int(8,16,32,64), float32, float64
resampling_method  Resampling method. One of nearest, cubic, bilinear, cubic_spline, lanczos, average.

name  Name of the NetCDF variable to store the data in.

nodata (optional)  No data value

### 11.3 Ingest Some Data

A command line tool is used to ingest data

Configuration samples are available as part of the open source Github repository.
Product description document defines some of the metadata common to all the datasets belonging to the products. It also describes the measurements that product has and some of the properties of the measurements.

```json
name: dsmlsv10
description: DSM 1sec Version 1.0
metadata_type: eo

metadata:
  platform:
    code: SRTM
  instrument:
    name: SIR
  product_type: DEM
  format:
    name: ENVI

storage:
  crs: EPSG:4326
  resolution:
    longitude: 0.000277777777780
    latitude: -0.000277777777780

measurements:
  - name: elevation
dtype: float32
  nodata: .nan
  units: 'metre'
```

- **name**  Product name
- **description**  Product description
- **metadata_type**  Name of the Metadata Type Definition
- **metadata**  Dictionary containing bits of metadata common to all the datasets in the product.
It is used during indexing to match datasets to their products.

**storage (optional)** Describes some of common storage attributes of all the datasets. While optional defining this will make product data easier to access and use.

- **crs** Coordinate reference system common to all the datasets in the product. ‘EPSG:<code>’ or WKT string.
- **resolution** Resolution of the data of all the datasets in the product specified in projection units. Use `latitude`, `longitude` if the projection is geographic and `x`, `y` otherwise.

**measurements** List of measurements in this product

- **name** Name of the measurement
- **units** Units of the measurement
- **dtype** Data type. One of `(u)int(8,16,32,64), float32, float64`
- **nodata** No data value

**spectral_definition (optional)** Spectral response of the reflectance measurement.

```plaintext
spectral_definition:
    wavelength: [410, 411, 412]
    response: [0.0261, 0.029, 0.0318]
```

**flags_definition (optional)** Bit flag meanings of the bitset ‘measurement’

```plaintext
flags_definition:
    platform:
        bits: [0,1,2,3]
        description: Platform name
        values:
            0: terra
            1: aqua_terra
            2: aqua
    contiguous:
        bits: 8
        description: All bands for this pixel contain non-null values
        values: {0: false, 1: true}
```
Dataset metadata documents define critical metadata about a dataset including:

- available data measurements
- platform and sensor names
- geospatial extents and projection
- acquisition time
- provenance information

```python
id: a06a2ab-42f7-4e72-bc6d-a47a558b8172
creation_dt: '2016-05-04T09:06:54'
product_type: DEM
platform: {code: SRTM}
instrument: {name: SIR}
format: {name: ENVI}
extent:
  coord:
    ll: {lat: -44.000138890272005, lon: 112.99986111}
    lr: {lat: -44.000138890272005, lon: 153.99986111032797}
    ul: {lat: -10.00013889, lon: 112.99986111}
    ur: {lat: -10.00013889, lon: 153.99986111032797}
  from_dt: '2000-02-11T17:43:00'
  center_dt: '2000-02-21T11:54:00'
  to_dt: '2000-02-22T23:23:00'
grid_spatial:
  projection:
    geo_ref_points:
      ll: {x: 112.99986111, y: -44.000138890272005}
      lr: {x: 153.999861110328, y: -44.000138890272005}
      ul: {x: 112.99986111, y: -10.00013889}
      ur: {x: 153.999861110328, y: -10.00013889}
    spatial_reference: GEOGCS["GCS_WGS_1984", DATUM["WGS_1984", Spheroid["WGS_84", a=6378137.0, e=0.0298257223563]], PRIMEM["Greenwich", 0.0], UNIT["degree", 0.0174532925199433], AUTHORITY["EPSG","4326"]]
```

(continues on next page)
image:
   bands:
      elevation: {path: dsmlsvl_0_Clean.img}
lineage:
   source_datasets: []

id  UUID of the dataset
creation_dt  Creation datetime
product_type, platform/code, instrument/name  Metadata fields supported by default
format  Format the data is stored in. For NetCDF and HDF formats it **must** be ‘NetCDF’ and ‘HDF’
extent  Spatio-temporal extents of the data. Used for search in the database.
grid_spatial/projection
   spatial_reference  Coordinate reference system the data is stored in. ‘EPSG:<code>’ or WKT string.
   geo_ref_points  Spatial extents of the data in the CRS of the data.
   valid_data (optional)  GeoJSON Geometry Object for the ‘data-full’ (non no-data) region of the data. Coordinates are assumed to be in the CRS of the data. Used to avoid loading useless parts of the dataset into memory. Only needs to be roughly correct. Prefer simpler geometry over accuracy.
image/bands  Dictionary of band names to band definitions
   path  Path to the file containing band data. Can be absolute or relative to the folder containing this document.
   layer (optional)  Variable name if format is ‘NetCDF’ or ‘HDF’. Band number otherwise. Default is 1.
lineage  Dataset lineage metadata
   source_datasets  Dictionary of source classifier to dataset documents like this one (yay recursion!).

```
source_datasets:
   level1:
      id: b7d01e8c-1cd2-11e6-b546-a0000100fe80
      product_type: level1
      creation_dt: 2016-05-18 08:09:34
      platform: { code: LANDSAT_5 }
      instrument: { name: TM }
      format: { name: GeoTIFF }
      ...
```

algorithm (optional)  Algorithm used to generate this dataset.
```
algorithm:
   name: brdf
   version: '2.0'
   doi: http://dx.doi.org/10.1109/JSTARS.2010.2042281
   parameters:
      aerosol: 0.078565
```
machine (optional)  Machine and software used to generate this dataset.
```
machine:
   hostname: r2200
   uname: 'Linux r2200 2.6.32-573.22.1.el6.x86_64 #1 SMP Wed Mar 23 03:35:39 ...
         → UTC 2016 x86_64'
```
ancillary (optional)  Additional data used to generate this dataset.

```yaml
ancillary:
  ephemeris:
    name: L52011318DEFEPH.S00
    url: /g/data/v10/ eoancillarydata/sensor-specific/LANDSAT5/
      DefinitiveEphemeris/LS5_YEAR/2011/L52011318DEFEPH.S00
    access_dt: 2016-05-18 18:30:03
    modification_dt: 2011-11-15 02:10:26
    checksum_sha1: f66265314f0c12e005deb356b69721a7031a71374
```

## 13.1 Metadata Type Definition

A Metadata Type defines which fields should be searchable in your product or dataset metadata.

A metadata type is added by default called `eo` with `platform/instrument/lat/lon/time` fields.

You would create a new metadata type if you want custom fields to be searchable for your products, or if you want to structure your metadata documents differently.

You can see the default metadata type in the repository at `datacube/index/default-metadata-types.yaml`.

Or more elaborate examples (with fewer comments) in GA's configuration repository: https://github.com/GeoscienceAustralia/datacube-ingestion

---

13.1. Metadata Type Definition 43
Some data you may want to load into your Data Cube will come pre-packaged with a dataset-description document and is ready to be **indexed/loaded** immediately.

In many other cases the data you want to load into your Data Cube will not have these description documents. Before loading them will need to generate them, using a tool which understands the format the dataset is in. Several of these tools are provided in `utils/` in the source repository.

The two examples below shows USGS landsat data for ingestion into the Data cube:

A specific example for USGS collection 1 MTL format is **USGS Landsat Prepare Script**

## 14.1 1. Preparing USGS Landsat Collection 1 - LEVEL1

Download the USGS Collection 1 landsat scenes from any of the links below:

- Earth-Explorer
- GloVis
- ESPA ordering

The prepare script for collection 1 - level 1 data is provided in `utils/ls_usgs_prepare.py`

```bash
$ python utils/ls_usgs_prepare.py --help
Usage: ls_usgs_prepare.py [OPTIONS] [DATASETS]...

Prepare USGS Landsat Collection 1 data for ingestion into the Data Cube.
This prepare script supports only MTL.txt metadata file
To Set the Path for referring the datasets -
Download the Landsat scene data from Earth Explorer or GloVis into
'some_space_available_folder' and unpack the file.
For example: yoursckrip.py --output [Yaml- which writes datasets into this file]
--for indexing]
[Path for dataset as : /home/some_space_available_folder/]
```

(continues on next page)
Options:
   --output PATH  Write datasets into this file
   --help        Show this message and exit.

$ python utils/ls_usgs_prepare.py --output ls8_usgs_lvl ~/earth_explorer/Collection1/
 → LANDSAT8

ls8_usgs_lvl is the output for required dataset for landsat 8 scene.

To add the product definitions:

For Landsat collection 1 level 1 product:

$ datacube product add docs/config_samples/dataset_types/ls_usgs.yaml
Added "ls8_level1_usgs"
Added "ls7_level1_usgs"
Added "ls5_level1_usgs"
Added "ls8_level1_pc_usgs"

An another example for USGS landsat surface reflectance USGS Landsat LEDAPS

14.2 2. Preparing USGS Landsat Surface Reflectance - LEDAPS

To prepare downloaded USGS LEDAPS Landsat scenes for use with the Data Cube, use the script provided in utils/USGS_precollection_oldscripts/ls_usgs_ard_prepare.py.

The following example generates the required Dataset Metadata files, named agdc-metadata.yaml for three landsat scenes.

$ python utils/USGS_precollection_oldscripts/usgslsprepare.py --help
Usage: usgslsprepare.py [OPTIONS] [DATASETS]...

Prepare USGS LS dataset for ingestion into the Data Cube.

Options:
   --help Show this message and exit.

$ python utils/usgslsprepare.py ~/USGS_LandsatLEDAPS/*
2016-06-09 15:32:51,641 INFO Processing ~/USGS_LandsatLEDAPS/LC80960852015365-→SC20160211222236
2016-06-09 15:32:52,096 INFO Writing ~/USGS_LandsatLEDAPS/LC80960852015365-→SC20160211222236/agdc-metadata.yaml
2016-06-09 15:32:52,119 INFO Processing ~/USGS_LandsatLEDAPS/LE70960852016024-→SC20160211221824
2016-06-09 15:32:52,157 INFO Writing ~/USGS_LandsatLEDAPS/LE70960852016024-→SC20160211221824/agdc-metadata.yaml
2016-06-09 15:32:52,191 INFO Processing ~/USGS_LandsatLEDAPS/LT50960852011290-→SC20160211221617
2016-06-09 15:32:52,230 INFO Writing ~/USGS_LandsatLEDAPS/LT50960852011290-→SC20160211221617/agdc-metadata.yaml

The scenes are now ready to be indexed and accessed using the Data Cube.

For Landsat Surface reflectance LEDAPS add:
Then *index the data.*

### 14.3 3. Prepare script and indexing Landsat data on AWS

Landsat 8 data is available to use on Amazon S3 without needing to worry about the download of all scenes from the start of imagery capture.

Landsat on AWS makes each band of each Landsat scene available as a stand-alone GeoTIFF and the scenes metadata is hosted as a text file.

About the data:

| Source | - | USGS and NASA |
| Category | - | GIS, Sensor Data, Satellite Imagery, Natural Resource |
| Format | - | GeoTIFF, txt, jpg |
| Storage Service | - | Amazon S3 |
| Location | - | s3://landsat-pds in US West (Oregon) Region |
| Update Frequency | - | New Landsat 8 scenes are added regularly as soon as they are available |

Each scene’s directory includes:

- a .TIF GeoTIFF for each of the scenes up to 12 bands (note that the GeoTIFFs include 512x512 internal tiling)
- .TIF.ovr overview file for each .TIF (useful in GDAL based applications)
- a _MTL.txt metadata file
- a small rgb preview jpeg, 3 percent of the original size
- a larger rgb preview jpeg, 15 percent of the original size
- an index.html file that can be viewed in a browser to see the RGB preview and links to the GeoTIFFs and metadata files

#### 14.3.1 Accessing data on AWS

The data are organized using a directory structure based on each scene’s path and row. For instance, the files for Landsat scene LC08_L1TP_139045_20170304_20170316_01_T1 are available in the following location: s3://landsat-pds/c1/L8/139/045/LC08_L1TP_139045_20170304_20170316_01_T1/

> The *c1* refers to Collection 1, the *L8* refers to Landsat 8, *139* refers to the scene’s path, *045* refers to the scene’s row, and the final directory matches the product’s identifier, which uses the following naming convention: LXSS_LLLL_PPPRRR YYYYMMDD_yyymmdd_CC_TX, in which:
L = Landsat
X = Sensor
SS = Satellite
PPP = WRS path
RRR = WRS row
YYYYMMDD = Acquisition date
yyyymmdd = Processing date
CC = Collection number
TX = Collection category

In this case, the scene corresponds to WRS path 139, WRS row 045, and was taken on March 4th, 2017. The full scene list is available here.

The prepare script to index Landsat AWS data ls_public_bucket.py

Instead of downloading all the scenes, the following prepare script helps to directly index the metadata available on S3 using the script utils/ls_public_bucket.py

Usage of the script:

```
$python ls_public_bucket.py --help
Usage: ls_public_bucket.py [OPTIONS] BUCKET_NAME

Enter Bucket name. Optional to enter configuration file to access a different database

Options:
-c, --config PATH          Pass the configuration file to access the database
-p TEXT                    Pass the prefix of the object to the bucket
--help                     Show this message and exit.
```

An example to use the script: ..

```
$python ls_public_bucket.py landsat-pds -p c1/139/045/
```

where landsat-pds is the amazon public bucket name, c1 refers to collection 1 and the numbers after represents the WRS path and row.

Index any path and row by changing the prefix in the above command

14.3.2 Before indexing:

1. You will need an AWS account and configure AWS credentials to access the data on S3 bucket

   For more detailed information refer amazon-docs.

   [default]
   aws_access_key_id = <Access key ID>
   aws_secret_access_key = <Secret access key>

2. Add the product definition to datacube

   Sample product definition for LANDSAT_8 Collection 1 Level1 data is available at docs/config_samples/dataset_types/ls_sample_product.yaml
14.4 Custom Prepare Scripts

We expect that many new Data Cube instances will require custom prepare scripts to be written. It is generally a straightforward task of mapping metadata from one form to another and writing out a YAML document. The code need not even be written in Python, although starting with one of our examples is generally the easiest way.
15.1 datacube-worker

```
datacube-worker [OPTIONS]
```

Options

```
--executor <executor>
    (distributed|dask(alias for distributed)|celery) host:port

--nprocs <nprocs>
    Number of worker processes to launch
```
16.1 Simple Data Cube Replication Tool

This tool provides a very simplistic way to download data and metadata from a remote Data Cube onto a local PC. It connects to a remote Data Cube via SSH, and downloads database records and files.

A configuration file is used to define which portions of which Product should be downloaded. If a Dataset is already available locally, it will not be downloaded again, meaning the tool can be run multiple times to keep the local system up to date with new datasets on the remote server.

It can be run from the command line as `datacube-simple-replica`, taking an optional parameter of a configuration file.

Provide a configuration file in `~/.datacube.replication.conf` in YAML format, or specify an alternate location on the command line.

16.2 Command line documentation

16.3 Caveats and limitations

- Remote datacube files and database are accessed via an SSH host that can be logged into without a password, ie. by using local SSH key agent.
- The remote datacube index must be same version as the local datacube code.
For examples on how to use the API, check out the example Jupyter notebooks

17.1 Datacube Class

Datacube([index, config, app, env,...])  
Interface to search, read and write a datacube.

17.1.1 datacube.Datacube

class datacube.Datacube(index=None, config=None, app=None, env=None, validate_connection=True)  
Interface to search, read and write a datacube.

__init__(index=None, config=None, app=None, env=None, validate_connection=True)  
Create the interface for the query and storage access.

If no index or config is given, the default configuration is used for database connection.

Parameters

- **index**(datacube.index.Index or None) – The database index to use.

- **config**(Union[LocalConfig,str]) – A config object or a path to a config file
  that defines the connection.

  If an index is supplied, config is ignored.

- **app**(str) – A short, alphanumeric name to identify this application.

  The application name is used to track down problems with database queries, so it is
  strongly advised that be used. Required if an index is not supplied, otherwise ignored.
- **env**(str) – Name of the datacube environment to use. ie. the section name in any config files. Defaults to ‘datacube’ for backwards compatibility with old config files.

  Allows you to have multiple datacube instances in one configuration, specified on load, eg. ‘dev’, ‘test’ or ‘landsat’, ‘modis’ etc.

- **validate_connection**(bool) – Should we check that the database connection is available and valid

**Returns** Datacube object

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>([index, config, app, env, ...])</td>
<td>Create the interface for the query and storage access.</td>
</tr>
<tr>
<td>close()</td>
<td>Close any open connections</td>
</tr>
<tr>
<td>create_storage(coords, geobox, measurements)</td>
<td>Create a <code>xarray.Dataset</code> and (optionally) fill it with data.</td>
</tr>
<tr>
<td>find_datasets(<strong>search_terms</strong>)</td>
<td>Search the index and return all datasets for a product matching the search terms.</td>
</tr>
<tr>
<td>find_datasets_lazy([limit, ensure_location])</td>
<td>Find datasets matching query.</td>
</tr>
<tr>
<td>group_datasets(datasets, group_by)</td>
<td>Group datasets along defined non-spatial dimensions (ie.</td>
</tr>
<tr>
<td>list_measurements([show_archived, with_pandas])</td>
<td>List measurements for each product</td>
</tr>
<tr>
<td>list_products([show_archived, with_pandas])</td>
<td>List products in the datacube</td>
</tr>
<tr>
<td>load([product, measurements, output_crs, ...])</td>
<td>Load data as a <code>xarray</code> object.</td>
</tr>
<tr>
<td>load_data(sources, geobox, measurements[, ...])</td>
<td>Load data from <code>group_datasets()</code> into an <code>xarray.Dataset</code>.</td>
</tr>
<tr>
<td>measurement_data(sources, geobox, measurement)</td>
<td>Retrieve a single measurement variable as a <code>xarray.DataArray</code>.</td>
</tr>
</tbody>
</table>

**17.2 Data Discovery**

**Datacube.list_products**

List products in the datacube

**Datacube.list_measurements**

List measurements for each product

**17.2.1 datacube.Datacube.list_products**

**Datacube.list_products**(show_archived=False, with_pandas=True)

List products in the datacube

**Parameters**

- **show_archived** – include products that have been archived.

- **with_pandas** – return the list as a Pandas DataFrame, otherwise as a list of dict.

**Return type** pandas.DataFrame or list(dict)
17.2.2 datacube.Datacube.list_measurements

Datacube.list_measurements(show_archived=False, with_pandas=True)
List measurements for each product

Parameters

- show_archived – include products that have been archived.
- with_pandas – return the list as a Pandas DataFrame, otherwise as a list of dict.

Return type pandas.DataFrame or list(dict)

17.3 Data Loading

Datacube.load
Load data as an xarray object.

17.3.1 datacube.Datacube.load

Datacube.load(product=None, measurements=None, output_crs=None, resolution=None, resampling=None, dask_chunks=None, like=None, fuse_func=None, align=None, datasets=None, **query)
Load data as an xarray object. Each measurement will be a data variable in the xarray.Dataset.

See the xarray documentation for usage of the xarray.Dataset and xarray.DataArray objects.

Product and Measurements A product can be specified using the product name, or by search fields that uniquely describe a single product.

```python
product='ls5_ndvi_albers'
```

See list_products() for the list of products with their names and properties.

A product can also be selected by searching using fields, but must only match one product. For example:

```python
platform='LANDSAT_5',
product_type='ndvi'
```

The measurements argument is a list of measurement names, as listed in list_measurements(). If not provided, all measurements for the product will be returned.

```python
measurements=['red', 'nir', 'swir2']
```

Dimensions Spatial dimensions can specified using the longitude/latitude and x/y fields.

The CRS of this query is assumed to be WGS84/EPGS:4326 unless the crs field is supplied, even if the stored data is in another projection or the output_crs is specified. The dimensions longitude/latitude and x/y can be used interchangeably.

```python
latitude=(-34.5, -35.2), longitude=(148.3, 148.7)
```

or

```python
x=(1516200, 1541300), y=(-3867375, -3867350), crs='EPSG:3577'
```

The time dimension can be specified using a tuple of datetime objects or strings with YYYY-MM-DD hh:mm:ss format. E.g.
For EO-specific datasets that are based around scenes, the time dimension can be reduced to the day level, using solar day to keep scenes together.

group_by='solar_day'

For data that has different values for the scene overlap the requires more complex rules for combining data, such as GA's Pixel Quality dataset, a function can be provided to the merging into a single time slice.

See `datacube.helpers.ga_pq_fuser()` for an example implementation.

Output To reproject or resample the data, supply the `output_crs`, `resolution`, `resampling` and `align` fields.

To reproject data to 25m resolution for EPSG:3577:

```python
dc.load(product='ls5_nbar_albers', x=(148.15, 148.2), y=(-35.15, -35.2),
       time=('1990', '1991'),
       output_crs='EPSG:3577', resolution=(-25, 25), resampling='cubic')
```

Parameters

- **product** (`str`) – the product to be included.
- **measurements** (`list(str)`, optional) – Measurements name or list of names to be included, as listed in `list_measurements()`.

If a list is specified, the measurements will be returned in the order requested. By default all available measurements are included.
- **query** – Search parameters for products and dimension ranges as described above.
- **output_crs** (`str`) – The CRS of the returned data. If no CRS is supplied, the CRS of the stored data is used.
- **resolution** (`(float,float)`) – A tuple of the spatial resolution of the returned data. This includes the direction (as indicated by a positive or negative number).

Typically when using most CRSs, the first number would be negative.
- **resampling** (`str|dict`) – The resampling method to use if re-projection is required.

Valid values are: 'nearest', 'cubic', 'bilinear', 'cubic_spline', 'lanczos', 'average', 'mode', 'gauss', 'max', 'min', 'med', 'q1', 'q3' .. seealso:: `load_data()`
- **align** (`(float,float)`) – Load data such that point ‘align’ lies on the pixel boundary. Units are in the co-ordinate space of the output CRS.

Default is (0,0)
- **dask_chunks** (`dict`) – If the data should be lazily loaded using `dask.array.Array`, specify the chunking size in each output dimension.

See the documentation on using `xarray with dask` for more information.
- **like** (`xarray.Dataset`) – Uses the output of a previous `load()` to form the basis of a request for another product. E.g.:

```python
pq = dc.load(product='ls5_pq_albers', like=nbar_dataset)
```
• **group_by** *(str)* – When specified, perform basic combining/reducing of the data.

• **fuse_func** – Function used to fuse/combine/reduce data with the *group_by* parameter. By default, data is simply copied over the top of each other, in a relatively undefined manner. This function can perform a specific combining step, eg. for combining GA PQ data. This can be a dictionary if different fusers are needed per band.

• **datasets** – Optional. If this is a non-empty list of *datacube.model.Dataset* objects, these will be loaded instead of performing a database lookup.

• **limit** *(int)* – Optional. If provided, limit the maximum number of datasets returned. Useful for testing and debugging.

  Returns  Requested data in a *xarray.Dataset*

  Return type  *xarray.Dataset*

### 17.3.2 Internal Loading Functions

This operations can be useful if you need to customise the loading process, for example, to pre-filter the available datasets before loading.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Datacube.find_datasets(**search_terms)**</code></td>
<td>Search the index and return all datasets for a product matching the search terms.</td>
</tr>
<tr>
<td><code>Datacube.group_datasets(datasets, group_by)</code></td>
<td>Group datasets along defined non-spatial dimensions (ie. time).</td>
</tr>
<tr>
<td><code>Datacube.load_data(sources, geobox, measurements)</code></td>
<td>Load data from <em>group_datasets()</em> into an <em>xarray.Dataset</em>.</td>
</tr>
<tr>
<td><code>Datacube.measurement_data(sources, geobox, ...)</code></td>
<td>Retrieve a single measurement variable as a <em>xarray.DataArray</em>.</td>
</tr>
</tbody>
</table>

**datacube.Datacube.find_datasets**

`Datacube.find_datasets(**search_terms)**`

Search the index and return all datasets for a product matching the search terms.

- **Parameters**  
  - `search_terms` – see *datacube.api.query.Query*

- **Returns**  list of datasets

- **Return type**  list[*datacube.model.Dataset*]

See also:

- `group_datasets()`  
- `load_data()`  
- `find_datasets_lazy()`

**datacube.Datacube.group_datasets**

`static Datacube.group_datasets(datasets, group_by)`

Group datasets along defined non-spatial dimensions (ie. time).

- **Parameters**
  - `datasets` – a list of datasets, typically from `find_datasets()`
  - `group_by (GroupBy)` – Contains: - a function that returns a label for a dataset - name of the new dimension - unit for the new dimension - function to sort by before grouping
Return type  xarray.DataArray

See also:

find_datasets(), load_data(), query_group_by()

datacube.Datacube.load_data

static Datacube.load_data(sources, geobox, measurements, resampling=None, fuse_func=None, dask_chunks=None, skip_broken_datasets=False, **extra)

Load data from group_datasets() into an xarray.Dataset.

Parameters

- **sources** (xarray.DataArray) – DataArray holding a list of datacube.model.Dataset, grouped along the time dimension
- **geobox** (GeoBox) – A GeoBox defining the output spatial projection and resolution
- **measurements** – list of Measurement objects
- **resampling** (str|dict) – The resampling method to use if re-projection is required. This could be a string or a dictionary mapping band name to resampling mode. When using a dict use '*' to indicate “apply to all other bands”, for example {'*': 'cubic', 'fmask': 'nearest'} would use cubic for all bands except fmask for which nearest will be used.
  
  Valid values are: 'nearest', 'cubic', 'bilinear', 'cubic_spline', 'lanczos', 'average', 'mode', 'gauss', 'max', 'min', 'med', 'q1', 'q3'

  Default is to use nearest for all bands.
- **fuse_func** – function to merge successive arrays as an output. Can be a dictionary just like resampling.
- **dask_chunks** (dict) – If provided, the data will be loaded on demand using using dask.array.Array. Should be a dictionary specifying the chunking size for each output dimension.

  See the documentation on using xarray with dask for more information.

Return type  xarray.Dataset

See also:

find_datasets(), group_datasets()

datacube.Datacube.measurement_data

static Datacube.measurement_data(sources, geobox, measurement, fuse_func=None, dask_chunks=None)

Retrieve a single measurement variable as a xarray.DataArray.

See also:

load_data()

Parameters

- **sources** (xarray.DataArray) – DataArray holding a list of datacube.model.Dataset objects
• **geobox** *(GeoBox)* – A GeoBox defining the output spatial projection and resolution
• **measurement** – *Measurement* object
• **fuse_func** – function to merge successive arrays as an output
• **dask_chunks** *(dict)* – If the data should be loaded as needed using *dask.array.Array*, specify the chunk size in each output direction. See the documentation on using *xarray* with *dask* for more information.

**Return type** **xarray.DataArray**

## 17.4 Grid Processing API

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GridWorkflow(index[, grid_spec, product])</code></td>
<td>GridWorkflow deals with cell- and tile-based processing using a grid defining a projection and resolution.</td>
</tr>
<tr>
<td><code>grid_workflow.Tile(sources, geobox)</code></td>
<td>The Tile object holds a lightweight representation of a datacube result.</td>
</tr>
<tr>
<td><code>GridWorkflow.list_cells([cell_index])</code></td>
<td>List cells that match the query.</td>
</tr>
<tr>
<td><code>GridWorkflow.list_tiles([cell_index])</code></td>
<td>List tiles of data, sorted by cell.</td>
</tr>
<tr>
<td><code>GridWorkflow.load(tile[, measurements, ...])</code></td>
<td>Load data for a cell/tile.</td>
</tr>
</tbody>
</table>

### 17.4.1 datacube.api.GridWorkflow

**class** **datacube.api.GridWorkflow**(index, grid_spec=None, product=None)  
GridWorkflow deals with cell- and tile-based processing using a grid defining a projection and resolution.

Use GridWorkflow to specify your desired output grid. The methods *list_cells()* and *list_tiles()* query the index and return a dictionary of cell or tile keys, each mapping to a Tile object.

The Tile object can then be used to load the data without needing the index, and can be serialized for use with the *distributed* package.

**__init__**(index, grid_spec=None, product=None)  
Create a grid workflow tool.

Either grid_spec or product must be supplied.

**Parameters**

- **index** *(datacube.index.Index)* – The database index to use.
- **grid_spec** *(GridSpec)* – The grid projection and resolution
- **product** *(str)* – The name of an existing product, if no grid_spec is supplied.

**Methods**

- **__init__**(index[, grid_spec, product])  
Create a grid workflow tool.

- **cell_observations**([cell_index, geopolygon, ...])  
List datasets, grouped by cell.

- **cell_sources**(observations, group_by)  

- **group_into_cells**(observations, group_by)  
Group observations into a stack of source tiles.

- **list_cells**([cell_index])  
List cells that match the query.

Continued on next page
Table 7 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list_tiles((cell_index))</td>
<td>List tiles of data, sorted by cell.</td>
</tr>
<tr>
<td>load(tile[, measurements, dask_chunks, ...])</td>
<td>Load data for a cell/tile.</td>
</tr>
<tr>
<td>tile_sources(observations, group_by)</td>
<td>Split observations into tiles and group into source tiles.</td>
</tr>
<tr>
<td>update_tile_lineage(tile)</td>
<td></td>
</tr>
</tbody>
</table>

### 17.4.2 datacube.api.grid_workflow.Tile

**class** datacube.api.grid_workflow.Tile*(sources, geobox)*

The Tile object holds a lightweight representation of a datacube result.

It is produced by GridWorkflow.list_cells() or GridWorkflow.list_tiles().

The Tile object can be passed to GridWorkflow.load() to be loaded into memory as an xarray.Dataset.

A portion of a tile can be created by using index notation. eg:

```
tile[0:1, 0:1000, 0:1000]
```

This can be used to load small portions of data into memory, instead of having to access the entire Tile at once.

**__init__**(sources, geobox)

Create a Tile representing a dataset that can be loaded.

**Parameters**

- **sources** *(xarray.DataArray)* – An array of non-spatial dimensions of the request, holding lists of datacube.storage.DatasetSource objects.
- **geobox** *(model.GeoBox)* – The spatial footprint of the Tile

**Methods**

- **__init__**(sources, geobox)

Create a Tile representing a dataset that can be loaded.

- **split**(dim[, step])

Splits along a non-spatial dimension into Tile objects with a length of 1 or more in the dim dimension.

- **split_by_time**([freq, time_dim])

Splits along the time dimension, into periods, using pandas offsets, such as:

**Attributes**

- **dims**

Names of the dimensions, eg ("time", "y", "x") :return: tuple(str)

- **product**

datacube.model.DatasetType

- **shape**

Lengths of each dimension, eg (285, 4000, 4000) :return: tuple(int)

### 17.4.3 datacube.api.GridWorkflow.list_cells

GridWorkflow.list_cells*(cell_index=None, **query)*

List cells that match the query.
Returns a dictionary of cell indexes to **Tile** objects.

Cells are included if they contain any datasets that match the query using the same format as `datacube.Datacube.load()`.

E.g.:

```python
gw.list_cells(product='ls5_nbar_albers',
               time=('2001-1-1 00:00:00', '2001-3-31 23:59:59'))
```

**Parameters**

- **cell_index** *(int, int)* – The cell index. E.g. (14, -40)
- **query** – see `datacube.api.query.Query`

**Return type** dict[(int, int), Tile]

### 17.4.4 `datacube.api.GridWorkflow.list_tiles`

`GridWorkflow.list_tiles(cell_index=None, **query)`

List tiles of data, sorted by cell.

```python
tiles = gw.list_tiles(product='ls5_nbar_albers',
                      time=('2001-1-1 00:00:00', '2001-3-31 23:59:59'))
```

The values can be passed to `load()`

**Parameters**

- **cell_index** *(int, int)* – The cell index (optional). E.g. (14, -40)
- **query** – see `datacube.api.query.Query`

**Return type** dict[(int, int, numpy.datetime64), Tile]

See also:

`load()`

### 17.4.5 `datacube.api.GridWorkflow.load`

`static GridWorkflow.load(tile, measurements=None, dask_chunks=None, fuse_func=None, resampling=None, skip_broken_datasets=False)`

Load data for a cell/tile.

The data to be loaded is defined by the output of `list_tiles()`.

This is a static function and does not use the index. This can be useful when running as a worker in a distributed environment and you wish to minimize database connections.

See the documentation on using `xarray with dask` for more information.

**Parameters**

- **tile** *(Tile)* – The tile to load.
- **measurements** *(list (str))* – The names of measurements to load
• **dask_chunks** *(dict)* – If the data should be loaded as needed using *dask.array.Array*, specify the chunk size in each output direction.

   See the documentation on using *xarray with dask* for more information.

• **fuse_func** – Function to fuse together a tile that has been pre-grouped by calling *list_cells()* with a *group_by* parameter.

• **resampling** *(str|dict)* – The resampling method to use if re-projection is required, could be configured per band using a dictionary *(meth: load_data)*

   Valid values are: 'nearest', 'cubic', 'bilinear', 'cubic_spline', 'lanczos', 'average'

   Defaults to 'nearest'.

• **skip_broken_datasets** *(bool)* – If True, ignore broken datasets and continue processing with the data that can be loaded. If False, an exception will be raised on a broken dataset. Defaults to False.

   **Return type** *xarray.Dataset*

   **See also:**

   * list_tiles()* *list_cells()*

### 17.4.6 Grid Processing API Internals

```python
datacube.api.GridWorkflow.cell_observations
```

**GridWorkflow.cell_observations** *(cell_index*, List datasets, grouped by cell.

...*)

```python
GridWorkflow.group_into_cells
```

**GridWorkflow.group_into_cells** *(observations, Group observations into a stack of source tiles.

...*)

```python
GridWorkflow.tile_sources
```

**GridWorkflow.tile_sources** *(observations, Split observations into tiles and group into source tiles

  group_by)*

**datacube.api.GridWorkflow.cell_observations**

GridWorkflow.cell_observations *(cell_index=None, geopolygon=None, tile_buffer=None, **indexers)*

**List datasets, grouped by cell.**

**Parameters**

• **geopolygon** *(datacube.utils.Geometry)* – Only return observations with data inside polygon.

• **tile_buffer** *(float, float)* – buffer tiles by (y, x) in CRS units

• **cell_index** *(int, int)* – The cell index. E.g. (14, -40)

• **indexers** – Query to match the datasets, see *datacube.api.query.Query*

**Returns** Datsets grouped by cell index

**Return type** *dict[[int,int], list[datacube.model.Dataset]]*

**See also:**

* datacube.Datacube.find_datasets()
* datacube.api.query.Query
datacube.api.GridWorkflow.group_into_cells

**static** GridWorkflow.group_into_cells(observations, group_by)

Group observations into a stack of source tiles.

**Parameters**

- **observations** – datasets grouped by cell index, like from `cell_observations()`
- **group_by** (datacube.api.query.GroupBy) – grouping method, as returned by `datacube.api.query.query_group_by()`

**Returns** tiles grouped by cell index

**Return type** `dict[(int, int), Tile]`

See also:

- `load()`
- `datacube.Datacube.group_datasets()`

datacube.api.GridWorkflow.tile_sources

**static** GridWorkflow.tile_sources(observations, group_by)

Split observations into tiles and group into source tiles

**Parameters**

- **observations** – datasets grouped by cell index, like from `cell_observations()`
- **group_by** (datacube.api.query.GroupBy) – grouping method, as returned by `datacube.api.query.query_group_by()`

**Returns** tiles grouped by cell index and time

**Return type** `dict[tuple(int, int, numpy.datetime64), Tile]`

See also:

- `load()`
- `datacube.Datacube.group_datasets()`

### 17.5 Internal Data Model

<table>
<thead>
<tr>
<th>Dataset</th>
<th>A Dataset.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurement(**kwargs)</td>
<td>Describes a single data variable of a Product or Dataset.</td>
</tr>
<tr>
<td>MetadataType(definition, Any[, ...])</td>
<td>Metadata Type definition</td>
</tr>
<tr>
<td>DatasetType(metadata_type, definition, Any[, id])</td>
<td>Product definition</td>
</tr>
<tr>
<td>GridSpec(crs, tile_size, float[, resolution[, ...]])</td>
<td>Definition for a regular spatial grid</td>
</tr>
<tr>
<td>Range(begin, end)</td>
<td></td>
</tr>
<tr>
<td>CellIndex</td>
<td></td>
</tr>
</tbody>
</table>
17.5.1 `datacube.model.Dataset`


A Dataset. A container of metadata, and refers typically to a multi-band raster on disk.

Most important parts are the metadata_doc and uri.

Parameters

- `metadata_doc` – the document (typically a parsed json/yaml)
- `uris` – All active uris for the dataset

`__init__`(type_, metadata_doc, local_uri, ...)

Initialize self. See `help(type(self))` for accurate signature.

Methods

`__init__`(type_, metadata_doc, local_uri, ...)

Initialize self.

`metadata_doc_without_lineage()`

Return metadata document without nested lineage datasets

Attributes

- `bounds` – bounding box of the dataset in the native crs
- `center_time` – mid-point of time range
- `crs` – Return CRS if available
- `extent` – valid extent of the dataset or None
- `format` – UUID of a dataset
- `id` – Is this dataset active?
- `is_archived` – Is this dataset archived?
- `key_time` – datetime.datetime
- `local_path` – A path to this dataset on the local filesystem (if available).
- `local_uri` – The latest local file uri, if any.
- `managed` –
- `measurements` –
- `metadata` –
- `metadata_type` –
- `time` –
- `transform` –
- `uri_scheme` –
17.5.2 **datacube.model.Measurement**

class **datacube.model.Measurement** (**kwargs**)

Describes a single data variable of a Product or Dataset.

Must include, which can be used when loading and interpreting data:

- name
- dtype - eg: int8, int16, float32
- nodata - What value represent No Data
- units

Attributes can be accessed using `dict []` syntax.

Can also include attributes like alternative names ‘aliases’, and spectral and bit flags definitions to aid with interpreting the data.

**__init__**( **kwargs**)

Initialize self. See `help(type(self))` for accurate signature.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong><strong>(kwargs)</strong></td>
<td>Initialize self. See <code>help(type(self))</code> for accurate signature.</td>
</tr>
<tr>
<td>clear()</td>
<td></td>
</tr>
<tr>
<td>copy()</td>
<td>Required as the super class <code>dict</code> method returns a <code>dict</code> and does not preserve <code>Measurement</code> class</td>
</tr>
<tr>
<td>dataarray_attrs()</td>
<td>This returns attributes filtered for display in a dataarray.</td>
</tr>
<tr>
<td>fromkeys()</td>
<td>Create a new dictionary with keys from iterable and values set to value.</td>
</tr>
<tr>
<td>get</td>
<td>Return the value for key if key is in the dictionary, else default.</td>
</tr>
<tr>
<td>items()</td>
<td></td>
</tr>
<tr>
<td>keys()</td>
<td></td>
</tr>
<tr>
<td>pop(key, default)</td>
<td>If key is not found, default is returned if given, otherwise <code>KeyError</code> is raised</td>
</tr>
<tr>
<td>popitem()</td>
<td>2-tuple; raise <code>KeyError</code> if D is empty.</td>
</tr>
<tr>
<td>setdefault()</td>
<td>Insert key with a value of default if key is not in the dictionary.</td>
</tr>
<tr>
<td>update([E, ]<strong>kwargs</strong>)</td>
<td>If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]</td>
</tr>
<tr>
<td>values()</td>
<td></td>
</tr>
</tbody>
</table>

### Attributes

- **ATTR_BLACKLIST**
- **OPTIONAL_KEYS**
- **REQUIRED_KEYS**
17.5.3 datacube.model.MetadataType

class datacube.model.MetadataType(definition: Mapping[str, Any], dataset_search_fields: Mapping[str, datacube.model.fields.Field], id_: Optional[int] = None)

    Metadata Type definition

    __init__(definition: Mapping[str, Any], dataset_search_fields: Mapping[str, datacube.model.fields.Field], id_: Optional[int] = None)
    Initialize self. See help(type(self)) for accurate signature.

    Methods

    __init__(definition, Any, ...) Initialize self.
dataset_reader(dataset_doc, ...)
validate(document)

    Attributes

    description
ame
    schema

17.5.4 datacube.model.DatasetType

class datacube.model.DatasetType(metadata_type: datacube.model.MetadataType, definition: Mapping[str, Any], id_: Optional[int] = None)

    Product definition

    Parameters

    • metadata_type (MetadataType)
    • definition (dict)

    __init__(metadata_type: datacube.model.MetadataType, definition: Mapping[str, Any], id_: Optional[int] = None)
    Initialize self. See help(type(self)) for accurate signature.

    Methods

    __init__(metadata_type, definition, Any, id_) Initialize self.
canonical_measurement(measurement) resolve measurement alias into canonical name
dataset_reader(dataset_doc)
lookup_measurements(measurements) Find measurements by name
to_dict() Convert to a dictionary representation of the available fields
validate(document)

    Attributes
### 17.5.5 datacube.model.GridSpec

**class** datacube.model.GridSpec(  
crs: datacube.utils.geometry._base.CRS, tile_size: Tuple[float, float], resolution: Tuple[float, float], origin: Optional[Tuple[float, float]] = None)  

Definition for a regular spatial grid

```python  
>>> gs = GridSpec(crs=geometry.CRS('EPSG:4326'), tile_size=(1, 1), resolution=(-0.1, 0.1), origin=(-50.05, 139.95))  
>>> gs.tile_resolution  
(10, 10)  
>>> list(gs.tiles(geometry.BoundingBox(140, -50, 141.5, -48.5)))  
[((0, 0), GeoBox(10, 10, Affine(0.1, 0.0, 139.95, 0.0, -0.1, -49.05), EPSG:4326)), ((1, 0), GeoBox(10, 10, Affine(0.1, 0.0, 140.95, 0.0, -0.1, -49.05), EPSG:4326)), ((0, 1), GeoBox(10, 10, Affine(0.1, 0.0, 139.95, 0.0, -0.1, -48.05), EPSG:4326)), ((1, 1), GeoBox(10, 10, Affine(0.1, 0.0, 140.95, 0.0, -0.1, -48.05), EPSG:4326))]
```

**Parameters**

- **crs** *(geometry.CRS)* – Coordinate System used to define the grid
- **tile_size** *(float, float)* – (Y, X) size of each tile, in CRS units
- **resolution** *(float, float)* – (Y, X) size of each data point in the grid, in CRS units. Y will usually be negative.
- **origin** *(float, float)* – (Y, X) coordinates of a corner of the (0,0) tile in CRS units. default is (0.0, 0.0)

**__init__**(crs, tile_size, float], resolution, ...) Initialize self. See help(type(self)) for accurate signature.

**Methods**

- **__init__**(crs, tile_size, float], resolution, ...)* Initialize self.
- **grid_range**(lower, upper, step) Returns the indices along a 1D scale.
- **tile_coords**(tile_index, int) Tile coordinates in (Y,X) order
- **tile_geobox**(tile_index, int) Tile geobox.

Continued on next page
Table 20 – continued from previous page

<table>
<thead>
<tr>
<th>tiles(bounds, geobox_cache)</th>
<th>Returns an iterator of tile_index, GeoBox tuples across the grid and overlapping with the specified bounds rectangle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>tiles_from_geopolygon(geopolygon,...)</td>
<td>Returns an iterator of tile_index, GeoBox tuples across the grid and overlapping with the specified geopolygon.</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>alignment</th>
<th>Pixel boundary alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>dimensions</td>
<td>List of dimension names of the grid spec</td>
</tr>
<tr>
<td>tile_resolution</td>
<td>Tile size in pixels in CRS dimension order (Usually y,x or lat,lon)</td>
</tr>
</tbody>
</table>

**17.5.6 datacube.model.Range**

```python
class datacube.model.Range(begin, end)
```

```python
__init__()
    Initialize self. See help(type(self)) for accurate signature.
```

**Methods**

<table>
<thead>
<tr>
<th>count</th>
<th>Return number of occurrences of value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>Return first index of value.</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>begin</th>
<th>Alias for field number 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>end</td>
<td>Alias for field number 1</td>
</tr>
</tbody>
</table>

**17.6 Database Index API**

**17.6.1 Dataset Querying**

When connected to an ODC Database, these methods are available for searching and querying:

```python
dc = Datacube()
dc.index.datasets.{method}
```

- **get**
  Get dataset by id

- **search**
  Perform a search, returning results as Dataset objects.

- **search_by_metadata**
  Perform a search using arbitrary metadata, returning results as Dataset objects.

Continued on next page
Table 24 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>search_by_product</td>
<td>Perform a search, returning datasets grouped by product type.</td>
</tr>
<tr>
<td>search_eager</td>
<td>Perform a search, returning results as Dataset objects.</td>
</tr>
<tr>
<td>search_product_duplicates</td>
<td>Find dataset ids who have duplicates of the given set of field names.</td>
</tr>
<tr>
<td>search_returning</td>
<td>Perform a search, returning only the specified fields.</td>
</tr>
<tr>
<td>search_summaries</td>
<td>Perform a search, returning just the search fields of each dataset.</td>
</tr>
<tr>
<td>has</td>
<td>Have we already indexed this dataset?</td>
</tr>
<tr>
<td>bulk_has</td>
<td>Like has but operates on a list of ids.</td>
</tr>
<tr>
<td>can_update</td>
<td>Check if dataset can be updated.</td>
</tr>
<tr>
<td>count</td>
<td>Perform a search, returning count of results.</td>
</tr>
<tr>
<td>count_by_product</td>
<td>Perform a search, returning a count of for each matching product type.</td>
</tr>
<tr>
<td>count_by_product_through_time</td>
<td>Perform a search, returning counts for each product grouped in time slices of the given period.</td>
</tr>
<tr>
<td>count_product_through_time</td>
<td>Perform a search, returning counts for a single product grouped in time slices of the given period.</td>
</tr>
<tr>
<td>get_derived</td>
<td>Get all derived datasets</td>
</tr>
<tr>
<td>get_field_names</td>
<td>Get the list of possible search fields for a Product</td>
</tr>
<tr>
<td>get_locations</td>
<td>Get the list of storage locations for the given dataset id.</td>
</tr>
<tr>
<td>get_archived_locations</td>
<td>Find locations which have been archived for a dataset</td>
</tr>
<tr>
<td>get_datasets_for_location</td>
<td>Find datasets that exist at the given URI</td>
</tr>
</tbody>
</table>

**datacube.index._datasets.DatasetResource.get**

DatasetResource.get(id_, include_sources=False)

Get dataset by id

Parameters

- id (UUID) – id of the dataset to retrieve
- include_sources (bool) – get the full provenance graph?

Return type Dataset

**datacube.index._datasets.DatasetResource.search**

DatasetResource.search(limit=None, **query)

Perform a search, returning results as Dataset objects.

Parameters

- query (Union[str, float, Range, list]) –
- limit (int) – Limit number of datasets

Return type __generator[Dataset]

**datacube.index._datasets.DatasetResource.search_by_metadata**

DatasetResource.search_by_metadata(metadata)

Perform a search using arbitrary metadata, returning results as Dataset objects.
Caution – slow! This will usually not use indexes.

Parameters metadata (dict) –
Return type list[Dataset]

datacube.index._datasets.DatasetResource.search_by_product

DatasetResource.search_by_product(**query)
Perform a search, returning datasets grouped by product type.

Parameters query (dict [str, str|float|datacube.model.Range]) –
Return type __generator[[DatasetType, __generator[Dataset]]]

datacube.index._datasets.DatasetResource.search_eager

DatasetResource.search_eager(**query)
Perform a search, returning results as Dataset objects.

Parameters query (dict [str, str|float|datacube.model.Range]) –
Return type list[Dataset]

datacube.index._datasets.DatasetResource.search_product_duplicates

Find dataset ids who have duplicates of the given set of field names.
Product is always inserted as the first grouping field.
Returns each set of those field values and the datasets that have them.

datacube.index._datasets.DatasetResource.search_returning

DatasetResource.search_returning(field_names, limit=None, **query)
Perform a search, returning only the specified fields.
This method can be faster than normal search() if you don’t need all fields of each dataset.
It also allows for returning rows other than datasets, such as a row per uri when requesting field ‘uri’.

Parameters

- field_names (tuple[str]) –
- query (Union[str, float, Range, list]) –
- limit (int) – Limit number of datasets

Returns __generator[tuple] sequence of results, each result is a namedtuple of your requested fields
**datacube.index._datasets.DatasetResource.search_summaries**

DatasetResource.search_summaries(**query**)

Perform a search, returning just the search fields of each dataset.

Parameters 

- query (dict[str, str|float|datacube.model.Range]) –

Return type __generator[dict]

**datacube.index._datasets.DatasetResource.has**

DatasetResource.has(id_)

Have we already indexed this dataset?

Parameters

- id (typing.Union[UUID,]) – dataset id

Return type bool

**datacube.index._datasets.DatasetResource.bulk_has**

DatasetResource.bulk_has(ids_)

Like has but operates on a list of ids.

For every supplied id check if database contains a dataset with that id.

Parameters

- ids ([typing.Union[UUID,]) – list of dataset ids

Return type [bool]

**datacube.index._datasets.DatasetResource.can_update**

DatasetResource.can_update(dataset, updates_allowed=None)

Check if dataset can be updated. Return bool,safe_changes,unsafe_changes

Parameters

- dataset (Dataset) – Dataset to update
- updates_allowed (dict) – Allowed updates

Return type bool,list[change],list[change]

**datacube.index._datasets.DatasetResource.count**

DatasetResource.count(**query**)

Perform a search, returning count of results.

Parameters 

- query (dict[str, str|float|datacube.model.Range]) –

Return type int

**datacube.index._datasets.DatasetResource.count_by_product**

DatasetResource.count_by_product(**query**)

Perform a search, returning a count of for each matching product type.

Parameters 

- query (dict[str, str|float|datacube.model.Range]) –
Returns Sequence of (product, count)
Return type __generator[((DatasetType, int))]

datacube.index._datasets.DatasetResource.count_by_product_through_time

DatasetResource.count_by_product_through_time(period, **query)
Perform a search, returning counts for each product grouped in time slices of the given period.

Parameters
• query (dict[str, str|float|datacube.model.Range]) – 
• period (str) – Time range for each slice: ‘1 month’, ‘1 day’ etc.

Returns For each matching product type, a list of time ranges and their count.
Return type __generator[((DatasetType, list[(datetime.datetime, datetime.datetime), int))]

datacube.index._datasets.DatasetResource.count_product_through_time

DatasetResource.count_product_through_time(period, **query)
Perform a search, returning counts for a single product grouped in time slices of the given period.

Will raise an error if the search terms match more than one product.

Parameters
• query (dict[str, str|float|datacube.model.Range]) – 
• period (str) – Time range for each slice: ‘1 month’, ‘1 day’ etc.

Returns For each matching product type, a list of time ranges and their count.
Return type list[(str, list[(datetime.datetime, datetime.datetime), int)])

datacube.index._datasets.DatasetResource.get_derived

DatasetResource.get_derived(id_)
Get all derived datasets

Parameters id (UUID) – dataset id
Return type list[Dataset]

datacube.index._datasets.DatasetResource.get_field_names

DatasetResource.get_field_names(product_name=None)
Get the list of possible search fields for a Product

Parameters product_name (str) –
Return type set[str]
**datacube.index._datasets.DatasetResource.get_locations**

DatasetResource.get_locations(id_)
Get the list of storage locations for the given dataset id

Parameters
- id (typing.Union[UUID, str]) – dataset id

Return type
- list[str]

**datacube.index._datasets.DatasetResource.get_archived_locations**

DatasetResource.get_archived_locations(id_)
Find locations which have been archived for a dataset

Parameters
- id (typing.Union[UUID, str]) – dataset id

Return type
- list[str]

**datacube.index._datasets.DatasetResource.get_datasets_for_location**

DatasetResource.get_datasets_for_location(uri, mode=None)
Find datasets that exist at the given URI

Parameters
- uri – search uri
- mode (str) – ‘exact’ or ‘prefix’

Returns

### 17.6.2 Dataset Writing

When connected to an ODC Database, these methods are available for adding, updating and archiving datasets:

```python
dc = Datacube()
dc.index.datasets.{method}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Add dataset to the index.</td>
</tr>
<tr>
<td>add_location</td>
<td>Add a location to the dataset if it doesn’t already exist.</td>
</tr>
<tr>
<td>archive</td>
<td>Mark datasets as archived</td>
</tr>
<tr>
<td>archive_location</td>
<td>Archive a location of the dataset if it exists.</td>
</tr>
<tr>
<td>remove_location</td>
<td>Remove a location from the dataset if it exists.</td>
</tr>
<tr>
<td>restore</td>
<td>Mark datasets as not archived</td>
</tr>
<tr>
<td>restore_location</td>
<td>Un-archive a location of the dataset if it exists.</td>
</tr>
<tr>
<td>update</td>
<td>Update dataset metadata and location :param Dataset dataset: Dataset to update :param updates_allowed: Allowed updates :rtype: Dataset</td>
</tr>
</tbody>
</table>

**datacube.index._datasets.DatasetResource.add**

DatasetResource.add(dataset, with_lineage=None, **kwargs)
Add dataset to the index. No-op if it is already present.
Parameters

- **dataset** (*Dataset*) – dataset to add
- **with_lineage** (*bool*) – True – attempt adding lineage if it’s missing, False don’t

Return type *Dataset*

datacube.index._datasets.DatasetResource.add_location

DatasetResource.add_location(id_, uri)
Add a location to the dataset if it doesn’t already exist.

Parameters

- **str** id (*typing.Union[UUID]*) – dataset id
- **uri** (*str*) – fully qualified uri

Returns *bool* Was one added?

datacube.index._datasets.DatasetResource.archive

DatasetResource.archive(ids)
Mark datasets as archived

Parameters **ids** (*list[UUID]*) – list of dataset ids to archive

datacube.index._datasets.DatasetResource.archive_location

DatasetResource.archive_location(id_, uri)
Archive a location of the dataset if it exists.

Parameters

- **str** id (*typing.Union[UUID]*) – dataset id
- **uri** (*str*) – fully qualified uri

Return *bool* location was able to be archived

datacube.index._datasets.DatasetResource.remove_location

DatasetResource.remove_location(id_, uri)
Remove a location from the dataset if it exists.

Parameters

- **str** id (*typing.Union[UUID]*) – dataset id
- **uri** (*str*) – fully qualified uri

Returns *bool* Was one removed?
**datacube.index._datasets.DatasetResource.restore**

DatasetResource.restore(ids)
Mark datasets as not archived

**Parameters**

- **ids** (*list*[UUID]) – list of dataset ids to restore

**datacube.index._datasets.DatasetResource.restore_location**

DatasetResource.restore_location(id, uri)
Un-archive a location of the dataset if it exists.

**Parameters**

- **str** id (*typing.Union*[UUID,]) – dataset id
- **uri** (*str*) – fully qualified uri

**Return** bool location was able to be restored

**datacube.index._datasets.DatasetResource.update**

DatasetResource.update(dataset, updates_allowed=None)
Update dataset metadata and location

**Parameters**

- **Dataset** dataset: Dataset to update
- **updates_allowed**: Allowed updates

**rtype**: Dataset

### 17.6.3 Product Querying

When connected to an ODC Database, these methods are available for discovering information about Products:

```python
dc = Datacube()
dc.index.products.{method}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>from_doc</td>
<td>Create a Product from its definitions</td>
</tr>
<tr>
<td>add</td>
<td>Add a Product.</td>
</tr>
<tr>
<td>can_update</td>
<td>Check if product can be updated.</td>
</tr>
<tr>
<td>add_document</td>
<td>Add a Product using its definition</td>
</tr>
<tr>
<td>get</td>
<td>Retrieve Product by id</td>
</tr>
<tr>
<td>get_by_name</td>
<td>Retrieve Product by name</td>
</tr>
<tr>
<td>get_unsafe</td>
<td></td>
</tr>
<tr>
<td>get_by_name_unsafe</td>
<td></td>
</tr>
<tr>
<td>get_with_fields</td>
<td>Return dataset types that have all the given fields.</td>
</tr>
<tr>
<td>search</td>
<td>Return dataset types that have all the given fields.</td>
</tr>
<tr>
<td>search_robust</td>
<td>Return dataset types that match match-able fields and dict of remaining un-matchable fields.</td>
</tr>
<tr>
<td>get_all</td>
<td>Retrieve all Products</td>
</tr>
</tbody>
</table>

**datacube.index._products.ProductResource.from_doc**

ProductResource.from_doc(definition)
Create a Product from its definitions

**Parameters**

- **definition** (*dict*) – product definition document

17.6. Database Index API
Return type  

```
```

datacube.index._products.ProductResource.add

ProductResource.add(product, allow_table_lock=False)

Add a Product.

Parameters

- **allow_table_lock** – Allow an exclusive lock to be taken on the table while creating the indexes. This will halt other user’s requests until completed.
  
If false, creation will be slightly slower and cannot be done in a transaction.

- **product** (DatasetType) – Product to add

Return type  

```
```

datacube.index._products.ProductResource.can_update

ProductResource.can_update(product, allow_unsafe_updates=False)

Check if product can be updated. Return bool, safe_changes, unsafe_changes

(An unsafe change is anything that may potentially make the product incompatible with existing datasets of that type)

Parameters

- **product** (DatasetType) – Product to update

- **allow_unsafe_updates** (bool) – Allow unsafe changes. Use with caution.

Return type  bool,list[change],list[change]

```
```

datacube.index._products.ProductResource.add_document

ProductResource.add_document(definition)

Add a Product using its definition

Parameters definition (dict) – product definition document

Return type  DatasetType

```
```

datacube.index._products.ProductResource.get

ProductResource.get(id_)

Retrieve Product by id

Parameters id (int) – id of the Product

Return type  DatasetType

```
```

datacube.index._products.ProductResource.get_by_name

ProductResource.get_by_name(name)

Retrieve Product by name

Parameters name (str) – name of the Product

```
```
Return type `DatasetType`

datacube.index._products.ProductResource.get_unsafe

ProductResource.get_unsafe(id_)

datacube.index._products.ProductResource.get_by_name_unsafe

ProductResource.get_by_name_unsafe(name)

datacube.index._products.ProductResource.get_with_fields

ProductResource.get_with_fields(field_names)

Return dataset types that have all the given fields.

Parameters

- `field_names` (tuple[str])

Return type `__generator[DatasetType]`

datacube.index._products.ProductResource.search

ProductResource.search(**query)

Return dataset types that have all the given fields.

Parameters

- `query` (dict)

Return type `__generator[DatasetType]`

datacube.index._products.ProductResource.search_robust

ProductResource.search_robust(**query)

Return dataset types that match match-able fields and dict of remaining un-matchable fields.

Parameters

- `query` (dict)

Return type `__generator[(DatasetType, dict)]`

datacube.index._products.ProductResource.get_all

ProductResource.get_all() → Iterable[datacube.model.DatasetType]

Retrieve all Products

17.6.4 Product Addition/Modification

When connected to an ODC Database, these methods are available for discovering information about Products:

```python
dc = Datacube()
dc.index.products.{method}
```

Create a Product from its definitions

Continued on next page
Table 27 – continued from previous page

| add         | Add a Product. |
| update     | Update a product. |
| update_document | Update a Product using its definition |
| add_document | Add a Product using its definition |

**datacube.index._products.ProductResource.update**

`ProductResource.update(product, allow_unsafe_updates=False, allow_table_lock=False)`

Update a product. Unsafe changes will throw a ValueError by default.

(An unsafe change is anything that may potentially make the product incompatible with existing datasets of that type)

**Parameters**

- **product** (*DatasetType*) – Product to update
- **allow_unsafe_updates** (*bool*) – Allow unsafe changes. Use with caution.
- **allow_table_lock** – Allow an exclusive lock to be taken on the table while creating the indexes. This will halt other user’s requests until completed.

If false, creation will be slower and cannot be done in a transaction.

**Return type** *DatasetType*

**datacube.index._products.ProductResource.update_document**

`ProductResource.update_document(definition, allow_unsafe_updates=False, allow_table_lock=False)`

Update a Product using its definition

**Parameters**

- **allow_unsafe_updates** (*bool*) – Allow unsafe changes. Use with caution.
- **definition** (*dict*) – product definition document
- **allow_table_lock** – Allow an exclusive lock to be taken on the table while creating the indexes. This will halt other user’s requests until completed.

If false, creation will be slower and cannot be done in a transaction.

**Return type** *DatasetType*

### 17.6.5 Database Index Connections

**index.index_connect**

Create a Data Cube Index that can connect to a PostgreSQL server

**index.Index**

Access to the datacube index.

**datacube.index.index_connect**

`datacube.index.index_connect(local_config=None, application_name=None, validate_connection=True)`

Create a Data Cube Index that can connect to a PostgreSQL server
It contains all the required connection parameters, but doesn’t actually check that the server is available.

**Parameters**

- **application_name** – A short, alphanumeric name to identify this application.
- **local_config** *(datacube.config.LocalConfig)* – Config object to use. (optional)
- **validate_connection** – Validate database connection and schema immediately

**Return type** `datacube.index.index.Index`

**Raises** `datacube.drivers.postgres._connections.IndexSetupError`

### datacube.index.Index

**class** `datacube.index.Index(db: datacube.drivers.postgres._connections.PostgresDb)`

Access to the datacube index.

DON’T INITIALISE THIS DIRECTLY (it will break in the future). Use `datacube.index.index_connect()` or access property `.index` on your existing `datacube.api.core.Datacube`.

These are thread safe. But not multiprocess safe once a connection is made (db connections cannot be shared between processes) You can close idle connections before forking by calling `close()`, provided you know no other connections are active. Or else use a separate instance of this class in each process.

**Variables**

- **datasets** *(datacube.index._datasets.DatasetResource)* – store and retrieve `datacube.model.Dataset`
- **products** *(datacube.index._products.ProductResource)* – store and retrieve `datacube.model.DatasetType` (should really be called Product)
- **metadata_types** *(datacube.index._metadata_types.MetadataTypeResource)* – store and retrieve `datacube.model.MetadataType`
- **users** *(UserResource)* – user management

**__init__** *(db: datacube.drivers.postgres._connections.PostgresDb) → None*

Initialize self. See help(type(self)) for accurate signature.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong><strong>init</strong></strong>(db)</td>
<td>Initialize self.</td>
</tr>
<tr>
<td>close()</td>
<td>Close any idle connections database connections.</td>
</tr>
<tr>
<td>from_config(config[, application_name,...])</td>
<td></td>
</tr>
<tr>
<td>get_dataset_fields(doc)</td>
<td></td>
</tr>
<tr>
<td>init_db([with_default_types, with_permissions])</td>
<td></td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td></td>
</tr>
</tbody>
</table>
17.6.6 Dataset to Product Matching

**Doc2Dataset**

Used for constructing *Dataset* objects from plain metadata documents.

datacube.index.hl.Doc2Dataset

class datacube.index.hl.Doc2Dataset(index, products=None, exclude_products=None, fail_on_missing_lineage=False, verify_lineage=True, skip_lineage=False)

Used for constructing *Dataset* objects from plain metadata documents.

This requires a database connection to perform the automatic matching against available products.

There are options for including and excluding the products to match against, as well as how to deal with source lineage.

Once constructed, call with a dictionary object and location URI, eg:

```python
resolver = Doc2Dataset(index)
dataset = resolver(dataset_dictionary, 'file:///tmp/test-dataset.json')
index.dataset.add(dataset)
```

**Parameters**

- **index** – an open Database connection
- **products** (*list*) – List of product names against which to match datasets

(including lineage datasets), if not supplied we will consider all products.

**Parameters**

- **exclude_products** (*list*) – List of products to exclude from matching
- **fail_on_missing_lineage** – If True fail resolve if any lineage datasets are missing from the DB

**Parameters** verify_lineage – If True check that lineage datasets in the supplied document are identical to DB versions

**Parameters** skip_lineage – If True ignore lineage sub-tree in the supplied document and construct dataset without lineage datasets

**Parameters**

- **index** – Database
- **products** – List of product names against which to match datasets (including lineage datasets), if not supplied will consider all products.
- **exclude_products** – List of products to exclude from matching
- **fail_on_missing_lineage** – If True fail resolve if any lineage datasets are missing from the DB
- **verify_lineage** – If True check that lineage datasets in the supplied document are identical to DB versions
• `skip_lineage` – If True ignore lineage sub-tree in the supplied document and construct
dataset without lineage datasets

```python
__init__(index, products=None, exclude_products=None, fail_on_missing_lineage=False, verify_lineage=True, skip_lineage=False)
```
Initialize self. See help(type(self)) for accurate signature.

### 17.7 Geometry Utilities

Open Data Cube includes a set of CRS aware geometry utilities.

#### 17.7.1 Geometry Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datacube.utils.geometry.BoundingBox</code></td>
<td>Bounding box, defining extent in cartesian coordinates.</td>
</tr>
<tr>
<td><code>datacube.utils.geometry.CRS</code></td>
<td>Wrapper around <code>osr.SpatialReference</code> providing a more pythonic interface</td>
</tr>
<tr>
<td><code>datacube.utils.geometry.Geometry</code></td>
<td>2D Geometry with CRS</td>
</tr>
<tr>
<td><code>model.GridSpec</code></td>
<td>Definition for a regular spatial grid</td>
</tr>
</tbody>
</table>

**datacube.utils.geometry.BoundingBox**

```python
class datacube.utils.geometry.BoundingBox

Bounding box, defining extent in cartesian coordinates.

__init__()

Initialize self. See help(type(self)) for accurate signature.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffered(xbuff, ybuff)</td>
<td>Return a new BoundingBox, buffered in the x and y dimensions.</td>
</tr>
<tr>
<td>count</td>
<td>Return number of occurrences of value.</td>
</tr>
<tr>
<td>index</td>
<td>Return first index of value.</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bottom</td>
<td>Alias for field number 1</td>
</tr>
<tr>
<td>height</td>
<td>Alias for field number 0</td>
</tr>
<tr>
<td>left</td>
<td>Alias for field number 0</td>
</tr>
<tr>
<td>right</td>
<td>Alias for field number 2</td>
</tr>
<tr>
<td>top</td>
<td>Alias for field number 3</td>
</tr>
<tr>
<td>width</td>
<td>Alias for field number 3</td>
</tr>
</tbody>
</table>
datacube.utils.geometry.CRS

class datacube.utils.geometry.CRS(crs_str)
Wrapper around osr.SpatialReference providing a more pythonic interface

__init__(crs_str)

Parameters crs_str -- string representation of a CRS, often an EPSG code like ‘EPSG:4326’

Raises InvalidCRSError

Methods

__init__(crs_str)

param crs_str string representation of a CRS, often an EPSG code like ‘EPSG:4326’

Attributes

dimensions List of dimension names of the CRS
epsg EPSG Code of the CRS or None
geographic bool
inverse_flattening proj
semi_major_axis
semi_minor_axis
units List of dimension units of the CRS
wkt WKT representation of the CRS

datacube.utils.geometry.Geometry

class datacube.utils.geometry.Geometry(geo, crs=None)

2D Geometry with CRS

Instantiate with a GeoJSON structure

If 3D coordinates are supplied, they are converted to 2D by dropping the Z points.

__init__(geo, crs=None)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(geo, crs)

buffer(distance[, quadsecs])
contains
crosses

Continued on next page
### Table 38 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>difference</td>
<td>Returns a point distance units along the line or None if underlying geometry doesn’t support this operation.</td>
</tr>
<tr>
<td>disjoint</td>
<td></td>
</tr>
<tr>
<td>interpolate(distance)</td>
<td></td>
</tr>
<tr>
<td>intersection</td>
<td></td>
</tr>
<tr>
<td>intersects</td>
<td></td>
</tr>
<tr>
<td>overlaps</td>
<td></td>
</tr>
<tr>
<td>segmented(resolution)</td>
<td>Possibly add more points to the geometry so that no edge is longer than resolution</td>
</tr>
<tr>
<td>simplify(tolerance)</td>
<td></td>
</tr>
<tr>
<td>symmetric_difference</td>
<td></td>
</tr>
<tr>
<td>to_crs(crs[, resolution, wrapdateline])</td>
<td>Convert geometry to a different Coordinate Reference System</td>
</tr>
<tr>
<td>touches</td>
<td></td>
</tr>
<tr>
<td>union</td>
<td></td>
</tr>
<tr>
<td>within</td>
<td></td>
</tr>
</tbody>
</table>

#### Attributes

- area
- boundary
- boundingbox
- centroid
- convex_hull
- coords
- envelope
- is_empty
- is_valid
- json
- length
- points
- type
- wkt

### 17.7.2 Creating Geometries

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>point(x, y, crs)</td>
<td>Create a 2D Point</td>
</tr>
<tr>
<td>multipoint(coords, crs)</td>
<td>Create a 2D MultiPoint Geometry</td>
</tr>
<tr>
<td>line(coords, crs)</td>
<td>Create a 2D LineString (Connected set of lines)</td>
</tr>
<tr>
<td>multiline(coords, crs)</td>
<td>Create a 2D MultiLineString (Multiple disconnected sets of lines)</td>
</tr>
<tr>
<td>polygon(outer, crs, *inners)</td>
<td>Create a 2D Polygon</td>
</tr>
<tr>
<td>multipolygon(coords, crs)</td>
<td>Create a 2D MultiPolygon</td>
</tr>
<tr>
<td>box(left, bottom, right, top, crs)</td>
<td>Create a 2D Box (Polygon)</td>
</tr>
<tr>
<td>polygon_from_transform(width, height, ...)</td>
<td>Create a 2D Polygon from an affine transform</td>
</tr>
</tbody>
</table>
**datacube.utils.geometry.point**

`datacube.utils.geometry.point(x, y, crs)`  
Create a 2D Point

```python
>>> point(10, 10, crs=None)
Geometry(POINT (10 10), None)
```

**Return type**  
`Geometry`

**datacube.utils.geometry.multipoint**

`datacube.utils.geometry.multipoint(coords, crs)`  
Create a 2D MultiPoint Geometry

```python
>>> multipoint([(10, 10), (20, 20)], None)
Geometry(MULTIPOINT (10 10,20 20), None)
```

**Parameters**  
`coords` (*list*) – list of x,y coordinate tuples

**Return type**  
`Geometry`

**datacube.utils.geometry.line**

`datacube.utils.geometry.line(coords, crs)`  
Create a 2D LineString (Connected set of lines)

```python
>>> line([(10, 10), (20, 20), (30, 40)], None)
Geometry(LINESTRING (10 10,20 20,30 40), None)
```

**Parameters**  
`coords` (*list*) – list of x,y coordinate tuples

**Return type**  
`Geometry`

**datacube.utils.geometry.multiline**

`datacube.utils.geometry.multiline(coords, crs)`  
Create a 2D MultiLineString (Multiple disconnected sets of lines)

```python
>>> multiline([(10, 10), (20, 20), (30, 40)], [(50, 60), (70, 80), (90, 99)], None)
Geometry(MULTILINESTRING ((10 10,20 20,30 40),(50 60,70 80,90 99)), None)
```

**Parameters**  
`coords` (*list*) – list of lists of x,y coordinate tuples

**Return type**  
`Geometry`

**datacube.utils.geometry.polygon**

`datacube.utils.geometry.polygon(outer, crs, *inners)`  
Create a 2D Polygon
```python
>>> polygon([(10, 10), (20, 20), (20, 10), (10, 10),], None)
Geometry(POLYGON ((10 10,20 20,20 10,10 10)), None)
```

Parameters `coords` *(list)* – list of 2d x,y coordinate tuples

Return type `Geometry`

`datacube.utils.geometry.multipolygon`

```python
>>> multipolygon([[[(10, 10), (20, 20), (20, 10), (10, 10)]], [[[40, 10), (50, 20), (50, 10), (40, 10)]], None)
Geometry(MULTIPOLYGON (((10 10,20 20,20 10,10 10)),((40 10,50 20,50 10,40 10))), None)
```

Parameters `coords` *(list)* – list of lists of x,y coordinate tuples

Return type `Geometry`

`datacube.utils.geometry.box`

```python
>>> box(10, 10, 20, 20, None)
Geometry(POLYGON ((10 10,10 20,20 20,20 10,10 10)), None)
```

`datacube.utils.geometry_polygon_from_transform`

```python
>>> polygon_from_transform(10, 20, 50, 40, None)
Geometry(POLYGON ((10 10,10 20,20 20,20 10,10 10)), None)
```

Parameters

• `width` *(float)*

• `height` *(float)*

• `transform` *(Affine)*

• `crs` – CRS

Return type `Geometry`

### 17.7.3 Multi-geometry ops

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unary_union</code> (geoms)</td>
<td>compute union of multiple (multi)polygons efficiently</td>
</tr>
<tr>
<td><code>unary_intersection</code> (geoms)</td>
<td>compute intersection of multiple (multi)polygons</td>
</tr>
</tbody>
</table>

17.7. Geometry Utilities
datacube.utils.geometry.unary_union

```python
datacube.utils.geometry.unary_union(geoms)
compute union of multiple (multi)polygons efficiently
```

datacube.utils.geometry.unary_intersection

```python
datacube.utils.geometry.unary_intersection(geoms)
compute intersection of multiple (multi)polygons
```

17.8 Masking

17.8.1 Bit Masking

Masking No Data Values

```python
masking.mask_invalid_data(keep_attrs=True)
Sets all nodata values to nan.
```

This will convert numeric data to type float.

**Parameters**

- `data (Dataset)`
  - `keep_attrs (bool)` - If the attributes of the data should be included in the returned.

**Returns** Dataset or DataArray

Masking with Bit-Flag Measurements

One of the common types of data used with the Data Cube contains discrete values stored within a numeric value. These values are often classifications and outputs from tests, and need to be interpreted in specific ways, not as simple scalar values. They are often used as a mask to exclude observations which deemed unsuitable for a given analysis.

For example, we want to exclude observations of clouds when we are interested in what is on the ground.

Several methods are used when encoding these types of variables:

- On-off bit flags, for a particular binary bit
- Collections of bits that can indicate more than two possible values
- Looking for a specific value stored using all available bits in the variable

From prior work, it is very easy to make mistakes when using these types of variables, which can lead to processing the wrong set of observations, and also making it quite difficult to read the code using them, and ensuring that they are used in a consistent way in different places.

Open Data Cube provides a way of describing the meanings that can be encoded in variables, which can then be used to give a readable method when using that variable.
How to Define Meanings on Measurements

**Warning:** TODO

How to Create Masks within code

```
masking.describe_variable_flags(with_pandas=True)
```

Returns either a Pandas Dataframe (with_pandas=True - default) or a string (with_pandas=False) describing the available flags for a masking variable.

Interprets the `flags_definition` attribute on the provided variable and returns a Pandas Dataframe or string like:

<table>
<thead>
<tr>
<th>Bits are listed from the MSB (bit 13) to the LSB (bit 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Value Flag Name Description</td>
</tr>
<tr>
<td>13 0 cloud_shadow_fmask Cloud Shadow (Fmask)</td>
</tr>
<tr>
<td>12 0 cloud_shadow_acca Cloud Shadow (ACCA)</td>
</tr>
<tr>
<td>11 0 cloud_fmask Cloud (Fmask)</td>
</tr>
<tr>
<td>10 0 cloud_acca Cloud (ACCA)</td>
</tr>
</tbody>
</table>

**Parameters**
- `variable` – Masking `xarray.Dataset` or `xarray.DataArray`

**Returns**
- Pandas Dataframe or str

```
masking.make_mask(**flags)**
```

Returns a mask array, based on provided flags.

When multiple flags are provided, they will be combined in a logical AND fashion.

For example:

```
>>> make_mask(pqa, cloud_acca=False, cloud_fmask=False, land_obs=True) # doctest: +SKIP
```

OR

```
>>> make_mask(pqa, **GOOD_PIXEL_FLAGS) # doctest: +SKIP
```

where `GOOD_PIXEL_FLAGS` is a dict of flag_name to True/False

**Parameters**
- `variable` (`xarray.Dataset` or `xarray.DataArray`) –
- `flags` – list of boolean flags

**Returns**
- boolean `xarray.DataArray` or `xarray.Dataset`

```
masking.mask_invalid_data(data[, keep_attrs])
```

Sets all `nodata` values to `nan`.

```
masking.describe_variable_flags(variable[, ...])
```

Returns either a Pandas Dataframe (with_pandas=True - default) or a string (with_pandas=False) describing the available flags for a masking variable.

```
masking.make_mask(variable, **flags)**
```

Returns a mask array, based on provided flags.
17.9 Query Class

```python
Query([index, product, geopolygon, like])
```

17.9.1 datacube.api.query.Query

class datacube.api.query.Query(index=None, product=None, geopolygon=None, like=None, **search_terms)

```python
__init__(index=None, product=None, geopolygon=None, like=None, **search_terms)
```

Parses search terms in preparation for querying the Data Cube Index.

Create a `Query` object by passing it a set of search terms as keyword arguments.

```python
>>> query = Query(product='ls5_nbar_albers', time=('2001-01-01', '2002-01-01'))
```

Use by accessing `search_terms`:

```python
>>> query.search_terms['time']  # doctest: +NORMALIZE_WHITESPACE
Range(begin=datetime.datetime(2001, 1, 1, 0, 0, tzinfo=<UTC>),
      end=datetime.datetime(2002, 1, 1, 23, 59, 59, 999999, tzinfo=tzutc()))
```

By passing in an `index`, the search parameters will be validated as existing on the product.

Used by `datacube.Datacube.find_datasets()` and `datacube.Datacube.load()`.

**Parameters**

- **index** (datacube.index.Index) – An optional `index` object, if checking of field names is desired.
- **product** (str) – name of product
- **geopolygon** (geometry.Geometry or None) – spatial bounds of the search
- **like** (xarray.Dataset) – spatio-temporal bounds of `like` are used for the search
- **search_terms** –
  - `measurements` - list of measurements to retrieve
  - `latitude, lat, y, longitude, lon, long, x` - tuples (min, max) bounding spatial dimensions
  - `crs` - spatial coordinate reference system to interpret the spatial bounds
  - `group_by` - observation grouping method. One of `time, solar_day`. Default is `time`

**Methods**

```python
__init__([index, product, geopolygon, like])
```

Parses search terms in preparation for querying the Data Cube Index.

**Attributes**
17.10 User Configuration

LocalConfig(config[, files_loaded, env]) System configuration for the user.

DEFAULT_CONF_PATHS Config locations in order.

17.10.1 datacube.config.LocalConfig

class datacube.config.LocalConfig (config, files_loaded=None, env=None)

System configuration for the user.

This loads from a set of possible configuration files which define the available environments. An environment contains connection details for a Data Cube Index, which provides access to available data.

__init__ (config, files_loaded=None, env=None)

Datacube environment resolution precedence is:

1. Supplied as a function argument env
2. DATACUBE_ENVIRONMENT environment variable
3. user.default_environment option in the config
4. ‘default’ or ‘datacube’ whichever is present

If environment is supplied by any of the first 3 methods is not present in the config, then throw an exception.

Methods

__init__ (config[, files_loaded, env])

find([paths, env]) Find config from possible filesystem locations.

get(item[, fallback])

17.10.2 datacube.config.DEFAULT_CONF_PATHS

datacube.config.DEFAULT_CONF_PATHS = ('/etc/datacube.conf', None, '/home/docs/.datacube.conf', 'datacube.conf')

Config locations in order. Properties found in latter locations override earlier ones.

- /etc/datacube.conf
- file at $DATACUBE_CONFIG_PATH environment variable
- ~/.datacube.conf
- datacube.conf
17.11 Everything Else

17.11.1 Analytics and Execution Engines

The old Analytics and Execution Engines have been deprecated, but replacements are taking shape inside the execution engine branch on GitHub. Check out the work there for the latest updates.

17.11.2 Array IO on Amazon S3

S3 Byte IO

class datacube.drivers.s3.storage.s3aios3IO(enable_s3=True, file_path=None, num_workers=30)

low level S3 byte IO interface.

bucket_exists (s3_bucket, new_session=False)

Check if bucket exists.

Parameters

• s3_bucket (str) – name of the s3 bucket.

• new_session (bool) – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

Returns Returns True if bucket exists, otherwise False.

delete_created_arrays ()

Delete all created shared memory arrays.

Arrays are prefixed by ‘S3’ or ‘DCCORE’.

delete_objects (s3_bucket, keys, new_session=False)

Delete S3 objects.

Parameters

• s3_bucket (str) – name of the s3 bucket.

• keys (int) – list of s3 keys to delete

• new_session (bool) – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

Returns Returns List of deleted objects.

get_byte_range (s3_bucket, s3_key, s3_start, s3_end, new_session=False)

Gets bytes from a S3 object within a range.

Parameters

• s3_bucket (str) – name of the s3 bucket.

• s3_key (str) – name of the s3 key.

• s3_start (int) – begin of range.

• s3_end (int) – begin of range.

• new_session (bool) – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

Returns Requested bytes
get_byte_range_mp \( (s3\_bucket, s3\_key, s3\_start, s3\_end, block\_size, new\_session=False) \)

Gets bytes from a S3 object within a range in parallel.

**Parameters**

- **s3_bucket** *(str)* – name of the s3 bucket.
- **s3_key** *(str)* – name of the s3 key.
- **s3_start** *(int)* – begin of range.
- **s3_end** *(int)* – begin of range.
- **block_size** *(int)* – block size for download.
- **new_session** *(bool)* – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

**Returns** Requested bytes

get_bytes \( (s3\_bucket, s3\_key, new\_session=False) \)

Gets bytes from a S3 object

**Parameters**

- **s3_bucket** *(str)* – name of the s3 bucket.
- **s3_key** *(str)* – name of the s3 key.
- **new_session** *(bool)* – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

**Returns** Requested bytes

list_buckets \( (new\_session=False) \)

List S3 buckets.

**Parameters** **new_session** *(bool)* – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

**Returns** Returns list of Buckets

list_created_arrays

List the created shared memory arrays.

Arrays are prefixed by ‘S3’ or ‘DCCORE’.

**Returns** Returns the list of created arrays.

list_objects \( (s3\_bucket, prefix=", max\_keys=100, new\_session=False) \)

List S3 objects.

**Parameters**

- **s3_bucket** *(str)* – name of the s3 bucket.
- **prefix** *(str)* – prefix of buckets to list
- **max_keys** *(int)* – max keys to return
- **new_session** *(bool)* – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

**Returns** Returns list of objects.

object_exists \( (s3\_bucket, s3\_key, new\_session=False) \)

Check if object exists.
Parameters

- **s3_bucket** *(str)* – name of the s3 bucket.
- **s3_key** *(str)* – name of the s3 key.
- **new_session** *(bool)* – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

Returns Returns True if object exsists, otherwise False.

**put_bytes** *(s3_bucket, s3_key, data, new_session=False)*

Put bytes into a S3 object.

Parameters

- **s3_bucket** *(str)* – name of the s3 bucket.
- **s3_key** *(str)* – name of the s3 key.
- **data** *(bytes)* – data to store in s3.
- **new_session** *(bool)* – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

**put_bytes_mpu** *(s3_bucket, s3_key, data, block_size, new_session=False)*

Put bytes into a S3 object using Multi-Part upload.

Parameters

- **s3_bucket** *(str)* – name of the s3 bucket.
- **s3_key** *(str)* – name of the s3 key.
- **data** *(bytes)* – data to store in s3.
- **block_size** *(int)* – block size for upload.
- **new_session** *(bool)* – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

Returns Multi-part upload response

**put_bytes_mpu_mp** *(s3_bucket, s3_key, data, block_size, new_session=False)*

Put bytes into a S3 object using Multi-Part upload in parallel.

Parameters

- **s3_bucket** *(str)* – name of the s3 bucket.
- **s3_key** *(str)* – name of the s3 key.
- **data** *(bytes)* – data to store in s3.
- **block_size** *(int)* – block size for upload.
- **new_session** *(bool)* – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

Returns Multi-part upload response

**put_bytes_mpu_mp_shm** *(s3_bucket, s3_key, array_name, block_size, new_session=False)*

Put bytes into a S3 object using Multi-Part upload in parallel with shared memory.

Parameters

- **s3_bucket** *(str)* – name of the s3 bucket.
- **s3_key** *(str)* – name of the s3 key.
• **data** *(bytes)* – data to store in s3.
• **block_size** *(int)* – block size for upload.
• **new_session** *(bool)* – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

**Returns** Multi-part upload response

**s3_bucket** *(s3_bucket, new_session=False)*
get a reference to a S3 bucket.

**Parameters**

• **s3_bucket** *(str)* – name of the s3 bucket.
• **new_session** *(bool)* – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

**Returns** Returns a reference to the S3 bucket.

**s3_object** *(s3_bucket, s3_key, new_session=False)*
get a reference to a S3 object.

**Parameters**

• **s3_bucket** *(str)* – name of the s3 bucket.
• **s3_key** *(str)* – name of the s3 key.
• **new_session** *(bool)* – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

**Returns** Returns a reference to the S3 object.

**s3_resource** *(new_session=False)*
Create a S3 resource.

**Parameters** **new_session** *(bool)* – Flag to create a new session or reuse existing session. True: create new session False: reuse existing session

**Returns** Returns a reference to the S3 resource.

### S3 Array IO

**class** `datacube.drivers.s3.storage.s3aio.S3AIO(enable_compression=True, enable_s3=True, file_path=None, num_workers=30)`

**get_point** *(index_point, shape, dtype, s3_bucket, s3_key)*
Gets a point in the nd array stored in S3.

Only works if compression is off.

**Parameters**

• **index_point** *(tuple)* – Index of the point to be retrieved.
• **shape** *(tuple)* – Shape of the stored data.
• **numpy.dtype** – dtype of the stored data.
• **s3_bucket** *(str)* – S3 bucket name
• **s3_key** *(str)* – S3 key name
Returns Returns the point data.

**get_slice** (*array_slice, shape, dtype, s3_bucket, s3_key*)

Gets a slice of the nd array stored in S3.

Only works if compression is off.

Parameters

- **array_slice** (*tuple*) – tuple of slices to retrieve.
- **shape** (*tuple*) – Shape of the stored data.
- **numpy.dtype** – dtype of the stored data.
- **s3_bucket** (*str*) – S3 bucket name
- **s3_key** (*str*) – S3 key name

Returns Returns the data slice.

**get_slice_by_bbox** (*array_slice, shape, dtype, s3_bucket, s3_key*)

Gets a slice of the nd array stored in S3 by bounding box.

Parameters

- **array_slice** (*tuple*) – tuple of slices to retrieve.
- **shape** (*tuple*) – Shape of the stored data.
- **numpy.dtype** – dtype of the stored data.
- **s3_bucket** (*str*) – S3 bucket name
- **s3_key** (*str*) – S3 key name

Returns Returns the data slice.

**get_slice_mp** (*array_slice, shape, dtype, s3_bucket, s3_key*)

Gets a slice of the nd array stored in S3 in parallel.

Only works if compression is off.

Parameters

- **array_slice** (*tuple*) – tuple of slices to retrieve.
- **shape** (*tuple*) – Shape of the stored data.
- **numpy.dtype** – dtype of the stored data.
- **s3_bucket** (*str*) – S3 bucket name
- **s3_key** (*str*) – S3 key name

Returns Returns the data slice.

**to_1d** (*index, shape*)

Converts nD index to 1D index.

Parameters

- **index** (*tuple*) – N-D Index to be converted.
- **shape** (*tuple*) – Shape to be used for conversion.

Returns Returns the 1D index.

**to_nd** (*index, shape*)

Converts 1D index to nD index.
Parameters

- **index** *(tuple)* – 1D Index to be converted.
- **shape** *(tuple)* – Shape to be used for conversion.

Returns Returns the ND index.

### S3 Labeled IO

class datacube.drivers.s3.storage.s3aio.S3LIO(enable_compression=True, enable_s3=True, file_path=None, num_workers=30)

assemble_array_from_s3 (array, indices, s3_bucket, s3_keys, dtype)
Reconstruct an array from S3.

Parameters

- **array** *(ndarray)* – array to be put into S3
- **indices** *(list)* – indices corresponding to the s3 keys
- **s3_bucket** *(str)* – S3 bucket to use
- **s3_keys** *(list)* – List of S3 keys corresponding to the indices.

Returns The assembled array.

chunk_indices_1d (begin, end, step, bound_slice=None, return_as_shape=False)
Chunk a 1D index.

Parameters

- **begin** *(int)* – Start of index range.
- **end** *(int)* – Stop of index range.
- **step** *(int)* – Step size of index range.
- **bound_slice** *(Slice)* – bounds of the index range
- **return_as_shape** *(bool)* – Returns as a shape if set to True, otherwise a Slice.

Returns Returns the chunked indices.

chunk_indices_nd (shape, chunk, array_slice=None, return_as_shape=False)
Chunk a nD index.

Parameters

- **shape** *(tuple)* – Shape of the index
- **chunk** *(tuple)* – desired chunk size
- **array_slice** *(tuple)* – The array slice
- **return_as_shape** *(bool)* – Returns as a shape if set to True, otherwise a Slice.

Returns Returns the chunked indices.

get_data (base_location, dimension_range, micro_shape, dtype, labeled_slice, s3_bucket, use_hash=False)
Gets geo-referenced indexed data from S3.

Not yet implemented.
Parameters

- **base_location** *(str)* – The base location of the requested data.
- **dimension_range** *(tuple)* – The dimension extents of the data.
- **micro_shape** *(tuple)* – The micro shape of the data.
- **dtype** *(numpy.dtype)* – The data type of the data.
- **labeled_slice** *(tuple)* – The requested nD array slice.
- **s3_bucket** *(str)* – The S3 bucket name.
- **use_hash** *(bool)* – Whether to prefix the key with a deterministic hash.

Returns

The nd array.

**get_data_unlabeled** *(base_location, macro_shape, micro_shape, dtype, array_slice, s3_bucket, use_hash=False)*

Gets integer indexed data from S3.

Parameters

- **base_location** *(str)* – The base location of the requested data.
- **macro_shape** *(tuple)* – The macro shape of the data.
- **micro_shape** *(tuple)* – The micro shape of the data.
- **dtype** *(numpy.dtype)* – The data type of the data.
- **array_slice** *(tuple)* – The requested nD array slice.
- **s3_bucket** *(str)* – The S3 bucket name.
- **use_hash** *(bool)* – Whether to prefix the key with a deterministic hash.

Returns

The nd array.

**put_array_in_s3** *(array, chunk_size, base_name, bucket, spread=False)*

Put array in S3.

Parameters

- **array** *(ndarray)* – array to be put into S3
- **chunk_size** *(tuple)* – chunk size to use for storage
• **base_name** (*str*) – The base name for the S3 key
• **bucket** (*str*) – S3 bucket to use
• **spread** (*bool*) – Flag to use a deterministic hash as a prefix.

**Returns**

Returns the a dict of (keys, indices, chunk ids)

```python
put_array_in_s3_mp(array, chunk_size, base_name, bucket, spread=False)
```

Put array in S3 in parallel.

**Parameters**

- **array** (*ndarray*) – array to be put into S3
- **chunk_size** (*tuple*) – chunk size to use for storage
- **base_name** (*str*) – The base name for the S3 key
- **bucket** (*str*) – S3 bucket to use
- **spread** (*bool*) – Flag to use a deterministic hash as a prefix.

**Returns**

Returns the a dict of (keys, indices, chunk ids)

```python
regular_index(query, dimension_range, shape, flatten=False)
```

converts positional(spatial/temporal) coordinates to array integer coordinates

**Parameters**

- **query** (*tuple*) – The range query.
- **dimension_range** (*tuple*) – Dimension range extents.
- **shape** (*tuple*) – Shape.
- **flatten** (*bool*) – returns 1D index

**Returns**

The regular index.

```python
shard_array_to_s3(array, indices, s3_bucket, s3_keys)
```

Shard array to S3.

**Parameters**

- **array** (*ndarray*) – array to be put into S3
- **indices** (*list*) – indices corrsponding to the s3 keys
- **s3_bucket** (*str*) – S3 bucket to use
- **s3_keys** (*list*) – List of S3 keys corresponding to the indices.

```python
shard_array_to_s3_mp(array, indices, s3_bucket, s3_keys)
```

Shard array to S3 in parallel.

**Parameters**

- **array** (*ndarray*) – array to be put into S3
- **indices** (*list*) – indices corrsponding to the s3 keys
- **s3_bucket** (*str*) – S3 bucket to use
- **s3_keys** (*list*) – List of S3 keys corresponding to the indices.
17.11.3 External Classes

**class affine.Affine**

Two dimensional affine transform for 2D linear mapping.

Parallel lines are preserved by these transforms. Affine transforms can perform any combination of translations, scales/flips, shears, and rotations. Class methods are provided to conveniently compose transforms from these operations.

Internally the transform is stored as a 3x3 transformation matrix. The transform may be constructed directly by specifying the first two rows of matrix values as 6 floats. Since the matrix is an affine transform, the last row is always \((0, 0, 1)\).

N.B.: multiplication of a transform and an \((x, y)\) vector *always* returns the column vector that is the matrix multiplication product of the transform and \((x, y)\) as a column vector, no matter which is on the left or right side. This is obviously not the case for matrices and vectors in general, but provides a convenience for users of this class.

**Parameters**

members (**float**) – 6 floats for the first two matrix rows.

**almost_equals** (**other**, **precision=1e-05**)

Compare transforms for approximate equality.

**Parameters**

other (**Affine**) – Transform being compared.

**Returns**

True if absolute difference between each element of each respective transform matrix < self.precision.

**column_vectors**

The values of the transform as three 2D column vectors

**determinant**

The determinant of the transform matrix.

This value is equal to the area scaling factor when the transform is applied to a shape.

**eccentricity**

The eccentricity of the affine transformation.

This value represents the eccentricity of an ellipse under this affine transformation.

Raises **NotImplementedError** for improper transformations.

**classmethod from_gdal** (**c**, **a**, **b**, **f**, **d**, **e**)

Use same coefficient order as GDAL’s GetGeoTransform().

**Parameters**

a, b, f, d, e (**c**), – 6 floats ordered by GDAL.

**Return type** **Affine**

**classmethod identity** ()

Return the identity transform.

**Return type** **Affine**

**is_conformal**

True if the transform is conformal.

i.e., if angles between points are preserved after applying the transform, within rounding limits. This implies that the transform has no effective shear.

**is_degenerate**

True if this transform is degenerate.
Which means that it will collapse a shape to an effective area of zero. Degenerate transforms cannot be inverted.

**is_identity**
True if this transform equals the identity matrix, within rounding limits.

**is_orthonormal**
True if the transform is orthonormal.

Which means that the transform represents a rigid motion, which has no effective scaling or shear. Mathematically, this means that the axis vectors of the transform matrix are perpendicular and unit-length. Applying an orthonormal transform to a shape always results in a congruent shape.

**is_proper**
True if this transform is proper.

Which means that it does not include reflection.

**is_rectilinear**
True if the transform is rectilinear.

i.e., whether a shape would remain axis-aligned, within rounding limits, after applying the transform.

**itransform**(seq)
Transform a sequence of points or vectors in place.

**Parameters**
seq – Mutable sequence of Vec2 to be transformed.

**Returns**
None, the input sequence is mutated in place.

**classmethod permutation**(scaling)
Create the permutation transform. For 2x2 matrices, there is only one permutation matrix that is not the identity.

**Return type**
Affine

**classmethod rotation**(angle, pivot=None)
Create a rotation transform at the specified angle.

A pivot point other than the coordinate system origin may be optionally specified.

**Parameters**

• **angle** (float) – Rotation angle in degrees, counter-clockwise about the pivot point.

• **pivot** (sequence) – Point to rotate about, if omitted the rotation is about the origin.

**Return type**
Affine

**rotation_angle**
The rotation angle in degrees of the affine transformation.

This is the rotation angle in degrees of the affine transformation, assuming it is in the form M = R S, where R is a rotation and S is a scaling.

Raises NotImplementedError for improper transformations.

**classmethod scale**(scaling)
Create a scaling transform from a scalar or vector.

**Parameters**

• **scaling** (float or sequence) – The scaling factor. A scalar value will scale in both dimensions equally. A vector scaling value scales the dimensions independently.

**Return type**
Affine

17.11. Everything Else 101
classmethod shear \((x\_angle=0, y\_angle=0)\)
Create a shear transform along one or both axes.

**Parameters**

- **x\_angle** \((\text{float})\) – Shear angle in degrees parallel to the x-axis.
- **y\_angle** \((\text{float})\) – Shear angle in degrees parallel to the y-axis.

**Return type** *Affine*

to\_gdal()
Return same coefficient order as GDAL’s SetGeoTransform().

**Return type** *tuple*

classmethod translation \((xoff, yoff)\)
Create a translation transform from an offset vector.

**Parameters**

- **xoff** \((\text{float})\) – Translation x offset.
- **yoff** \((\text{float})\) – Translation y offset.

**Return type** *Affine*

\(xoff\)
Alias for ‘c’

\(yoff\)
Alias for ‘f’

For **Exploratory Data Analysis** see *Datacube Class* for more details

For **Writing Large Scale Workflows** see *Grid Processing API* for more details
CHAPTER 18

Developer Setup

18.1 Ubuntu Developer Setup

Base OS: Ubuntu 16.04 LTS

This guide will setup an ODC core development environment and includes:

- Anaconda python using conda environments to isolate the odc development environment
- Installation of required software and useful developer manuals for those libraries
- Postgres database installation with a local user configuration
- Integration tests to confirm both successful development setup and for ongoing testing
- Build configuration for local ODC documentation

18.1.1 Required software

GDAL, HDF5, and netCDF4:

```
sudo apt-get install libgdal1-dev libhdf5-serial-dev libnetcdf-dev
```

Postgres:

```
sudo apt-get install postgresql-9.5 postgresql-client-9.5 postgresql-contrib-9.5
```

Optional packages (useful utilities, docs):

```
sudo apt-get install postgresql-doc-9.5 libhdf5-doc netcdf-doc libgdal-doc
sudo apt-get install hdf5-tools netcdf-bin gdal-bin pgadmin3
```
18.1.2 Python and packages

Python 3.5+ is required. Python 3.6 is recommended.

**Anaconda Python**

Install Anaconda Python

Add conda-forge to package channels:

```bash
conda config --add channels conda-forge
```

Conda Environments are recommended for use in isolating your ODC development environment from your system installation and other python environments.

Install required python packages and create an odc conda environment.

Python 3.6:

```bash
conda env create -n odc --file .travis/environment.yaml sphinx
```

Activate odc python environment:

```bash
source activate odc
```

18.1.3 Postgres database configuration

This configuration supports local development using your login name.

If this is a new installation of Postgres on your system it is probably wise to set the postgres user password. As the local “postgres” Linux user, we are allowed to connect and manipulate the server using the psql command.

In a terminal, type:

```bash
sudo -u postgres psql postgres
```

Set a password for the “postgres” database role using the command:

```bash
\password postgres
```

and set the password when prompted. The password text will be hidden from the console for security purposes.

Type **Control+D** or **q** to exit the postgresQL prompt.

By default in Ubuntu, Postgresql is configured to use **ident sameuser** authentication for any connections from the same machine which is useful for development. Check out the excellent Postgresql documentation for more information, but essentially this means that if your Ubuntu username is foo and you add foo as a Postgresql user then you can connect to a database without requiring a password for many functions.

Since the only user who can connect to a fresh install is the postgres user, here is how to create yourself a database account (which is in this case also a database superuser) with the same name as your login name and then create a password for the user:

```bash
sudo -u postgres createuser --superuser $USER
sudo -u postgres psql
postges=# \password $USER
```
Now we can create an `agdcintegration` database for testing:

```bash
createdb agdcintegration
```

Connecting to your own database to try out some SQL should now be as easy as:

```bash
psql -d agdcintegration
```

### 18.1.4 Open Data Cube source and development configuration

Download the latest version of the software from the repository

```bash
git clone https://github.com/opendatacube/datacube-core
cd datacube-core
```

We need to specify the database user and password for the ODC integration testing. To do this:

```bash
cp integration_tests/agdcintegration.conf ~/.datacube_integration.conf
```

Then edit the `~/.datacube_integration.conf` with a text editor and add the following lines replacing `<foo>` with your username and `<foobar>` with the database user password you set above (not the postgres one, your `<foo>` one):

```ini
[datacube]
db_hostname: localhost
db_database: agdcintegration
db_username: <foo>
db_password: <foobar>
```

### 18.1.5 Verify it all works

Run the integration tests:

```bash
cd datacube-core
./check-code.sh integration_tests
```

Build the documentation:

```bash
cd datacube-core/docs
make html
```

Then open `_build/html/index.html` in your browser to view the Documentation.

### 18.2 Windows Developer Setup

Base OS: Windows 10

This guide will setup an ODC core development environment and includes:

- Anaconda python using conda environments to isolate the odc development environment
- installation of required software and useful developer manuals for those libraries
- Postgres database installation with a local user configuration
• Integration tests to confirm both successful development setup and for ongoing testing
• Build configuration for local ODC documentation

18.2.1 Required software

Postgres:

Download and install from here.

18.2.2 Python and packages

Python 3.5+ is required. Python 3.6 is recommended.

Anaconda Python

Install Anaconda Python

Add conda-forge to package channels:

```bash
conda config --add channels conda-forge
```

Conda Environments are recommended for use in isolating your ODC development environment from your system installation and other python environments.

Install required python packages and create an odc conda environment.

Python 3.6:

```bash
conda env create -n odc --file .travis/environment.yaml sphinx
```

Activate odc python environment:

```bash
activate odc
```

18.2.3 Postgres database configuration

This configuration supports local development using your login name.

18.2.4 Open Data Cube source and development configuration

Download the latest version of the software from the repository

```bash
git clone https://github.com/opendatacube/datacube-core
cd datacube-core
```

We need to specify the database user and password for the ODC integration testing. To do this:

```bash
copy integration_tests\agdcintegration.conf %HOME%\datacube_integration.conf
```

Then edit the %HOME%\datacube_integration.conf with a text editor and add the following lines replacing <foo> with your username and <foobar> with the database user password you set above (not the postgres one, your <foo> one):
18.2.5 Verify it all works

Run the integration tests:

```
cd datacube-core
pytest
```

Build the documentation:

```
cd datacube-core/docs
make html
open _build/html/index.html
```

18.3 Mac OSX Developer Setup

Under construction

Base OS: Mac OSX

This guide will setup an ODC core development environment and includes:

- Anaconda python using conda environments to isolate the odc development environment
- installation of required software and useful developer manuals for those libraries
- Postgres database installation with a local user configuration
- Integration tests to confirm both successful development setup and for ongoing testing
- Build configuration for local ODC documentation

18.3.1 Required software

Postgres:

Download and install the EnterpriseDB distribution from here

18.3.2 Python and packages

Python 3.5+ is required. Python 3.6 is recommended.

Anaconda Python

Install Anaconda Python
Add conda-forge to package channels:
conda config --add channels conda-forge

Conda Environments are recommended for use in isolating your ODC development environment from your system installation and other python environments.

Install required python packages and create an odc conda environment.

Python 3.6:

conda env create -n odc --file .travis/environment.yaml sphinx

Activate odc python environment:

source activate odc

18.3.3 Postgres database configuration

This configuration supports local development using your login name.

If this is a new installation of Postgres on your system it is probably wise to set the postgres user password. As the local “postgres” Linux user, we are allowed to connect and manipulate the server using the psql command.

In a terminal, type:

```bash
sudo -u postgres psql postgres
```

Set a password for the “postgres” database role using the command:

```bash
\password postgres
```

and set the password when prompted. The password text will be hidden from the console for security purposes.

Type Control+D or q to exit the posgreSQL prompt.

By default in Ubuntu, Postgresql is configured to use ident sameuser authentication for any connections from the same machine which is useful for development. Check out the excellent Postgresql documentation for more information, but essentially this means that if your Ubuntu username is foo and you add foo as a Postgresql user then you can connect to a database without requiring a password for many functions.

Since the only user who can connect to a fresh install is the postgres user, here is how to create yourself a database account (which is in this case also a database superuser) with the same name as your login name and then create a password for the user:

```bash
sudo -u postgres createuser --superuser $USER
sudo -u postgres psql
postgres=# \password $USER
```

Now we can create an agdcintegration database for testing:

```bash
createdb agdcintegration
```

Connecting to your own database to try out some SQL should now be as easy as:

```bash
psql -d agdcintegration
```
18.3.4 Open Data Cube source and development configuration

Download the latest version of the software from the repository:

```
git clone https://github.com/opendatacube/datacube-core
cd datacube-core
```

We need to specify the database user and password for the ODC integration testing. To do this:

```
cp integration_tests/agdcintegration.conf ~/.datacube_integration.conf
```

Then edit the `~/.datacube_integration.conf` with a text editor and add the following lines replacing `<foo>` with your username and `<foobar>` with the database user password you set above (not the postgres one, your `<foo>` one):

```
[datacube]
db_hostname: localhost
db_database: agdcintegration
db_username: <foo>
db_password: <foobar>
```

18.3.5 Verify it all works

Run the integration tests:

```
cd datacube-core
./check-code.sh integration_tests
```

Build the documentation:

```
cd datacube-core/docs
make html
```

Then open `_build/html/index.html` in your browser to view the Documentation.
Virtual Products

19.1 Introduction

Virtual products enable ODC users to declaratively combine data from multiple products and perform on-the-fly computation while the data is loaded. The workflow is deduced from a lightweight configuration that can help datacube optimize the query to avoid loading data that would be eventually discarded.

An example virtual product would be a cloud-free surface reflectance (SR) product derived from a base surface reflectance product and a pixel quality (PQ) product that classifies cloud. Virtual products are especially useful when the datasets from the different products have the same spatio-temporal extent and the operations are to be applied pixel-by-pixel.

Functionalities related to virtual products are mainly in the `datacube.virtual` module.

19.2 Design

Currently, virtual products are constructed by applying a fixed set of combinators to either existing products or other virtual products. That is, a virtual product can be viewed as a tree whose nodes are combinators and leaves are ordinary datacube products.

Continuing the example in the previous section, consider the configuration (or the “recipe”) for a cloud-free SR product from SR products for two sensors (`ls7_nbar_albers` and `ls8_nbar_albers`) and their corresponding PQ products (`ls7_pq_albers` and `ls8_pq_albers`):

```python
from datacube.virtual import construct_from_yaml

cloud_free_ls_nbar = construct_from_yaml(""
    collate:
    - transform: apply_mask
      mask_measurement_name: pixelquality
      input:
        juxtapose:
    ")
```

(continues on next page)
- product: ls7_nbar_albers
  measurements: [red, green, blue]
- transform: make_mask
  input:
    product: ls7_pq_albers
  flags:
    blue_saturated: false
    cloud_acca: no_cloud
    cloud_fmask: no_cloud
    cloud_shadow_acca: no_cloud_shadow
    cloud_shadow_fmask: no_cloud_shadow
    contiguous: true
    green_saturated: false
    nir_saturated: false
    red_saturated: false
    swir1_saturated: false
    swir2_saturated: false
    mask_measurement_name: pixelquality
- transform: apply_mask
  mask_measurement_name: pixelquality
  input:
    juxtapose:
      - product: ls8_nbar_albers
        measurements: [red, green, blue]
      - transform: make_mask
        input:
          product: ls8_pq_albers
        flags:
          blue_saturated: false
          cloud_acca: no_cloud
          cloud_fmask: no_cloud
          cloud_shadow_acca: no_cloud_shadow
          cloud_shadow_fmask: no_cloud_shadow
          contiguous: true
          green_saturated: false
          nir_saturated: false
          red_saturated: false
          swir1_saturated: false
          swir2_saturated: false
          mask_measurement_name: pixelquality
      
"

The virtual product `cloud_free_ls_nbar` can now be used to load cloud-free SR imagery. The dataflow for loading the data reflects the tree structure of the recipe:

### 19.3 Grammar

Currently, there are four combinators for creating virtual products:

#### 19.3.1 product

The recipe to construct a virtual product from an existing datacube product has the form:
where *settings* can include `datacube.Datacube.load()` settings such as:

- measurements
- output_crs, resolution, align
- resampling
- group_by, fuse_func

The *product* nodes are at the leaves of the virtual product syntax tree.

## 19.3.2 2. collate

This combinator concatenates observations from multiple sensors having the same set of measurements. The recipe for a *collate* node has the form:

```json
{'collate': [<virtual-product-1>,
              <virtual-product-2>,
              ...
              ,
              <virtual-product-N>]
}
```

Observations from different sensors get interlaced:

Optionally, the source product of a pixel can be captured by introducing another measurement in the loaded data that consists of the index of the source product:

```json
{'collate': [<virtual-product-1>,
              <virtual-product-2>,
              ...
              ,
              <virtual-product-N>]
              ,
              'index_measurement_name': <measurement-name>
}
```

## 19.3.3 3. transform

This node applies an on-the-fly data transformation on the loaded data. The recipe for a *transform* has the form:

```json
{'transform': <transformation-class>,
           'input': <input-virtual-product>,
           **settings}
```

where the *settings* are keyword arguments to the initializer of the transformation class that implements the `datacube.virtual.Transformation` interface:

```python
class Transformation:
    def __init__(self, **settings):
        ''' Initialize the transformation object with the given settings. '''

    def compute(self, data):
        ''' xarray.Dataset -> xarray.Dataset '''

    def measurements(self, input_measurements):
```

19.3. Grammar
ODC has a (growing) set of built-in transformations:

- make_mask
- apply_mask
- to_float
- rename
- select

For more information on transformations, see *User-defined transformations*.

### 19.3.4 4. juxtapose

This node merges disjoint sets of measurements from different products into one. The form of the recipe is:

```python
{'juxtapose': [<virtual-product-1>,
              <virtual-product-2>,
              ...
              <virtual-product-N>]
}
```

Observations without corresponding entries in the other products will get dropped.

### 19.4 Using virtual products

Virtual products provide a common interface to query and then to load the data. The relevant methods are:

- `query(dc, **search_terms)` Retrieves datasets that match the `search_terms` from the database index of the datacube instance `dc`.

- `group(datasets, **search_terms)` Groups the datasets from `query` by the timestamps, and optionally restricts the region of interest. Does not connect to the database.

- `fetch(grouped, **load_settings)` Loads the data from the grouped datasets according to `load_settings`. Does not connect to the database. The on-the-fly transformations are applied at this stage. The resampling method or `dask_chunks` size can be specified in the `load_settings`.

Currently, virtual products also provide a `load(dc, **query)` method that roughly correspond to `dc.load`. However, this method exists only to facilitate code migration, and its extensive use is not recommended. It implements the pipeline:

For advanced use cases, the intermediate objects `VirtualDatasetBag` and `VirtualDatasetBox` may be directly manipulated.

### 19.5 User-defined transformations

Custom transformations must inherit from `datacube.virtual.Transformation`. If the user-defined transformation class is already installed in the Python environment the datacube instance is running from, the recipe may refer to it by its fully qualified name. Otherwise, for example for a transformation defined in a Notebook, the virtual product using the custom transformation is best constructed using the combinators directly.
For example, calculating the NDVI from a SR product (say, `ls8_nbar_albers`) would look like:

```python
from datacube.virtual import construct, Transformation, Measurement

class NDVI(Transformation):
    def compute(self, data):
        result = ((data.nir - data.red) / (data.nir + data.red))
        return result.to_dataset(name='NDVI')

    def measurements(self, input_measurements):
        return {'NDVI': Measurement(name='NDVI', dtype='float32', nodata=float('nan'), units='1')}

ndvi = construct(transform=NDVI, input=dict(product='ls8_nbar_albers', measurements=['red', 'nir']))
ndvi_data = ndvi.load(dc, **search_terms)
```

for the required geo-spatial `search_terms`. Note that the `measurement` method describes the output from the `compute` method.
Chapter 19. Virtual Products
20.1 Introduction

20.1.1 Assumptions and Design Constraints

On a HPC (High Performance Computing) system, the resources to provided to host the database may limited. During execution of a task across many compute nodes, the database should not be relied upon to serve concurrent access from all of the compute nodes.

The system must be able to support some particular mandated file and metadata formats on some platforms. E.g. NCI requires data be NetCDF-CF compliant.

3rd-party data can be accessed without being manipulated or reformatted.

Data of differing resolutions and projections can be used together. E.g. Landsat-MODIS blending.

20.2 High Level Architecture

20.2.1 Summary

The Open Data Cube (ODC) is an open source solution for accessing, managing, and analyzing large quantities of Geographic Information System (GIS) data - namely Earth observation (EO) data. It presents a common analytical framework composed of a series of data structures and tools which facilitate the organization and analysis of large gridded data collections. The Open Data Cube was developed for the analysis of temporally-rich earth observation data, however the flexibility of the platform also allows other gridded data collections to be included and analyzed. Such data may include elevation models, geophysical grids, interpolated surfaces and model outputs. A key characteristic of the Open Data Cube is that every unique observation is kept, which contrasts with many other methods used to handle large gridded data collections. Some of the major advantages of ODC are the following:

- Flexible framework
- User maintains control and ownership over their data
• Paradigm shift from scene-based analysis to pixel based
• Lower barrier to entry for remote sensing data analysis.

In this section, we briefly describe and illustrate the high-level architecture and ecosystem of the ODC framework in order to provide a better understanding to those who are new to ODC. This document only covers major components of the ODC and the relationships between them.

20.2.2 High-Level ODC Overview

The ODC core serves as a layer between satellite data and end user applications.

It provides a common analytical framework to allow multiple data sources to produce information for multiple uses. The ODC can handle data from any satellite data provider. The ODC eliminates the need for difficult and time-consuming pre-processing of the data from individual applications. This allows an increased capacity for development of information products by the Earth Observation (EO) community, and increased value for the public from EO information. Figure 1 illustrates data from many satellite data providers being managed by an ODC system.

Figure 1: High-Level Open Data Cube Overview

Several international space agencies provide data and make provisions to supply this data in an Analysis Ready Data (ARD) format for immediate application.

Figure 1 illustrates a diverse set of data being managed by an ODC core system. The ODC core system is then used as a simplified basis on which end users conduct analysis using ODC compatible analysis tools.

20.2.3 High-Level ODC Ecosystem

As stated earlier, the ODC core serves as a layer between satellite data providers and applications. A set of open source tools exist to help scientists conduct research using data managed by the ODC.

Figure 2 illustrates popular tools used within the community that utilizes the ODC Core as its basis:
• Command Line Tools: A tool used by programmers/developers to interface with the ODC.

• Open Data Cube Explorer: A visual and interactive web application that lets users explore their inventory of available data.

• Open Data Cube Stats: An optimized means of defining and executing advanced analysis on ODC system. This tool is oriented towards scientists.

• Web User Interface (UI): A web application that allows developers to interactively showcase and visualize the output of algorithms.

• Jupyter Notebooks: Research documents centered around techniques in EO sciences. A notebook contains executable code detailing examples of how the data cube is used in a research setting, and therefore is an invaluable reference material for new users.

• Open Geospatial Consortium (OGC) Web Services: Adapters that can connect non-ODC applications to the ODC.

![High-Level ODC Ecosystem](image)

*Figure 2: High-Level ODC Ecosystem*

### 20.2.4 Download Data Locally and Index

In the previous section, Figure 2 showed that the ODC framework can make data accessible to a sizeable ecosystem of applications. The following section briefly covers a process called indexing. Described plainly, indexing is about making the ODC aware of the existence of imagery. In the process of indexing data, the ODC tracks information that is useful when it comes to loading imagery, searching for imagery, or performing more advanced operations such as realigning imagery. The data can be either downloaded locally or stored in the cloud. In this section, we describe the process of indexing where data is downloaded locally.
Here is a brief outline of the indexing process:

1. As shown in Figure 3, the first step in this process is to describe the source of the imagery. We include basic details about which sensor the data comes from, what format to expect the data in, as well as its measurements, e.g. bands. This is done by drafting a document called a product definition for each data type. This product definition is then added to the system. Adding a product definition enables the system to accept that product.

2. The second step in the process is about extracting details from an individual satellite image. This is called the data preparation step. Scripts are available to extract information or metadata from many types of images.

3. The data extracted in step 2 typically includes date and time of acquisition, spatial bounds, etc. as metadata. In the third step, called indexing, metadata (documents) are indexed into the ODC’s database. Most importantly, the process stores the location of the data within a local system.

Figure 3: Download Data Locally and Index

20.2.5 Download Data Locally and Ingest

In the previous section, we briefly touched on the process of indexing. In this section, we will discuss the process called ingestion. Ingestion is a process that takes indexed data and performs some operations to turn it into a new file format or structure. This optimization step can increase the efficiency of data storage and retrieval. For example, there are significant improvements when converting downloaded GeoTIFFs to a format like NetCDF. Ingestion also splits large scene files into several smaller tiles to help organize large multidimensional data-sets for both fast and flexible data access. Geospatial transformations can also be defined in the ingestion process. The ingestion process can be configured using an ingestion configuration to reformat and apply geospatial transforms to the data. Figure 4 illustrates the ingestion process. The indexed imagery is transformed and reformatted per ingestion configuration specifications. It may involve re-sampling, re-projection, repackaging, re-compression, etc. The newly formatted/transformed data is re-indexed in the database under a new product name that gets registered in the database.
User-supplied query parameters are used as a lookup into the metadata database in order to determine which datasets hold data requested by the user. Those datasets are then grouped and ordered, and the actual data is loaded from the file system. The resulting data is organized into an Xarray Dataset with appropriate temporal-spatial dimensions and separate data variables for each band.
20.3 Data Model

20.3.1 Dataset

“The smallest aggregation of data independently described, inventoried, and managed.”

—Definition of “Granule” from NASA EarthData Unified Metadata Model

Examples of ODC Datasets:

- a Landsat Scene
- an Albers Equal Area tile portion of a Landsat Scene

20.3.2 Product

Products are collections of datasets that share the same set of measurements and some subset of metadata.
20.3.3 Metadata Types

Metadata Types define custom index search fields across products. The default eo metadata type defines fields such as ‘platform’, ‘instrument’ and the spatial bounds.
Fig. 1: Sequence of steps when creating an index
20.3.4 How the Index Works

20.4 Metadata Index

20.4.1 Dataset Interface

Add/update/archive datasets Search for datasets Manage locations (i.e. file path) Basic stats (Count of datasets)

20.4.2 Implementations

Postgres

The current implementation uses Postgres 9.5, with JSONB support.

The \textit{metadata\_types} are used to create database indexes into the JSONB dataset documents.

Future Possiblities

- Spatial:
  - PostGIS - PostgreSQL with optimised geometry datatypes and operations
- Lightweight:
  - SQLite
  - File-based (eg YAML)
- Enterprise:
  - DynamoDB
- Remote:
  - OGC Catalog Service-Web (CS-W)
  - NASA Common Metadata Repository
  - Radiant Earth Spatial Temporal Asset Metadata

20.4.3 Problems

Spatial Support

Currently the index stores spatial regions for the data, but indexes it on a ranges of latitudes and longitudes.

A database with support for spatial objects, such as the PostGIS extension for Postgres, could improve the efficiency (and simplicity of implementation) of spatial queries.

Pre-computed Summaries

We don’t currently store the spatial and temporal extents of a product.

To calculate this in the database requires scanning the entire dataset table to get min and max extent values. More commonly, to do this in Python, every dataset record is retrieved which is a very memory and CPU intensive operation.
This is an important feature for apps such as *datacube-wms* and *cubedash* that need to know the entire bounds for sensibly displaying the user interface.

### Replication

Syncing the index across systems (e.g. from NCI to an external system) requires a standard interface. There are issues using Postgres tools that require locking tables, etc, that need to be investigated.

### Publication

There is no standard way to access the *Index* from remote systems. Directly using the database exposes implementation specifics. We could possibly use an existing protocol, such as:

- Various OGC standards, such as CS-W, WCS2-EO or WFS2
- NASA Common Metadata Repository
- Radiant Earth Spatial Temporal Asset Metadata

## 20.5 Data Loading

### 20.5.1 Types of Data Loading

There are two major use-cases for loading data from the Datacube *Ad hoc access*, and *Large scale processing*. These are described below:

1. Ad hoc access
   - A small spatial region and time segment are chosen by the user
   - Data is expected to fit into RAM

2. Large scale processing (*GridWorkflow*)
   - Continental scale processing
   - Used to compute new products or to perform statistics on existing data
   - Often unconstrained spatially
   - Often unconstrained along time dimension
   - Data is accessed using regular grid in *small enough* chunks
   - The specific access pattern is algorithm/compute environment dependent and is supplied by the user and requires manual tuning

### Ad hoc data access

Typically a small spatial region and time range are chosen by the user, and all of the returned Data is expected to fit into RAM.

One database query maps to one *xarray.Dataset* which is processed/displayed/analyzed by custom user code. This happens as a two step process:

1. Build a Virtual Storage Resource (VSR) – a description of what data needs to be loaded, possibly from many disparate sources
2. Load data from the VSR into a contiguous memory representation `xarray.Dataset`.

Building the VSR involves querying the database to find all possible `storage units` that might contribute to the ROI (region of interest, x,y,t) and performing various post processing on the result:

- Possibly pruning results somewhat based on `valid data` geo-polygons, or any other metric
- Grouping result by time (i.e. assembling several VSR2D into VSR3D)

**Code refs:** `find_datasets()`, `group_datasets()`, `datacube.model.Dataset`

Once a VSR is built it can be loaded into memory either as a whole or small portions at a time.

**Code refs:** `load_data()`, `GeoBox`

### Large scale processing data access

Just like in the ad hoc scenario described above there are two steps. One involves querying the database in order to build a VSR, and the second step is loading data. The difference is in the way the VSR is built. Rather than constructing one giant VSR covering the entire collection a large number of VSRs are constructed each covering a non-overlapping region (one of the cells on a grid). Rather than querying the database once for each grid cell, a single query is performed and the result is then binned according to the `grid spec`.

Data loading happens in exactly the same way as in the ad hoc approach, except it usually happens in parallel across multiple processing nodes.

### 20.5.2 Data Structures

**Virtual Storage Resource**

- **Virtual** as opposite of Real/Physical, meaning constructed on the fly as opposed to read from database or file. `Logical` is another name often used for this kind of thing
- **Storage** as in just container of data, no possibility for compute beyond maybe projection changes, not specific to raster data
- **Resource** as in `URI`, `URL`, possibly file on some local/network file system, but could be S3, HTTP, FTP, OPeNDAP, etc.

Provides a unified view of a collection of disparate storage resources.

At the moment there is no actual `Virtual Storage Resource` class instead we use

- VSR3D is an `xarray.Dataset` that has a time dimension and contains a VSR2D for every timestamp
- VSR2D is a list of `datacube.model.Dataset`
- `datacube.model.Dataset` aggregates multiple bands into one storage resource. It is stored in the database and is used for provenance tracking.

All the information about individual `storage units` is captured in the `datacube.model.Dataset`, it includes:

- Mapping from band names to underlying files/URIs
- Geo-spatial info: CRS, extent
- Time range covered by the observation
- Complete metadata document (excluding lineage data)

It's important to note that `datacube.model.Dataset` describes observations for one timeslice only.
**TODO**: describe issues with timestamps, each pixel has its own actual capture time, which we do not store or track, but it does mean that single time slice is not just a point in time, but rather an interval.

The relationship between `datacube.model.Dataset` and storage units is complex, it's not one to one, nor is one to many. Common scenarios are listed below:

1. `datacube.model.Dataset` refers to several GeoTiff files, one for each band. Each GeoTiff file is referenced by exactly one dataset.
2. `datacube.model.Dataset` refers to one netCDF4 file containing single timeslice, all bands are stored in that one file. NetCDF4 file is referenced by one dataset.
3. `datacube.model.Dataset` refers to one time slice within a *stacked* netCDF4 file. This same netCDF4 file is referenced by a large number of datasets, each referring to a single time slice within the file.

It is assumed that individual storage units within a `datacube.model.Dataset` are of the same format. In fact storage format is usually shared by all datasets belonging to the same Product, although it is possible to index different formats under one product.

### 20.5.3 Data load in detail

VSR, GeoBox, [bands of interest, opts] → pixel data

Once you have VSR constructed you can load all or part of it into memory using `load_data()`. At this point users can customise which bands they want, how to deal with overlapping data, and other options like a per band re-sampling strategy can also be supplied.

**Internal interfaces**

The primary internal interface for loading data from storage is `datacube.storage.storage.BandDataSource`, unfortunately this rather generic name is taken by the specific implementation based on the rasterio library. `datacube.storage.storage.BandDataSource` is responsible for describing data stored for a given band, one can query:

- The Shape (in pixels) and data type
- Geospatial information: CRS + Affine transform

and also provides access to pixel data via 2 methods

- `read()`: access a section of source data in native projection but possibly in different resolution
- `reproject()`: access a section of source data, re-projecting to an arbitrary projection/resolution

This interface follows very closely the interface provided by the rasterio library. Conflating the reading and transformation of pixel data into one function is motivated by the need for efficient data access. Some file formats support multi-resolution storage for example, so it is more efficient to read data at the appropriate scale rather than reading highest resolution version followed by down sampling. Similarly re-projection can be more memory efficient if source data is loaded in smaller chunks interleaved with raster warping execution compared to a conceptually simpler but less efficient *load all then warp all* approach.

**Code refs**: `load_data()`, GeoBox, BandDataSource, RasterDatasetDataSource

### 20.5.4 Fuse function customisation

A VSR2D might consist of multiple overlapping pixel planes. This is either due to duplicated data (e.g. consecutive Landsat scenes include a north/south overlap, and all derived products keep those duplicates) or due to grouping using
a larger time period (e.g. one month). Whatever the reason, the overlap needs to be resolved when loading data since the user expects a single plane of pixels.

The strategy for dealing with overlapping data can be supplied by the user at the load time. The default strategy is to simply pick the first observed valid pixel value, where any pixel that is different from the nodata value is considered valid. In situations where pixel validity is defined by a more complex metric, one can supply a custom fuse function. Fuse function takes two pixel planes (numpy.ndarray) of the same shape and data type, the first contains fused result so far, and the second one is the new data. The fuse function is expected to update fused result so far with the new data in place.

Below is a pseudo-code of the load code that uses a fuse function (reproject_and_fuse() is the actual implementation).

```python
dst = ndarray_filled_with_nodata_values()
for ds in datasets_for_this_timeslot:
    new_data = get_the_data_in_the_right_projection(ds)
    # tmp and dst have the same shape and dtype
    fuse(dst, new_data)  ## << update dst in place
```

**Code refs:** reproject_and_fuse(), _fuse_measurement(), load_data()

**Problems with the current approach to fusing**

One major limitation is that the fuse function is customised per product, but should really be customised per band. It is completely reasonable for different bands of the same product to be sufficiently different as to require a different fusing strategy. And since a fuse function doesn’t know which band it is processing it can not dispatch to different implementations internally.

The types of computation a fuse function can perform is limited by the interface, for example one can not implement average nor median. With some modification it should be possible to support arbitrary incremental computations, like average, without loading all the data at once.

**20.5.5 Lazy load with dask**

In computer science context lazy means roughly not computed until needed. Rather then loading all the data immediately load_data() function can instead construct an xarray.Dataset that the user can use in the same way as a fully loaded data set, except that pixel data will be fetched from disk/network on demand as needed. The on-demand loading functionality is provided by third party libraries xarray and dask(used internally by xarray). Datacube code constructs a recipe for loading data on demand, this recipe is executed as needed by xarray/dask library when real data is required to be loaded for the first time.

**Note:** TODO

- Discuss chunks and how they relate to on-disk storage chunks
- Discuss memory management, how data is unloaded from RAM, avoiding out of memory errors when processing large arrays.
- We need to provide a clear guidance as to when this mode should be used and how
20.5.6 Limitations and problems

One of the original goals of Datacube is to support a wide variety of different input data sources, as such flexibility has been preferred to efficiency. When designing an API one would strive for simplicity, generality and efficiency. An “Ideal API” would have all three turned up to the max, but often it is necessary to balance one at the expense of the other. Efficiency in particular often has significant complexity costs, it is also harder to achieve when striving to be as generic as possible.

Internal interfaces for reading data is per time slice per band. Description of a storage unit for a given band for a given time slice (datacube.model.Dataset) is passed from the database to storage specific loading code one by one, and the results are assembled into a 3D structure by generic loading code.

On a plus side this maps nicely to the way things work in gdal/rasterio land and is the most generic representation that allows for greatest variety of storage regimes

- bands/time slices split across multiple files
- bands stored in one files, one file per time slice
- stacked files that store multiple time slices and all the bands

On the other hand this way of partitioning code leads to less than optimal I/O access patterns. This is particularly noticeable when using “stacked files” (a common use case on the NCI installation of the datacube) while doing “pixel drill” type of access.

Problems are:

- Same netCDF file is opened/closed multiple times – no netCDF chunk cache sharing between reads
- Larger more complex (many bands) files might have slightly larger “open overhead” to begin with, not a problem if you share the same file handle to load all the data of interest, but adds to a significant cost when you re-open the same file many times needlessly.
- File open overhead increases as we move towards cloud storage solutions like Amazon S3.
- Chunking along time dimension makes depth reads even more costly when using this access pattern since data is read and decompressed just to be thrown away (in the case of NCI install, chunking along time dimension is 5 time slices per chunk, so 80% of decoded data is thrown away due to access pattern, since we only read one time slice at a time).

Possible Solutions

One possible solution is to keep internal interfaces as they are and introduce global IO cache to allow sharing of opened files/partially loaded data. This adds quite a bit of complexity, particularly around memory management: can’t just keep adding data to the cache, need to purge some data eventually, meaning that depending on the use pattern efficiency improvements aren’t guaranteed. Global state that such a solution will need to rely on is problematic in the multi-threaded environment and often leads to hard to debug errors even in a single threaded application. Global state makes testing harder too.

As such we believe that a more practical approach is to modify internal IO interfaces to support efficient reads from stacked multi-band storage. To do that we need to move internal interface boundary up to VSR3D level, VSR in xarray.Dataset out.

We propose roughly the following interface

1. open :: VSR, [output CRS, output scale, opts] -> VSRDataSource
2. read :: VSRDataSource, [GeoBox, bands of interest, time of interest, opts] -> xarray.Dataset
A two step process, first construct pixel data source supplying ahead of time output projection and scale (optional, defaulting to native projection and resolution when possible), then read sections of data as needed, user can choose what spatio-temporal region they want to access and select a subset of bands they need to read into memory. Note that read might perform re-projection under the hood, based on whether output projection/resolution was supplied and whether it differs from native.

### 20.5.7 Storage Drivers

**GDAL**

The GDAL-based driver uses `rasterio` to read a single time slice of a single variable/measurement at a time, in a synchronous manner.

**S3IO**

This driver provides access to chunked array storage on Amazon S3.

### 20.5.8 Supporting Diagrams

**Data Read Process**

**Storage Classes**

### 20.6 Extending Datacube

Beyond the configuration available in ODC, there are three extension points provided for implementing different types of data storage and indexing.

- Drivers for Reading Data
- Drivers for Writing Data
- Alternative types of Index

#### 20.6.1 Support for Plug-in drivers

A light weight implementation of a driver loading system has been implemented in `datacube/drivers/driver_cache.py` which uses `setuptools` dynamic service and plugin discovery mechanism to name and define available drivers. This code caches the available drivers in the current environment, and allows them to be loaded on demand, as well as handling any failures due to missing dependencies or other environment issues.

#### 20.6.2 Data Read Plug-ins

**Entry point group** `datacube.plugins.io.read`.

Read plug-ins are specified as supporting particular `uri protocols` and `formats`, both of which are fields available on existing `Datasets`.

A `ReadDriver` returns a `DataSource` implementation, which is chosen based on:

- Dataset URI protocol, eg. `s3://`
Fig. 2: Current Data Read Process

Fig. 3: Classes currently implementing the DataCube Data Read Functionality
- Dataset format. As stored in the Data Cube Dataset.
- Current system settings
- Available IO plugins

If no specific DataSource can be found, a default datacube.storage.storage.RasterDatasetDataSource is returned, which uses rasterio to read from the local file system or a network resource.

The DataSource maintains the same interface as before, which works at the individual dataset+time+band level for loading data. This is something to be addressed in the future.

**Example code to implement a reader driver**

```python
def init_reader_driver():
    return AbstractReaderDriver()

class AbstractReaderDriver(object):
    def supports(self, protocol: str, fmt: str) -> bool:
        pass
    def new_datasource(self, band: BandInfo) -> DataSource:
        return AbstractDataSource(band)

class AbstractDataSource(object):  # Same interface as before
    ...
```

**S3 Driver**

URI Protocol s3://
Dataset Format aio
Implementation location datacube/drivers/s3/driver.py

**Example Pickle Based Driver**

Available in /examples/io_plugin. Includes an example setup.py as well as an example Read and Write Drivers.

**20.6.3 Data Write Plug-ins**

Entry point group datacube.plugins.io.write

Are selected based on their name. The storage.driver field has been added to the ingestion configuration file which specifies the name of the write driver to use. Drivers can specify a list of names that they can be known by, as well as publicly defining their output format, however this information isn’t used by the ingester to decide which driver to use. Not specifying a driver counts as an error, there is no default.

At this stage there is no decision on what sort of a public API to expose, but the write_dataset_to_storage() method implemented in each driver is the closest we’ve got. The ingester is using it to write data.
Example code to implement a writer driver

```python
def init_writer_driver():
    return AbstractWriterDriver()

class AbstractWriterDriver(object):
    @property
    def aliases(self):
        return []  # List of names this writer answers to

    @property
    def format(self):
        return ''  # Format that this writer supports

    def write_dataset_to_storage(self, dataset, filename,
        global_attributes=None,
        variable_params=None,
        storage_config=None,
        **kwargs):
        ...
        return {}  # Can return extra metadata to be saved in the index with the dataset
```

NetCDF Writer Driver

- **Name**: netcdf, NetCDF CF
- **Format**: NetCDF
- **Implementation**: datacube.drivers.netcdf.driver.NetcdfWriterDriver

S3 Writer Driver

- **Name**: s3aio
- **Protocol**: s3
- **Format**: aio
- **Implementation**: datacube.drivers.s3.driver.S3WriterDriver

20.6.4 Index Plug-ins

**Entry point group**: datacube.plugins.index

A connection to an Index is required to find data in the Data Cube. Already implemented in the `develop` branch was the concept of environments which are a named set of configuration parameters used to connect to an Index. This PR extends this with an `index_driver` parameter, which specifies the name of the Index Driver to use. If this parameter is missing, it falls back to using the default PostgreSQL Index.

Example code to implement an index driver
def index_driver_init():
    return AbstractIndexDriver()

class AbstractIndexDriver(object):
    @staticmethod
    def connect_to_index(config, application_name=None, validate_connection=True):
        return Index.from_config(config, application_name, validate_connection)

Default Implementation

The default Index uses a PostgreSQL database for all storage and retrieval.

S3 Extensions

The datacube.drivers.s3aio_index.S3AIOIndex driver subclasses the default PostgreSQL Index with support for saving additional data about the size and shape of chunks stored in S3 objects. As such, it implements an identical interface, while overriding the dataset.add() method to save the additional data.

20.6.5 Drivers Plugin Management Module

Drivers are defined in setup.py -> entry_points:

```python
entry_points={
    'datacube.plugins.io.read': [  
        's3aio = datacube.drivers.s3.driver:reader_driver_init'
    ],
    'datacube.plugins.io.write': [  
        'netcdf = datacube.drivers.netcdf.driver:writer_driver_init',
        's3aio = datacube.drivers.s3.driver:writer_driver_init',
        's3aio_test = datacube.drivers.s3.driver:writer_test_driver_init',
    ],
}
```

20.6.6 Data Cube Drivers API

This module implements a simple plugin manager for storage and index drivers.


Returns a newly constructed data source to read dataset band data.

An appropriate DataSource implementation is chosen based on:

- Dataset URI (protocol part)
- Dataset format
- Current system settings
- Available IO plugins

This function will return the default RasterDatasetDataSource if no more specific DataSource can be found.

Parameters
• **dataset** – The dataset to read.

• **band_name** *(str)* – the name of the band to read.

datacube.drivers.storage_writer_by_name(*name*)

  Lookup writer driver by name

  **Returns**  Initialised writer driver instance

  **Returns**  None if driver with this name doesn’t exist

adatacube.drivers.index_driver_by_name(*name*)

  Lookup writer driver by name

  **Returns**  Initialised writer driver instance

  **Returns**  None if driver with this name doesn’t exist

adatacube.drivers.index_drivers()

  Returns list driver names

adatacube.drivers.reader_drivers() → List[str]

  Returns list driver names

adatacube.drivers.writer_drivers() → List[str]

  Returns list driver names

### 20.6.7 References and History

• PR 346

• Pluggable Back Ends Discussion [7 December 2017]

• Teleconference with @omad @petewa @rtai @Kirill888 on 12 December 2017.

• Notes from ODC Storage and Index Driver Meeting
CHAPTER 21

Release Process

21.1 Build a version

1. **Pick a release name for the next version** Releases are versioned using the `major.minor.bugfix` numbering system.

2. **Update the release notes on the What’s New page** Check the git log for changes since the last release.

3. **Check that Travis and readthedocs are passing for the latest commit** Make sure that the tests have finished running!

4. **Tag the branch** Use the format of `datacube-major.minor.bugfix`.

   ```
   git tag datacube-1.6.0
   git push --tags
   ```

5. **Draft a new release on the Datacube releases GitHub page** Include the items added to the release notes in step 2.

21.2 Marking it stable

Once a built version has been tested, found to be stable, and the team agrees, we make it the new stable version.

1. **Merge changes leading up to the release into the stable branch** This will also update the stable docs.

2. **Upload the build to PyPi** You might need a PyPi account with appropriate authorization.

   ```
   python setup.py sdist bdist_wheel
twine upload dist/*
   ```

   This should upload the project to https://pypi.python.org/pypi/datacube/.

3. **Update conda-forge recipe** Follow the instructions under Updating datacube-feedstock in the Datcube Feedstock repository.
It should involve modifying the version number in the recipe and updating the SHA hash. The hash should be generated from the `.tar.gz` mentioned in the source of the recipe.

```bash
openssl sha256 <downloaded-datacube-source.tar.gz>
```
CHAPTER 22

What's New

22.1 v1.7dev

- Allow specifying different resampling methods for different data variables of the same Product. (PR 551)
- Allow all resampling methods supported by rasterio. (PR 622)
- Bugfixes and improved performance of dask-backed arrays (PR 547)
- Improve API reference documentation.
- Bug fix (Index out of bounds causing ingestion failures)
- Add “virtual products” (PR 522, PR 597) for multi-product loading
- Support indexing data directly from HTTP/HTTPS/S3 URLs (PR 607)

22.2 v1.6.1 (27 August 2018)

Correction release. By mistake, v1.6.0 was identical to v1.6rc2!

22.3 v1.6.0 (23 August 2018)

- Enable use of aliases when specifying band names
- Fix ingestion failing after the first run (PR 510)
- Docker images now know which version of ODC they contain (PR 523)
- Fix data loading when nodata is NaN (PR 531)
- Allow querying based on python datetime.datetime objects. (PR 499)
• **Require** [rasterio 1.0.2](https://rasterio.readthedocs.io/en/stable/) or higher, which fixes several critical bugs when loading and reprojecting from multi-band files.

• **Assume fixed paths for** `id` and `sources` metadata fields ([issue 482](https://github.com/OSGeo/arcgisdesktop/issues/482))

• `datacube.model.Measurement` was put to use for loading in attributes and made to inherit from `dict` to preserve current behaviour. ([PR 502](https://github.com/datacube-open-source/datacube-core/pull/502))

• **Updates when indexing data with** `datacube dataset add` ([See PR 485](https://github.com/datacube-open-source/datacube-core/pull/485), [issue 451](https://github.com/datacube-open-source/datacube-core/issues/451) and [issue 480](https://github.com/datacube-open-source/datacube-core/issues/480))
  - Allow indexing without lineage `datacube dataset add --ignore-lineage`
  - Removed the `--sources-policy=skip|verify|ensure`. Instead use `--[no-]auto-add-lineage` and `--[no-]verify-lineage`
  - New option `datacube dataset add --exclude-product <name>` allows excluding some products from auto-matching

• Preliminary API for indexing datasets ([PR 511](https://github.com/datacube-open-source/datacube-core/pull/511))

• Enable creation of MetadataTypes without having an active database connection ([PR 535](https://github.com/datacube-open-source/datacube-core/pull/535))

### 22.4 v1.6rc2 (29 June 2018)

#### 22.4.1 Backwards Incompatible Changes

• **The** `helpers.write_geotiff()` function has been updated to support files smaller than 256x256. It also no longer supports specifying the time index. Before passing data in, use `xarray_data.isel(time=<my_time_index>).` ([PR 277](https://github.com/datacube-open-source/datacube-core/pull/277))

• Removed product matching options from `datacube dataset update` ([PR 445](https://github.com/datacube-open-source/datacube-core/pull/445)). No matching is needed in this case as all datasets are already in the database and are associated to products.

• Removed `--match-rules` option from `datacube dataset add` ([PR 447](https://github.com/datacube-open-source/datacube-core/pull/447))

• The seldom-used `stack` keyword argument has been removed from `Datcube.load`. ([PR 461](https://github.com/datacube-open-source/datacube-core/pull/461))

• The behaviour of the time range queries has changed to be compatible with standard Python searches (eg. time slice an `xarray`). Now the time range selection is inclusive of any unspecified time units. ([PR 440](https://github.com/datacube-open-source/datacube-core/pull/440))

**Example 1:** `time=('2008-01', '2008-03')` previously would have returned all data from the start of 1st January, 2008 to the end of 1st of March, 2008. Now, this query will return all data from the start of 1st January, 2008 and 23:59:59.999 on 31st of March, 2008.

**Example 2:** To specify a search time between 1st of January and 29th of February, 2008 (inclusive), use a search query like `time=('2008-01', '2008-02')`. This query is equivalent to using any of the following in the second time element:

```python
('2008-02-29')
('2008-02-29 23')
('2008-02-29 23:59')
('2008-02-29 23:59:59')
('2008-02-29 23:59:59.999')
```
22.4.2 Changes

- A `--location-policy` option has been added to the `datacube dataset update` command. Previously this command would always add a new location to the list of URIs associated with a dataset. It’s now possible to specify *archive* and *forget* options, which will mark previous location as archived or remove them from the index altogether. The default behaviour is unchanged. (PR 469)

- The masking related function `describe_variable_flags()` now returns a pandas DataFrame by default. This will display as a table in Jupyter Notebooks. (PR 422)

- Usability improvements in `datacube dataset [add|update]` commands (issue 447, issue 448, issue 398)
  - Embedded documentation updates
  - Deprecated `--auto-match` (it was always on anyway)
  - Renamed `--dtype` to `--product` (the old name will still work, but with a warning)
  - Add option to skip lineage data when indexing (useful for saving time when testing) (PR 473)

- Enable compression for metadata documents stored in NetCDFs generated by `stacker` and `ingestor` (issue 452)

- Implement better handling of stacked NetCDF files (issue 415)
  - Record the slice index as part of the dataset location URI, using `#part=<int>` syntax, index is 0-based
  - Use this index when loading data instead of fuzzy searching by timestamp
  - Fall back to the old behaviour when `#part=<int>` is missing and the file is more than one time slice deep

- Expose the following dataset fields and make them searchable:
  - `indexed_time` (when the dataset was indexed)
  - `indexed_by` (user who indexed the dataset)
  - `creation_time` (creation of dataset: when it was processed)
  - `label` (the label for a dataset)

(See PR 432 for more details)

22.4.3 Bug Fixes

- The `.dimensions` property of a product no longer crashes when product is missing a `grid_spec`. It instead defaults to `time,y,x`

- Fix a regression in v1.6rc1 which made it impossible to run `datacube ingest` to create products which were defined in 1.5.5 and earlier versions of ODC. (issue 423, PR 436)

- Allow specifying the chunking for string variables when writing NetCDFs (issue 453)

22.5 v1.6rc1 Easter Bilby (10 April 2018)

This is the first release in a while, and so there’s a lot of changes, including some significant refactoring, with the potential having issues when upgrading.
22.5.1 Backwards Incompatible Fixes

• Drop Support for Python 2. Python 3.5 is now the earliest supported Python version.
• Removed the old ndexpr, analytics and execution engine code. There is work underway in the execution engine branch to replace these features.

22.5.2 Enhancements

• Support for third party drivers, for custom data storage and custom index implementations
• The correct way to get an Index connection in code is to use datacube.index.index_connect()
• Changes in ingestion configuration
  – Must now specify the Data Write Plug-ins to use. For s3 ingestion there was a top level container specified, which has been renamed and moved under storage. The entire storage section is passed through to the Data Write Plug-ins, so drivers requiring other configuration can include them here. eg:

```python
... storage:
  ... driver: s3aio
  bucket: my_s3_bucket
...```

• Added a Dockerfile to enable automated builds for a reference Docker image.
• Multiple environments can now be specified in one datacube config. See PR 298 and the Runtime Config
  – Allow specifying which index_driver should be used for an environment.
• Command line tools can now output CSV or YAML. (Issue issue 206, PR 390)
• Support for saving data to NetCDF using a Lambert Conformal Conic Projection (PR 329)
• Lots of documentation updates:
  – Information about Bit Masking.
  – A description of how data is loaded.
  – Some higher level architecture documentation.
  – Updates on how to index new data.

22.5.3 Bug Fixes

• Allow creation of datacube.utils.geometry.Geometry objects from 3d representations. The Z axis is simply thrown away.
• The datacube --config_file option has been renamed to datacube --config, which is shorter and more consistent with the other options. The old name can still be used for now.
• Fix a severe performance regression when extracting and reprojecting a small region of data. (PR 393)
• Fix for a somewhat rare bug causing read failures by attempt to read data from a negative index into a file. (PR 376)
• Make CRS equality comparisons a little bit looser. Trust either a Proj.4 based comparison or a GDAL based comparison. (Closed issue 243)
22.5.4 New Data Support

- Added example prepare script for Collection 1 USGS data; improved band handling and downloads.
- Add a product specification and prepare script for indexing Landsat L2 Surface Reflectance Data (PR 375)
- Add a product specification for Sentinel 2 ARD Data (PR 342)

22.6 v1.5.4 Dingley Dahu (13th December 2017)

- Minor features backported from 2.0:
  - Support for \texttt{limit} in searches
  - Alternative lazy search method \texttt{find_lazy}
- Fixes:
  - Improve native field descriptions
  - Connection should not be held open between multi-product searches
  - Disable prefetch for celery workers
  - Support \texttt{jsonify-ing} decimals

22.7 v1.5.3 Purpler Unicorn with Starlight (16 October 2017)

- Use \texttt{cloudpickle} as the \texttt{celery} serialiser

22.8 v1.5.2 Purpler Unicorn with Stars (28 August 2017)

- Fix bug when reading data in native projection, but outside source area. Often hit when running \texttt{datacube-stats}
- Fix error loading and fusing data using \texttt{dask}. (Fixes issue 276)
- When reading data, implement \texttt{skip\_broken\_datasets} for the \texttt{dask} case too

22.9 v1.5.4 Dingley Dahu (13th December 2017)

- Minor features backported from 2.0:
  - Support for \texttt{limit} in searches
  - Alternative lazy search method \texttt{find_lazy}
- Fixes:
  - Improve native field descriptions
  - Connection should not be held open between multi-product searches
  - Disable prefetch for celery workers
  - Support \texttt{jsonify-ing} decimals
22.10 v1.5.3 Purpler Unicorn with Starlight (16 October 2017)

- Use cloudpickle as the celery serialiser
- Allow celery tests to run without installing it
- Move datacube-worker inside the main datacube package
- Write metadata_type from the ingest configuration if available
- Support config parsing limitations of Python 2
- Fix issue 303: resolve GDAL build dependencies on Travis
- Upgrade rasterio to newer version

22.11 v1.5.2 Purpler Unicorn with Stars (28 August 2017)

- Fix bug when reading data in native projection, but outside source area. Often hit when running datacube-stats
- Fix error loading and fusing data using dask. (Fixes issue 276)
- When reading data, implement skip_broken_datasets for the dask case too

22.12 v1.5.1 Purpler Unicorn (13 July 2017)

- Fix bug issue 261. Unable to load Australian Rainfall Grid Data. This was as a result of the CRS/Transformation override functionality being broken when using the latest rasterio version 1.0a9

22.13 v1.5.0 Purple Unicorn (9 July 2017)

22.13.1 New Features

- Support for AWS S3 array storage
- Driver Manager support for NetCDF, S3, S3-file drivers.

22.13.2 Usability Improvements

- When datacube dataset add is unable to add a Dataset to the index, print out the entire Dataset to make it easier to debug the problem.
- Give datacube system check prettier and more readable output.
- Make celery and redis optional when installing.
- Significantly reduced disk space usage for integration tests
- Dataset objects now have an is_active field to mirror is_archived.
- Added index.datasets.get_archived_location_times() to see when each location was archived.
22.14 v1.4.1 (25 May 2017)

- Support for reading multiband HDF datasets, such as MODIS collection 6
- Workaround for rasterio issue when reprojecting stacked data
- Bug fixes for command line arg handling

22.15 v1.4.0 (17 May 2017)

- Adds more convenient year/date range search expressions (see PR 226)
- Adds a simple replication utility (see PR 223)
- Fixed issue reading products without embedded CRS info, such as bom_rainfall_grid (see issue 224)
- Fixed issues with stacking and ncml creation for NetCDF files
- Various documentation and bug fixes
- Added CircleCI as a continuous build system, for previewing generated documentation on pull
- Require xarray >= 0.9. Solves common problems caused by losing embedded flag_def and crs attributes.

22.16 v1.3.1 (20 April 2017)

- Docs now refer to “Open Data Cube”
- Docs describe how to use conda to install datacube.
- Bug fixes for the stacking process.
- Various other bug fixes and document updates.

22.17 v1.3.0

- Updated the Postgres product views to include the whole dataset metadata document.
- datacube system init now recreates the product views by default every time it is run, and now supports Postgres 9.6.
- URI searches are now better supported from the cli: datacube dataset search uri = file://some/uri/here
- datacube user now supports a user description (via --description) when creating a user, and delete accepts multiple user arguments.
- Platform-specific (Landsat) fields have been removed from the default eo metadata type in order to keep it minimal. Users & products can still add their own metadata types to use additional fields.
- Dataset locations can now be archived, not just deleted. This represents a location that is still accessible but is deprecated.
- We are now part of Open Data Cube, and have a new home at https://github.com/opendatacube/datacube-core
This release now enforces the uri index changes to be applied: it will prompt you to rerun `init` as an administrator to update your existing cubes: `datacube -v system init` (this command can be run without affecting read-only users, but will briefly pause writes)

### 22.18 v1.2.2

- Added `--allow-exclusive-lock` flag to product add/update commands, allowing faster index updates when system usage can be halted.
- `{version}` can now be used in ingester filename patterns

### 22.19 v1.2.0 Boring as Batman (15 February 2017)

- Implemented improvements to `dataset search` and `info` cli outputs
- Can now specify a range of years to process to `ingest` cli (e.g. 2000-2005)
- Fixed `metadata_type update` not creating indexes (running `system init` will create missing ones)
- Enable indexing of datacube generated NetCDF files. Making it much easier to pull selected data into a private datacube index. Use by running `datacube dataset add selected.nc`
- Switch versioning system to increment the second digit instead of the third.

### 22.20 v1.1.18 Mushroom Milkshake (9 February 2017)

- Added `sources-policy` options to `dataset add` cli
- Multiple dataset search improvements related to locations
- Keep hours/minutes when grouping data by `solar_day`
- Code Changes: `datacube.model.{CRS,BoundingBox,Coordinate,GeoBox}` have moved into `datacube.utils.geometry`. Any code using these should update their imports.

### 22.21 v1.1.17 Happy Festivus Continues (12 January 2017)

- Fixed several issues with the geometry utils
- Added more operations to the geometry utils
- Updated `Code Recipes` to use geometry utils
- Enabled Windows CI (python 3 only)

### 22.22 v1.1.16 Happy Festivus (6 January 2017)

- Added `update` command to `datacube dataset` cli
- Added `show` command to `datacube product` cli
- Added `list` and `show commands` to `datacube metadata_type` cli
- Added ‘storage unit’ stacker application
- Replaced `model.GeoPolygon` with `utils.geometry` library

22.23 v1.1.15 Minion Party Hangover (1 December 2016)

- Fixed a data loading issue when reading HDF4_EOS datasets.

22.24 v1.1.14 Minion Party (30 November 2016)

- Added support for buffering/padding of GridWorkflow tile searches
- Improved the Query class to make filtering by a source or parent dataset easier. For example, this can be used to filter Datasets by Geometric Quality Assessment (GQA). Use `source_filter` when requesting data.
- Additional data preparation and configuration scripts
- Various fixes for single point values for lat, lon & time searches
- Grouping by solar day now overlays scenes in a consistent, northern scene takes precedence manner. Previously it was non-deterministic which scene/tile would be put on top.

22.25 v1.1.13 Black Goat (15 November 2016)

- Added support for accessing data through `http` and `s3` protocols
- Added `dataset search` command for filtering datasets (lists `id`, `product`, `location`)
- `ingestion_bounds` can again be specified in the ingester config
- Can now do range searches on non-range fields (e.g. `dc.load(orbit=(20, 30))`)
- Merged several bug-fixes from CEOS-SEO branch
- Added Polygon Drill recipe to `Code Recipes`

22.26 v1.1.12 Unnamed Unknown (1 November 2016)

- Fixed the affine deprecation warning
- Added `datacube metadata_type` cli tool which supports add and update
- Improved `datacube product` cli tool logging

22.27 v1.1.11 Unnamed Unknown (19 October 2016)

- Improved ingester task throughput when using distributed executor
- Fixed an issue where loading tasks from disk would use too much memory
- `model.GeoPolygon.to_crs()` now adds additional points (~every 100km) to improve reprojection accuracy
22.28 v1.1.10 Rabid Rabbit (5 October 2016)

- Ingester can now be configured to have WELD/MODIS style tile indexes (thanks Chris Holden)
- Added –queue-size option to datacube ingest to control number of tasks queued up for execution
- Product name is now used as primary key when adding datasets. This allows easy migration of datasets from one database to another
- Metadata type name is now used as primary key when adding products. This allows easy migration of products from one database to another
- DatasetResource.has() now takes dataset id insted of model.Dataset
- Fixed an issues where database connections weren’t recycled fast enough in some cases
- Fixed an issue where DatasetTypeResource.get() and DatasetTypeResource.get_by_name() would cache None if product didn’t exist

22.29 v1.1.9 Pest Hippo (20 September 2016)

- Added origin, alignment and GeoBox-based methods to model.GridSpec
- Fixed satellite path/row references in the prepare scripts (Thanks to Chris Holden!)
- Added links to external datasets in Indexing Data
- Improved archive and restore command line features: datacube dataset archive and datacube dataset restore
- Improved application support features
- Improved system configuration documentation

22.30 v1.1.8 Last Mammoth (5 September 2016)

- GridWorkflow.list_tiles() and GridWorkflow.list_cells() now return a Tile object
- Added resampling parameter to Datacube.load() and GridWorkflow.load(). Will only be used if the requested data requires resampling.
- Improved Datacube.load() like parameter behaviour. This allows passing in a xarray.Dataset to retrieve data for the same region.
- Fixed an issue with passing tuples to functions in Analytics Expression Language
- Added a user_guide section to the documentation containing useful code snippets
- Reorganized project dependencies into required packages and optional ‘extras’
- Added performance dependency extras for improving run-time performance
- Added analytics dependency extras for analytics features
- Added interactive dependency extras for interactivity features
22.31 v1.1.7 Bit Shift (22 August 2016)

- Added bit shift and power operators to Analytics Expression Language
- Added `datacube product update` which can be used to update product definitions
- Fixed an issue where dataset geo-registration would be ignored in some cases
- Fixed an issue where Execution Engine was using dask arrays by default
- Fixed an issue where int8 data could not sometimes be retrieved
- Improved search and data retrieval performance

22.32 v1.1.6 Lightning Roll (8 August 2016)

- Improved spatio-temporal search performance. `datacube system init` must be run to benefit
- Added `info`, `archive` and `restore` commands to `datacube dataset`
- Added `product-counts` command to `datacube-search` tool
- Made Index object thread-safe
- Multiple masking API improvements
- Improved database Index API documentation
- Improved system configuration documentation

22.33 v1.1.5 Untranslatable Sign (26 July 2016)

- Updated the way database indexes are partitioned. Use `datacube system init --rebuild` to rebuild indexes
- Added `fuse_data` ingester configuration parameter to control overlapping data fusion
- Added `--log-file` option to `datacube dataset add` command for saving logs to a file
- Added `index.datasets.count` method returning number of datasets matching the query

22.34 v1.1.4 Imperfect Inspiration (12 July 2016)

- Improved dataset search performance
- Restored ability to index telemetry data
- Fixed an issue with data access API returning uninitialized memory in some cases
- Fixed an issue where dataset center_time would be calculated incorrectly
- General improvements to documentation and usability
22.35 v1.1.3 Speeding Snowball (5 July 2016)

- Added framework for developing distributed, task-based application
- Several additional Ingester performance improvements

22.36 v1.1.2 Wind Chill (28 June 2016)

This release brings major performance and usability improvements
- Major performance improvements to GridWorkflow and Ingester
- Ingestion can be limited to one year at a time to limit memory usage
- Ingestion can be done in two stages (serial followed by highly parallel) by using –save-tasks/load-task options. This should help reduce idle time in distributed processing case.
- General improvements to documentation.

22.37 v1.1.1 Good Idea (23 June 2016)

This release contains lots of fixes in preparation for the first large ingestion of Geoscience Australia data into a production version of AGDCv2.
- General improvements to documentation and user friendliness.
- Updated metadata in configuration files for ingested products.
- Full provenance history is saved into ingested files.
- Added software versions, machine info and other details of the ingestion run into the provenance.
- Added valid data region information into metadata for ingested data.
- Fixed bugs relating to changes in Rasterio and GDAL versions.
- Refactored GridWorkflow to be easier to use, and include preliminary code for saving created products.
- Improvements and fixes for bit mask generation.
- Lots of other minor but important fixes throughout the codebase.

22.38 v1.1.0 No Spoon (3 June 2016)

This release includes restructuring of code, APIs, tools, configurations and concepts. The result of this churn is cleaner code, faster performance and the ability to handle provenance tracking of Datasets created within the Data Cube.

The major changes include:
- The `datacube-config` and `datacube-ingest` tools have been combined into `datacube`.
- Added dependency on pandas for nicer search results listing and handling.
- `Indexing` and `Ingesting Data` have been split into separate steps.
- Data that has been indexed can be accessed without going through the ingestion process.
- Data can be requested in any projection and will be dynamically reprojected if required.
• Dataset Type has been replaced by Product
• Storage Type has been removed, and an Ingestion Configuration has taken it’s place.
• A new Datacube Class for querying and accessing data.

22.39 1.0.4 Square Clouds (3 June 2016)

Pre-Unification release.

22.40 1.0.3 (14 April 2016)

Many API improvements.

22.41 1.0.2 (23 March 2016)

22.42 1.0.1 (18 March 2016)

22.43 1.0.0 (11 March 2016)

This release is to support generation of GA Landsat reference data.

22.44 pre-v1 (end 2015)

First working Data Cube v2 code.
This page lists definitions for terms used in the Open Data Cube and this manual.

**AGDC**

**Australian Geoscience Data Cube** An Australian project that developed the python library that became *ODC*.

**API**

**Application Programming Interface** The Data Cube Application Programming Interface gives programmers full access to the capabilities of the Cube, allowing query and advanced data retrieval.

**CEOS-SEO**

**Committee on Earth Observation, Systems Engineering Office** Another group of developers working on ODC

**CRS**

**Coordinate Reference System** Specifies how a set of coordinates relates to a point on the earth.

**CSIRO**

**Commonwealth Scientific and Industrial Research Organisation** An Australian research agency participating in development of ODC

**DEA**

**Digital Earth Australia** Geoscience Australia’s deployment of ODC, hosted at the *NCI*. See the DEA user guide for details on how to access the platform.

**GA**

**Geoscience Australia** The primary public sector geoscience organisation in Australia. It supports ODC through the development of Digital Earth Australia. For more information see http://www.ga.gov.au/dea/.

**grid spec**

**Grid Specification** A definition of how some raster data relates to a geospatial grid. Includes a CRS, a zero point, and an X and Y scale.

**HPC**
**High Performance Computing**  A supercomputer. These days typically a cluster of high performance computers linked together with high speed networking.

**Landsat**  A series of US owned earth observation satellites, with freely available data.

**MODIS**  US owned earth observation satellites with freely available data.

**NCI**

**National Computation Infrastructure**  Australia’s national research computing facility. It provides computing facilities for use by Australian researchers, industry and government. For more information see [http://nci.org.au/](http://nci.org.au/).

**ODC**

**Open Data Cube**  Software for storing, managing and accessing Earth Observation data, and the community formed to further it’s development.

**OGC**

**Open Geospatial Consortium**  [Open Geospatial Consortium. An organisation which makes geospatial standards, especially relating to web services.](http://www.opengeospatial.org/)

**PostgreSQL**  The high performance database engine used as an index of Datasets by the Data Cube. It is both a relational and document database, and the Data Cube schema makes use of both of these capabilities.

**Python**  The programming language used to develop the Data Cube. It is easy to use while still allowing high performance access and processing capabilities. See [https://www.python.org/](https://www.python.org/) for more.

**USGS**

**United States Geological Survey**  The USA government organisation which operates both the Landsat and MODIS satellites, which are available as free an open earth observation data.

**VDI**


**WMS**

**Web Map Service**  A standard for delivery published maps into web browsers and GIS software, defined by the [OGC](http://www.opengis.org/).

**YAML**

**Yet Another Markup Language**  A human friendly data serialization standard for all programming languages. It is used for all on disk metadata files and product definitions in the Data Cube.

For more information on the YAML standard see [http://yaml.org/](http://yaml.org/).
ODC License

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.
The best way to get help with the Data Cube is to open an issue on Github.
CHAPTER 26

Indices and tables

- genindex
- modindex
- search
d
datacube.drivers, 135
Symbols

-executor <executor>
  datacube-worker command line option, 51
-nprocs <nprocs>
  datacube-worker command line option, 51
__init__() (datacube.Datacube method), 55
__init__() (datacube.api.GridWorkflow method), 61
__init__() (datacube.api.grid_workflow.Tile method), 62
__init__() (datacube.api.query.Query method), 90
__init__() (datacube.config.LocalConfig method), 91
__init__() (datacube.index.Index method), 81
__init__() (datacube.index.hl.Doc2Dataset method), 83
__init__() (datacube.model.Dataset method), 66
__init__() (datacube.model.DatasetType method), 68
__init__() (datacube.model.GridSpec method), 69
__init__() (datacube.model.Measurement method), 67
__init__() (datacube.model.MetadataType method), 68
__init__() (datacube.model.Range method), 70
__init__() (datacube.utils.geometry.BoundingBox method), 83
__init__() (datacube.utils.geometry.CRS method), 84
__init__() (datacube.utils.geometry.Geometry method), 84

A
add() (datacube.index._datasets.DatasetResource method), 75
add() (datacube.index._products.ProductResource method), 78
add_document() (datacube.index._products.ProductResource method), 78

add_location() (datacube.index._datasets.DatasetResource method), 76

Affine (class in affine), 100
AGDC, 153
almost_equals() (affine.Affine method), 100
API, 153
Application Programming Interface, 153
archive() (datacube.index._datasets.DatasetResource method), 76
archive_location() (datacube.index._datasets.DatasetResource method), 76
assemble_array_from_s3() (datacube.drivers.s3.storage.s3aio.S3LIO method), 97
Australian Geoscience Data Cube, 153

B
BoundingBox (class in datacube.utils.geometry), 83
box() (in module datacube.utils.geometry), 87
bucket_exists() (datacube.drivers.s3.storage.s3aio.S3LIO method), 92
bulk_has() (datacube.index._datasets.DatasetResource method), 73

can_update() (datacube.index._datasets.DatasetResource method), 73
can_update() (datacube.index._products.ProductResource method), 78
cell_observations() (datacube.api.GridWorkflow method), 64
CEOS-SEO, 153
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chunk_indices_1d()</td>
<td>(datacube.drivers.s3.storage.s3aio.S3LIO method), 97</td>
</tr>
<tr>
<td>chunk_indices_nd()</td>
<td>(datacube.drivers.s3.storage.s3aio.S3LIO method), 97</td>
</tr>
<tr>
<td>column_vectors</td>
<td>(affine.Affine attribute), 100</td>
</tr>
<tr>
<td>count()</td>
<td>(datacube.index._datasets.DatasetResource method), 73</td>
</tr>
<tr>
<td>count_by_product()</td>
<td>(datacube.index._datasets.DatasetResource method), 73</td>
</tr>
<tr>
<td>count_by_product_through_time()</td>
<td>(datacube.index._datasets.DatasetResource method), 74</td>
</tr>
<tr>
<td>count_product_through_time()</td>
<td>(datacube.index._datasets.DatasetResource method), 74</td>
</tr>
<tr>
<td>CRS</td>
<td>153</td>
</tr>
<tr>
<td>CRS</td>
<td>(class in datacube.utils.geometry), 84</td>
</tr>
<tr>
<td>CSIRO</td>
<td>153</td>
</tr>
<tr>
<td>Datacube</td>
<td>(class in datacube), 55</td>
</tr>
</tbody>
</table>
| Datacube-worker command line option | -executor <executor>, 51
-nprocs <nprocs>, 51 |
| datacube.drivers | (module), 135 |
| Dataset | (class in datacube.model), 66 |
| DatasetType | (class in datacube.model), 68 |
| DEA | 153 |
| DEFAULT_CONF_PATHS | (in module datacube.config), 91 |
| delete_created_arrays() | (datacube.drivers.s3.storage.s3aio.S3LIO method), 92 |
| delete_objects() | (datacube.drivers.s3.storage.s3aio.S3LIO method), 92 |
| describe_variable_flags() | (datacube.storage.masking method), 89 |
| determinant | (affine.Affine attribute), 100 |
| Digital Earth Australia | 153 |
| Doc2Dataset | (class in datacube.index.hl), 82 |
| E | eccentricity | (affine.Affine attribute), 100 |

**G**

<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find_datasets()</td>
<td>(datacube.Datacube method), 59</td>
</tr>
<tr>
<td>from_doc()</td>
<td>(datacube.index._products.ProductResource method), 77</td>
</tr>
<tr>
<td>from_gdal()</td>
<td>(affine.Affine class method), 100</td>
</tr>
</tbody>
</table>

**GA**

<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometry</td>
<td>(class in datacube.utils.geometry), 84</td>
</tr>
<tr>
<td>Geoscience Australia</td>
<td>153</td>
</tr>
<tr>
<td>get()</td>
<td>(datacube.index._products.ProductResource method), 71</td>
</tr>
<tr>
<td>get()</td>
<td>(datacube.index._products.ProductResource method), 78</td>
</tr>
<tr>
<td>get_all()</td>
<td>(datacube.index._products.ProductResource method), 79</td>
</tr>
<tr>
<td>get_archived_locations()</td>
<td>(datacube.index._datasets.DatasetResource method), 75</td>
</tr>
<tr>
<td>get_by_name()</td>
<td>(datacube.index._products.ProductResource method), 78</td>
</tr>
<tr>
<td>get_by_name_unsafe()</td>
<td>(datacube.index._products.ProductResource method), 79</td>
</tr>
<tr>
<td>get_byte_range()</td>
<td>(datacube.drivers.s3.storage.s3aio.S3LIO method), 92</td>
</tr>
<tr>
<td>get_byte_range_mp()</td>
<td>(datacube.drivers.s3.storage.s3aio.S3LIO method), 93</td>
</tr>
<tr>
<td>get_bytes()</td>
<td>(datacube.drivers.s3.storage.s3aio.S3LIO method), 93</td>
</tr>
<tr>
<td>get_data()</td>
<td>(datacube.drivers.s3.storage.s3aio.S3LIO method), 97</td>
</tr>
<tr>
<td>get_data_unlabeled()</td>
<td>(datacube.drivers.s3.storage.s3aio.S3LIO method), 98</td>
</tr>
<tr>
<td>get_data_unlabeled_mp()</td>
<td>(datacube.drivers.s3.storage.s3aio.S3LIO method), 98</td>
</tr>
<tr>
<td>get_datasets_for_location()</td>
<td>(datacube.index._datasets.DatasetResource method), 75</td>
</tr>
<tr>
<td>get_derived()</td>
<td>(datacube.index._datasets.DatasetResource method), 74</td>
</tr>
<tr>
<td>get_field_names()</td>
<td>(datacube.index._datasets.DatasetResource method), 74</td>
</tr>
<tr>
<td>get_locations()</td>
<td>(datacube.index._datasets.DatasetResource method), 75</td>
</tr>
</tbody>
</table>
get_point() (datacube.drivers.s3.storage.s3aio.S3IO method), 95
get_slice() (datacube.drivers.s3.storage.s3aio.S3IO method), 96
get_slice_by_bbox() (datacube.drivers.s3.storage.s3aio.S3IO method), 96
get_slice_mp() (datacube.drivers.s3.storage.s3aio.S3IO method), 96
get_unsafe() (datacube.index._products.ProductResource method), 79
get_with_fields() (datacube.index._products.ProductResource method), 79
grid spec, 153
Grid Specification, 153
GridSpec (class in datacube.model), 69
GridWorkflow (class in datacube.api), 61
group_datasets() (datacube.Datacube static method), 59
group_into_cells() (datacube.api.GridWorkflow static method), 65
has() (datacube.index._datasets.DatasetResource method), 73
High Performance Computing, 154
HPC, 153
identity() (affine.Affine class method), 100
Index (class in datacube.index), 81
index_connect() (in module datacube.index), 80
index_driver_by_name() (in module datacube.drivers), 136
index_drivers() (in module datacube.drivers), 136
is_conformal (affine.Affine attribute), 100
is_degenerate (affine.Affine attribute), 100
is_identity (affine.Affine attribute), 101
is_orthonormal (affine.Affine attribute), 101
is_proper (affine.Affine attribute), 101
is_rectilinear (affine.Affine attribute), 101
itransform() (affine.Affine method), 101
L
Landsat, 154
line() (in module datacube.utils.geometry), 86
list_buckets() (datacube.drivers.s3.storage.s3aio.S3IO method), 93
list_cells() (datacube.api.GridWorkflow method), 62
list_created_arrays() (datacube.drivers.s3.storage.s3aio.S3IO method), 93
list_measurements() (datacube.Datacube method), 57
list_objects() (datacube.drivers.s3.storage.s3aio.S3IO method), 93
list_products() (datacube.Datacube method), 56
list_tiles() (datacube.Datacube method), 63
load() (datacube.api.GridWorkflow static method), 63
load() (datacube.Datacube method), 57
load_data() (datacube.Datacube static method), 60
LocalConfig (class in datacube.config), 91
M
make_mask() (datacube.storage.masking method), 89
mask_invalid_data() (datacube.storage.masking method), 88
Measurement (class in datacube.model), 67
measurement_data() (datacube.Datacube static method), 60
MetadataType (class in datacube.model), 68
MODIS, 154
multiline() (in module datacube.utils.geometry), 86
multipoint() (in module datacube.utils.geometry), 86
multipolygon() (in module datacube.utils.geometry), 87
N
National Computation Infrastructure, 154
NCI, 154
new_datasource() (in module datacube.drivers), 135
O
object_exists() (datacube.drivers.s3.storage.s3aio.S3IO method), 93
ODC, 154
OGC, 154
Open Data Cube, 154
Open Geospatial Consortium, 154
P
permutation() (affine.Affine class method), 101
point() (in module datacube.utils.geometry), 86
polygon() (in module datacube.utils.geometry), 86
polygon_from_transform() (in module datacube.utils.geometry), 87
PostgreSQL, 154
put_array_in_s3() (datacube.drivers.s3.storage.s3aio.S3LIO method), 98
put_array_in_s3_mp() (datacube.drivers.s3.storage.s3aio.S3LIO method), 99
put_bytes() (datacube.drivers.s3.storage.s3aio.S3LIO method), 94
put_bytes_mpu() (datacube.drivers.s3.storage.s3aio.S3LIO method), 94
put_bytes_mpu_mp() (datacube.drivers.s3.storage.s3aio.S3LIO method), 94
put_bytes_mpu_mp_shm() (datacube.drivers.s3.storage.s3aio.S3LIO method), 94

Python, 154

Q
Query (class in datacube.api.query), 90

R
Range (class in datacube.model), 70
reader_drivers() (in module datacube.drivers), 136
regular_index() (datacube.drivers.s3.storage.s3aio.S3LIO method), 99
remove_location() (datacube.index._datasets.DatasetResource method), 76
restore() (datacube.index._datasets.DatasetResource method), 77
restore_location() (datacube.index._datasets.DatasetResource method), 77
rotation() (affine.Affine class method), 101
rotation_angle (affine.Affine attribute), 101

S
s3_bucket() (datacube.drivers.s3.storage.s3aio.S3LIO method), 95
s3_object() (datacube.drivers.s3.storage.s3aio.S3LIO method), 95
s3_resource() (datacube.drivers.s3.storage.s3aio.S3LIO method), 95
S3AIO (class in datacube.drivers.s3.storage.s3aio), 95
S3LIO (class in datacube.drivers.s3.storage.s3aio), 92
S3LIO (class in datacube.drivers.s3.storage.s3aio), 97
scale() (affine.Affine class method), 101
search() (datacube.index._datasets.DatasetResource method), 71
search() (datacube.index._products.ProductResource method), 79
search_by_metadata() (datacube.index._datasets.DatasetResource method), 71
search_by_product() (datacube.index._datasets.DatasetResource method), 72
search_eager() (datacube.index._datasets.DatasetResource method), 72
search_product_duplicates() (datacube.index._datasets.DatasetResource method), 72
search_returning() (datacube.index._datasets.DatasetResource method), 72
search_robust() (datacube.index._products.ProductResource method), 79
search_summaries() (datacube.index._datasets.DatasetResource method), 73
shard_array_to_s3() (datacube.drivers.s3.storage.s3aio.S3LIO method), 99
shard_array_to_s3_mp() (datacube.drivers.s3.storage.s3aio.S3LIO method), 99
shear() (affine.Affine class method), 101
storage_writer_by_name() (in module datacube.drivers), 136

T
Tile (class in datacube.api.grid_workflow), 62
tile_sources() (datacube.api.GridWorkflow static method), 65
to_1d() (datacube.drivers.s3.storage.s3aio.S3LIO method), 96
to_gdal() (affine.Affine method), 102
to_nd() (datacube.drivers.s3.storage.s3aio.S3LIO method), 96
translation() (affine.Affine class method), 102

U
unary_intersection() (in module datacube.utils.geometry), 88
unary_union() (in module datacube.utils.geometry), 88
United States Geological Survey, 154
update() (datacube.index._datasets.DatasetResource method), 77
update() (datacube.index._products.ProductResource method), 80
update_document() (datacube.index._products.ProductResource method), 80
USGS, 154

V
VDI, 154
Virtual Desktop Infrastructure, 154

W
Web Map Service, 154
WMS, 154
writer_drivers() (in module datacube.drivers), 136

X
xoff (affine.Affine attribute), 102

Y
YAML, 154
Yet Another Markup Language, 154
yoff (affine.Affine attribute), 102