Cubes Documentation

Release 1.1

Stefan Urbanek

Apr 04, 2017

Contents

Cubes is a light-weight Python framework and set of tools for development of reporting and analytical applications, Online Analytical Processing (OLAP), multidimensional analysis and browsing of aggregated data. It is part of Data Brewery.

CHAPTER 1

Getting Started

Introduction

Why cubes?

Purpose is to provide a framework for giving analyst or any application end-user understandable and natural way of reporting using concept of data Cubes – multidimensional data objects.

It is meant to be used by application builders that want to provide analytical functionality.

Features:

- logical view of analysed data - how analysts look at data, how they think of data, not how the data are physically implemented in the data stores
- OLAP and aggregated browsing (default backend is for relational database - ROLAP)
- hierarchical dimensions (attributes that have hierarchical dependencies, such as category-subcategory or country-region)
- multiple hierarchies in a dimension
- localizable metadata and data (see *Localization*)
- authentication and authorization of cubes and their data
- pluggable data warehouse – plug-in other cube-like (multidimensional) data sources

The framework is very extensible.

Cube, Dimensions, Facts and Measures

The framework models the data as a cube with multiple dimensions:

The most detailed unit of the data is a *fact*. Fact can be a contract, invoice, spending, task, etc. Each fact might have a *measure* – an attribute that can be measured, such as: price, amount, revenue, duration, tax, discount, etc.

The *dimension* provides context for facts. Is used to:

- filter queries or reports
- controls scope of aggregation of facts

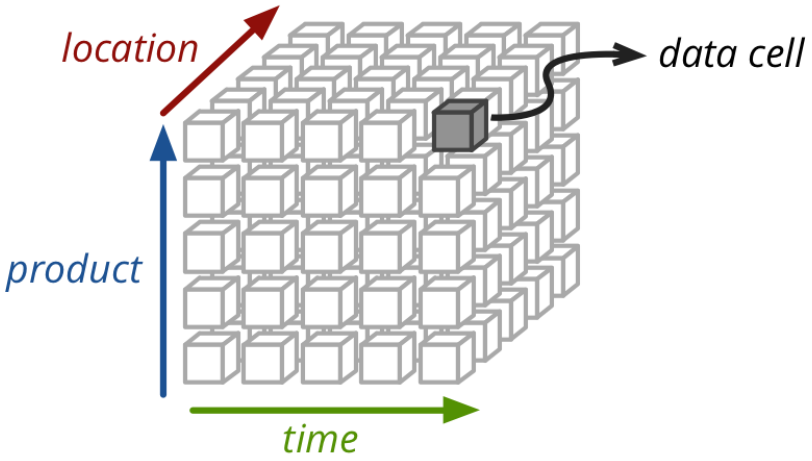


Fig. 1.1: a data cube

- used for ordering or sorting
- defines master-detail relationship

Dimension can have multiple *hierarchies*, for example the date dimension might have year, month and day levels in a hierarchy.

Feature Overview

Core cube features:

- **Workspace** – Cubes analytical workspace (see *docs*, *reference*)
- **Model** - Description of data (*metadata*): cubes, dimensions, concept hierarchies, attributes, labels, localizations. (see *docs*, *reference*)
- **Browser** - Aggregation browsing, slicing-and-dicing, drill-down. (see *docs*, *reference*)
- **Backend** - Actual aggregation implementation and utility functions. (see docs, *reference*)
- **Server** - WSGI HTTP server for Cubes (see *docs*, *reference*)
- **Formatters** - Data formatters (see *docs*, *reference*)
- *slicer - Command Line Tool* - command-line tool

Model

Logical model describes the data from user's or analyst's perspective: data how they are being measured, aggregated and reported. Model is independent of physical implementation of data. This physical independence makes it easier to focus on data instead on ways of how to get the data in understandable form.

More information about logical model can be found in the chapter *Logical Model and Metadata*.

See also developer's *reference*.

Browser

Core of the Cubes analytics functionality is the aggregation browser. The browser module contains utility classes and functions for the browser to work.

More information about browser can be found in the chapter *Slicing and Dicing*. See also programming *reference*.

Backends

Backends provide the actual data aggregation and browsing functionality. Cubes comes with built-in ROLAP backend which uses SQL database using SQLAlchemy.

Framework has modular nature and supports multiple database backends, therefore different ways of cube computation and ways of browsing aggregated data.

Multiple backends can be used at once, even multiple sources from the same backend might be used in the analytical workspace.

More about existing backends can be found in the backends documentation. See also the backends programming reference *reference*.

Server

Cubes comes with built-in WSGI HTTP OLAP server called *slicer - Command Line Tool* and provides json API for most of the cubes framework functionality. The server is based on the Werkzeug WSGI framework.

More information about the Slicer server requests can be found in the chapter *OLAP Server*. See also programming reference of the `server` module.

See also:

Schemas and Models Example database schemas and use patterns with their respective models.

Installation

There are two options how to install cubes: basic common installation - recommended mostly for users starting with Cubes. Then there is customized installation with requirements explained.

Dependencies:

- SQLAlchemy
- *expressions*
- *python-dateutil*

Basic Installation

The cubes has optional requirements:

- Flask for Slicer OLAP HTTP server

Note: If you never used Python before, you might have to get the pip installer first, if you do not have it already.

Note: The command-line tool *Slicer* does not require knowledge of Python. You do not need to know the language if you just want to *serve* OLAP data.

You may install Cubes with the minimal dependencies,

```
pip install cubes
```

with certain extras (html, sql, mongo, or slicer),

```
pip install cubes[slicer]
```

or with all of the extras.

```
pip install cubes[all]
```

If you are developing cubes, you should install `cubes[all]`.

Quick Start or Hello World!

Download the sources from the Cubes Github repository. Go to the `examples/hello_world` folder:

```
git clone git://github.com/DataBrewery/cubes.git
cd cubes
cd examples/hello_world
```

Prepare data and run the *OLAP server*:

```
python prepare_data.py
slicer serve slicer.ini
```

And try to do some queries:

```
curl "http://localhost:5000/cube/irbd_balance/aggregate"
curl "http://localhost:5000/cube/irbd_balance/aggregate?drilldown=year"
curl "http://localhost:5000/cube/irbd_balance/aggregate?drilldown=item"
curl "http://localhost:5000/cube/irbd_balance/aggregate?drilldown=item&cut=item:e"
```

Customized Installation

The project sources are stored in the Github repository.

Download from Github:

```
git clone git://github.com/DataBrewery/cubes.git
```

Install:

```
cd cubes
pip install -r requirements.txt
pip install -r requirements-optional.txt
python setup.py install
```

Note: The requirements for SQLAlchemy and Flask are optional and you do not need them if you are going to use another kind of backend or don't going to use the Slicer server.

Related Projects

Visualization

Cubes Viewer

Author: Jose Juan Montes

Links: Home , Github source

Cubes Viewer is a visual, responsive HTML5 application and library for exploring and visualizing different types of datasets.

CubesViewer can be used for data exploration and data auditory, generation of reports, chart design and embedding, and as a (simple) company-wide analytics application.

Other Languages

TypeScript: Static typed version of cubes.js

Author: Abbas (martianboy)

Tutorial

This chapter describes step-by-step how to use the Cubes. You will learn:

- model preparation
- measure aggregation
- drill-down through dimensions
- how to slice&dice the dube

The tutorial contains examples for both: standard tool use and Python use. You don't need to know Python to follow this tutorial.

Data Preparation

The example data used are IBRD Balance Sheet taken from The World Bank. Backend used for the examples is `sql.browser`.

Create a tutorial directory and download `IBRD_Balance_Sheet__FY2010.csv`.

Start with imports:

```
>>> from sqlalchemy import create_engine
>>> from cubes.tutorial.sql import create_table_from_csv
```

Note: Cubes comes with tutorial helper methods in `cubes.tutorial`. It is advised **not** to use them in production; they are provided just to simplify the tutorial.

Prepare the data using the tutorial helpers. This will create a table and populate it with contents of the CSV file:

```
>>> engine = create_engine('sqlite:///data.sqlite')
... create_table_from_csv(engine,
...                       "IBRD_Balance_Sheet__FY2010.csv",
...                       table_name="ibrd_balance",
...                       fields=[
...                             ("category", "string"),
...                             ("category_label", "string"),
...                             ("subcategory", "string"),
...                             ("subcategory_label", "string"),
...                             ("line_item", "string"),
...                             ("year", "integer"),
...                             ("amount", "integer")],
...                       create_id=True
...                     )
```

Analytical Workspace

Everything in Cubes happens in an *analytical workspace*. It contains cubes, maintains connections to the data stores (with cube data), provides connection to external cubes and more.

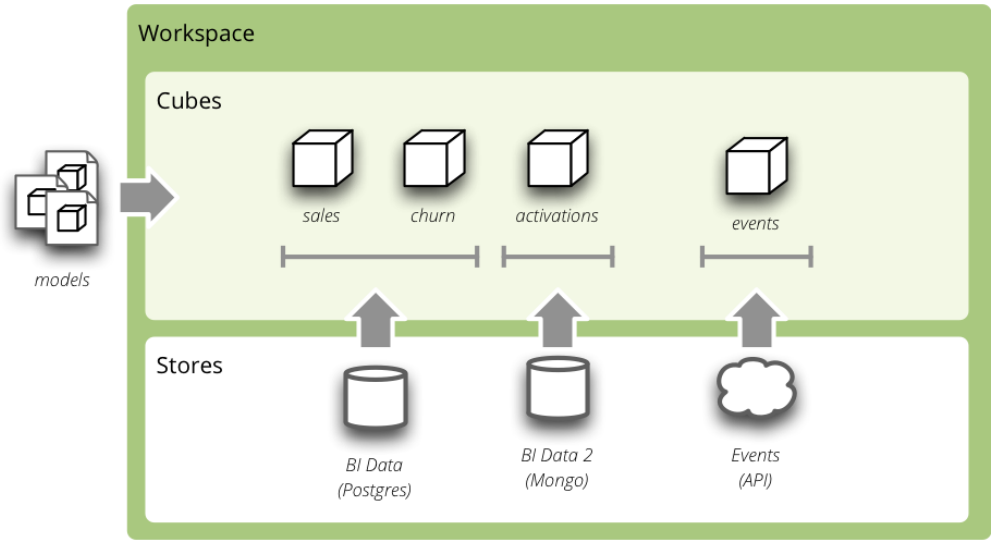


Fig. 1.2: Analytical workspace and it's content

The workspace properties are specified in a configuration file *slicer.ini* (default name). First thing we have to do is to specify a data store – the database containing the cube's data:

```
[store]
type: sql
url: sqlite:///data.sqlite
```

In Python, a workspace can be configured using the *ini* configuration:

```
from cubes import Workspace

workspace = Workspace(config="slicer.ini")
```

or programatically:

```
workspace = Workspace()
workspace.register_default_store("sql", url="sqlite:///data.sqlite")
```

Model

Download the `tutorial model` and save it as `tutorial_model.json`.

In the *slicer.ini* file specify the model:

```
[workspace]
model: tutorial_model.json
```

For more information about how to add more models to the workspace see the *configuration documentation*.

Equivalent in Python is:

```
>>> workspace.import_model("tutorial_model.json")
```

You might call `import_model()` with as many models as you need. Only limitation is that the public cubes and public dimensions should have unique names.

Aggregations

Browser is an object that does the actual aggregations and other data queries for a cube. To obtain one:

```
>>> browser = workspace.browser("ibrd_balance")
```

Compute the aggregate. Measure fields of `AggregationResult` have aggregation suffix. Also a total record count within the cell is included as `record_count`.

```
>>> result = browser.aggregate()
>>> result.summary["record_count"]
62
>>> result.summary["amount_sum"]
1116860
```

Now try some drill-down by *year* dimension:

```
>>> result = browser.aggregate(drilldown=["year"])
>>> for record in result:
...     print record
{u'record_count': 31, u'amount_sum': 550840, u'year': 2009}
{u'record_count': 31, u'amount_sum': 566020, u'year': 2010}
```

Drill-down by item category:

```
>>> result = browser.aggregate(drilldown=["item"])
>>> for record in result:
...     print record
{u'item.category': u'a', u'item.category_label': u'Assets', u'record_count': 32, u
↪'amount_sum': 558430}
{u'item.category': u'e', u'item.category_label': u'Equity', u'record_count': 8, u
↪'amount_sum': 77592}
{u'item.category': u'l', u'item.category_label': u'Liabilities', u'record_count':␣
↪22, u'amount_sum': 480838}
```

Credits

Cubes was created and is maintained by Stefan Urbanek.

Major contributing authors:

- Stefan Urbanek, stefan.urbanek@gmail.com, Twitter, Github
- Robin Thomas, rthomas@squarespace.com, Github

Thanks to Squarespace for sponsoring the development time.

People who have submitted patches, reported bugs, consulted features or generally made Cubes better:

- Jose Juan Montes (jjmontesl)
- Jonathan Camile (deytao)
- Cristian Salamea
- Travis Truman

CHAPTER 2

Data Modeling

Logical Model and Metadata

Logical model describes the data from user's or analyst's perspective: data how they are being measured, aggregated and reported. Model is independent of physical implementation of data. This physical independence makes it easier to focus on data instead on ways of how to get the data in understandable form.

See also:

Schemas and Models Example database schemas and their respective models.

Model Reference Reference of model classes and functions.

Cubes Models Repository of basic cubes models.

Introduction

The logical model enables users to:

- see the data from the business perspective
- hide physical structure of the data ("how application's use it")
- **specify concept hierarchies of attributes, such as:**
  - *product category* > *product* > *subcategory* > *product*
  - *country* > *region* > *county* > *town*.
- provide more descriptive attribute labels for display in the applications or reports
- transparent localization of metadata and data

Analysts or report writers do not have to know where name of an organisation or category is stored, nor he does not have to care whether customer data is stored in single table or spread across multiple tables (customer, customer types, ...). They just ask for *customer.name* or *category.code*.

In addition to abstraction over physical model, localization abstraction is included. When working in multi-lingual environment, only one version of report/query has to be written, locales can be switched as desired. If requesting "contract type name", analyst just writes *contract_type.name* and Cubes framework takes care about appropriate localisation of the value.

Example: Analysts wants to report contract amounts by geography which has two levels: country level and region level. In original physical database, the geography information is normalised and stored in two separate tables, one for countries and another for regions. Analyst does not have to know where the data are stored, he just queries for *geography.country* and/or *geography.region* and will get the proper data. How it is done is depicted on the following image:

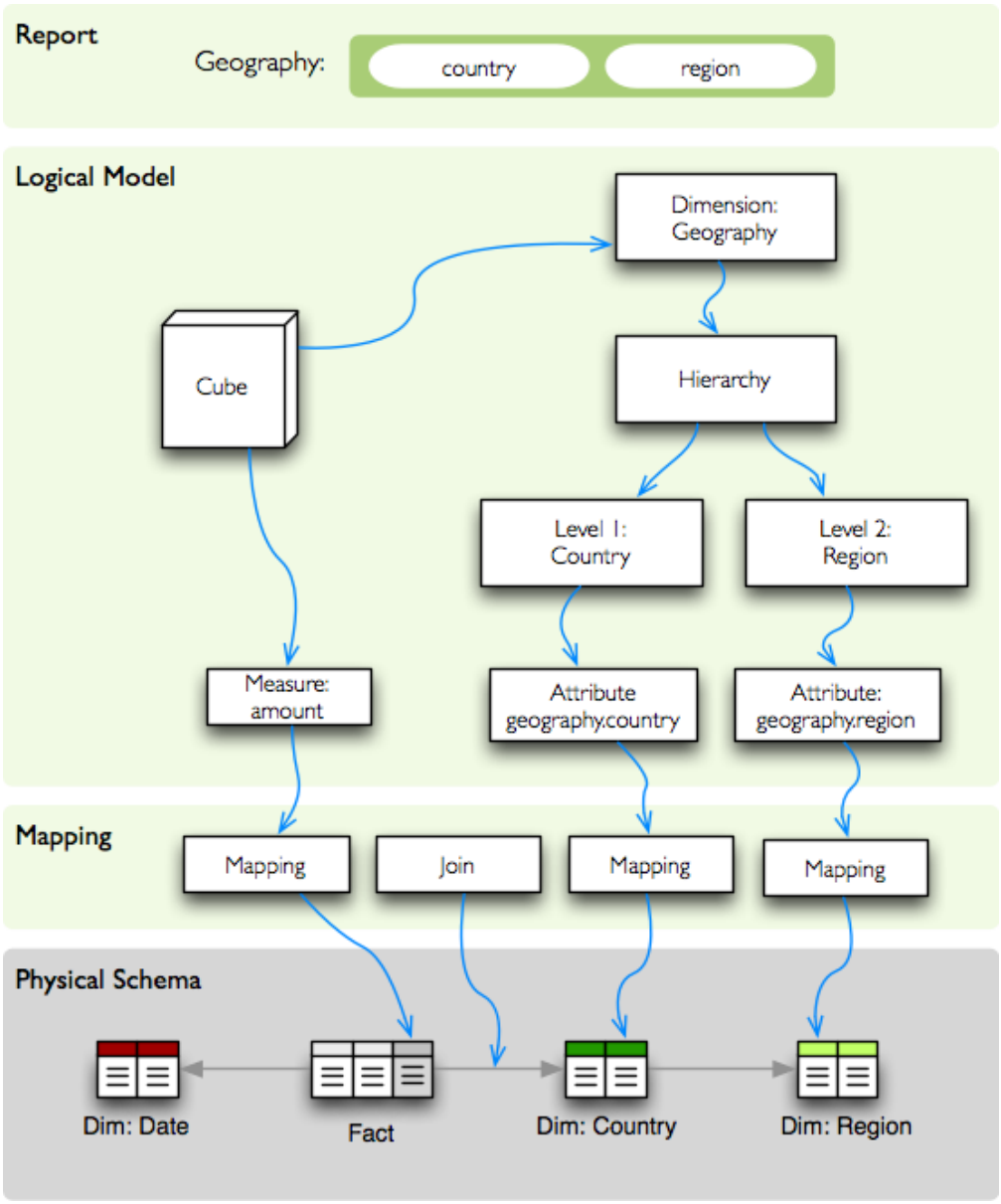


Fig. 2.1: Mapping from logical model to physical data.

The logical model describes dimensions *geography* in which default hierarchy has two levels: *country* and *region*. Each level can have more attributes, such as code, name, population... In our example report we are interested only in geographical names, that is: *country.name* and *region.name*.

Model

The logical model is described using *model metadata dictionary*. The content is description of logical objects, physical storage and other additional information.

Logical part of the model description:

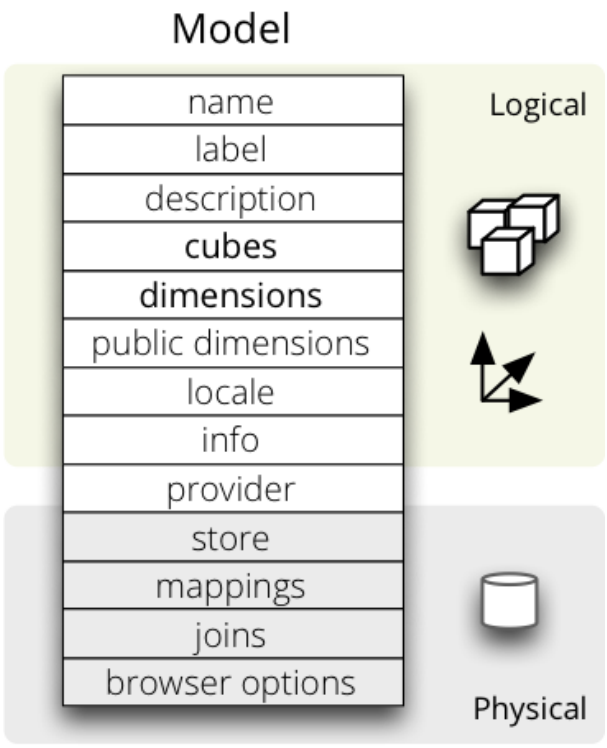


Fig. 2.2: Logical model metadata

- `name` – model name
- `label` – human readable model label *(optional)*
- `description` – human readable description of the model *(optional)*
- `locale` – locale the model metadata are written in *(optional, used for localizable models)*
- `cubes` – list of cubes metadata (see below)
- `dimensions` – list of dimension metadata (see below)

Physical part of the model description:

- `store` – name of the datastore where model's cubes are stored. Default is `default`. See *Analytical Workspace* for more information.
- `mappings` - backend-specific logical to physical mapping dictionary. This dictionary is inherited by every cube in the model.
- `joins` - backend-specific join specification (used for example in the SQL backend). It should be a list of dictionaries. This list is inherited by the cubes in the model.
- `browser_options` – options passed to the browser. The options are merged with options in the cubes.

Example model snippet:

```
{
    "name": "public_procurements",
    "label": "Public Procurements of Slovakia",
    "description": "Contracts of public procurement winners in Slovakia"
    "cubes": [...]
    "dimensions": [...]
}
```

Mappings and Joins

One can specify shared mappings and joins on the model-level. Those mappings and joins are inherited by all the cubes in the model.

The `mappings` dictionary of a cube is merged with model's global mapping dictionary. Cube's values overwrite the model's values.

The `joins` can be considered as named templates. They should contain `name` property that will be referenced by a cube.

Visibility: The joins and mappings are local to a single model. They are not shared within the workspace.

Inheritance

Cubes in a model will inherit mappings and joins from the model. The mappings are merged in a way that cube's mappings replace existing model's mappings with the same name. Joins are concatenated or merged by their name.

Example from the SQL backend: Say you would like to join a date dimension table in `dim_date` to every cube. Then you specify the join at the model level as:

```
"joins": [
    {
        "name": "date",
        "detail": "dim_date.date_id",
        "method": "match"
    }
]
```

The join has a name specified, which is used to match joins in the cube. Note that the join contains incomplete information: it contains only the `detail` part, that is the dimension key. To use the join in a cube which has two date dimensions *start date* and *end date*:

```
"joins": [
    {
        "name": "date",
        "master": "fact_contract.contract_start_date_id",
    },
    {
        "name": "date",
        "master": "fact_sales.contract_sign_date_id",
    }
]
```

The model's joins are searched for a template with given name and then cube completes (or even replaces) the join information.

For more information about mappings and joins refer to the backend documentation for your data store, such as *SQL*

File Representation

The model can be represented either as a JSON file or as a directory with JSON files. The single-file model specification is just a dictionary with model properties. The model directory bundle should have the following content:

- `model.json` – model's master metadata – same as single-file model
- `dim_*.json` – dimension metadata file – single dimension dictionary
- `cube_*.json` – cube metadata – single cube dictionary

The list of dimensions and cubes in the `model.json` are merged with the dimensions and cubes in the separate files. Avoid duplicate definitions.

Example directory bundle model:

```
model.cubesmodel/
    model.json
    dim_date.json
    dim_organization.json
    dim_category.json
    cube_contracts.json
    cube_events.json
```

Model Provider and External Models

If the model is provided from an external source, such as an API or a database, then name of the provider should be specified in `provider`.

The provider receives the model's metadata and the model's data store (if the provider so desires). Then the provider generates all the cubes and the dimensions.

Example of a model that is provided from an external source (Mixpanel):

```
{
    "name": "Events",
    "provider": "mixpanel"
}
```

Note: The *cubes* and *dimensions* in the generated model are just informative for the model provider. The provider can yield different set of cubes and dimensions as specified in the metadata.

See also:

`cubes.ModelProvider()` Load a model from a file or a URL.

`cubes.StaticModelProvider()` Create model from a dictionary.

Dimension Visibility

All dimensions from a static (file) model are shared in the workspace by default. That means that the dimensions can be reused freely among cubes from different models.

One can define a master model with dimensions only and no cubes. Then define one model per cube category, datamart or any other categorization. The models can share the master model dimensions.

To expose only certain dimensions from a model specify a list of dimension names in the `public_dimensions` model property. Only dimensions from the list can be shared by other cubes in the workspace.

Note: Some backends, such as Mixpanel, don't share dimensions at all.

Cubes

Cube descriptions are stored as a dictionary for key `cubes` in the model description dictionary or in json files with prefix `cube_` like `cube_contracts`.

Key	Description
Basic	
`name` *	Cube name, unique identifier. Required.
`label`	Human readable name - can be used in an application
`description`	Longer human-readable description of the cube *(optional)*
`info`	Custom info, such as formatting. Not used by cubes framework.
`dimensions` *	List of dimension names or dimension links (recommended, but might be empty for dimension-less cubes). Recommended.
`measures`	List of cube measures (recommended, but might be empty for measure-less, record count only cubes). Recommended.
`aggregates`	List of aggregated measures. Required, if no measures are specified.
`details`	List of fact details (as Attributes) - attributes that are not relevant to aggregation, but are nice-to-have when displaying facts (might be separately stored)
Physical	
`joins`	Specification of physical table joins (required for star/snowflake schema)
`mappings`	Mapping of logical attributes to physical attributes
`key`	Fact key field or column name. If not specified, backends might either refuse to generate facts or might use some default column name such as `id`.
`fact`	Fact table, collection or source name – interpreted by the backend. The fact table does not have to be specified, as most of the backends will derive the name from the cube's name.
Advanced	
`browser_options`	Browser specific options, consult the backend for more information
`store`	Name of a datastore where the cube is stored. Use this only when default store assignment is different from your requirements.

Fields marked with * are required.

Example:

```
{
    "name": "sales",
    "label": "Sales",
    "dimensions": [ "date", ... ]

    "measures": [...],
    "aggregates": [...],
    "details": [...],

    "fact": "fact_table_name",
    "mappings": { ... },
    "joins": [ ... ]
}
```

Note: The `key` might be required by some backends, such as SQL, to be able to generate detailed facts or to get a single fact. Please refer to the backend's documentation for more information.

Measures and Aggregates

Measures are numerical properties of a fact. They might be represented, for example, as a table column. Measures are aggregated into measure aggregates. The measure is described as:

- `name` – measure identifier (required)
- `label` – human readable name to be displayed (localized)
- `info` – additional custom information (unspecified)
- `aggregates` – list of aggregate functions that are provided for this measure. This property is for generating default aggregates automatically. It is highly recommended to list the aggregates explicitly and avoid

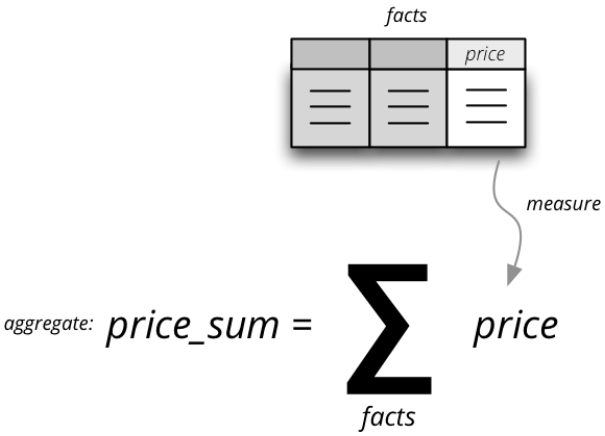


Fig. 2.3: Measure and measure aggregate

using this property.

- `window_size` – number of elements within a window for window functions such as moving average. If not provided and function requires it then 1 (one element) is assumed.

Example:

```
"measures": [
    {
        "name": "amount",
        "label": "Sales Amount"
    },
    {
        "name": "vat",
        "label": "VAT"
    }
]
```

Measure aggregate is a value computed by aggregating measures over facts. It's properties are:

- `name` – aggregate identifier, such as: *amount_sum*, *price_avg*, *total*, *record_count*
- `label` – human readable label to be displayed (localized)
- `measure` – measure the aggregate is derived from, if it exists or it is known. Might be empty.
- `function` - name of an aggregate function applied to the *measure*, if known. For example: *sum*, *min*, *max*.
- `window_size` – number of elements within a window for window functions such as moving average. If not provided and function requires it then 1 (one element) is assumed.
- `info` – additional custom information (unspecified)
- `expression` - to be used instead of `function`, this allows you to use simple, SQL-like expressions to calculate the value of an aggregate based on attributes of the fact. Alternatively, remind that fields can also be calculated at database level if your database system supports views.

Example:

```
"aggregates": [
    {
        "name": "amount_sum",
        "label": "Total Sales Amount",
        "measure": "amount",
        "function": "sum"
    },
    {
        "name": "vat_sum",
```

```
        "label": "Total VAT",
        "measure": "vat",
        "function": "sum"
    },
    {
        "name": "sales_minus_tax",
        "label": "Sales less VAT",
        "expression": "sum(amount) - sum(vat)"
    },
    {
        "name": "item_count",
        "label": "Item Count",
        "function": "count"
    }

]
```

Note the last aggregate `item_count` – it counts number of the facts within a cell. No measure required as a source for the aggregate.

If no aggregates are specified, Cubes generates default aggregates from the measures. For a measure:

```
"measures": [
    {
        "name": "amount",
        "aggregates": ["sum", "min", "max"]
    }
]
```

The following aggregates are created:

```
"aggregates" = [
    {
        "name": "amount_sum",
        "measure": "amount",
        "function": "sum"
    },
    {
        "name": "amount_min",
        "measure": "amount",
        "function": "min"
    },
    {
        "name": "amount_max",
        "measure": "amount",
        "function": "max"
    }
]
```

If there is a list of aggregates already specified in the cube explicitly, both lists are merged together.

Note: To prevent automated creation of default aggregates from measures, there is an advanced cube option `implicit_aggregates`. Set this property to *False* if you want to keep only explicit list of aggregates.

In previous version of Cubes there was omnipresent measure aggregate called `record_count`. It is no longer provided by default and has to be explicitly defined in the model. The name can be of any choice, it is not a built-in aggregate anymore. To keep the original behavior, the following aggregate should be added:

```
"aggregates": [
    {
        "name": "record_count",
```

```
        "function": "count"
    }
]
```

Note: Some aggregates do not have to be computed from measures. They might be already provided by the data store as computed aggregate values (for example Mixpanel's *total*). In this case the *measure* and *function* serves only for the backend or for informational purposes. Consult the backend documentation for more information about the aggregates and measures.

See also:

cubes.Cube Cube class reference.

cubes.Measure Measure class reference.

cubes.MeasureAggregate Measure Aggregate class reference.

Customized Dimension Linking

It is possible to specify how dimensions are linked to the cube. The `dimensions` list might contain, besides dimension names, also a specification how the dimension is going to be used in the cube's context. The specification might contain:

- `hierarchies` – list of hierarchies that are relevant for the cube. For example the *date* dimension might be defined as having *day* granularity, but some cubes might be only at the *month* level. To specify only relevant hierarchies use `hierarchies` metadata property:
- `exclude_hierarchies` – hierarchies to be excluded when cloning the original dimension. Use this instead of `hierarchies` if you would like to preserve most of the hierarchies and remove just a few.
- `default_hierarchy_name` – name of default hierarchy for a dimension in the context of the cube
- `cardinality` – cardinality of the dimension with regards to the cube. For example one cube might contain housands product types, another might have only a few, but they both share the same *products* dimension
- `alias` – how the dimension is going to be called in the cube. For example, you might have two date dimensions and name them *start_date* and *end_date* using the alias

Example:

```
{
    "name": "churn",

    "dimensions": [
        {"name": "date", "hierarchies": ["ym", "yqm"]},
        "customer",
        {"name": "date", "alias": "contract_date"}
    ],

    ...
}
```

The above cube will have three dimensions: *date*, *customer* and *contract_date*. The *date* dimension will have only two hierarchies with lowest granularity of *month*, the *customer* dimension will be assigned as-is and the *contract_date* dimension will be the same as the original *date* dimension.

Dimensions

Dimension descriptions are stored in model dictionary under the key `dimensions`.

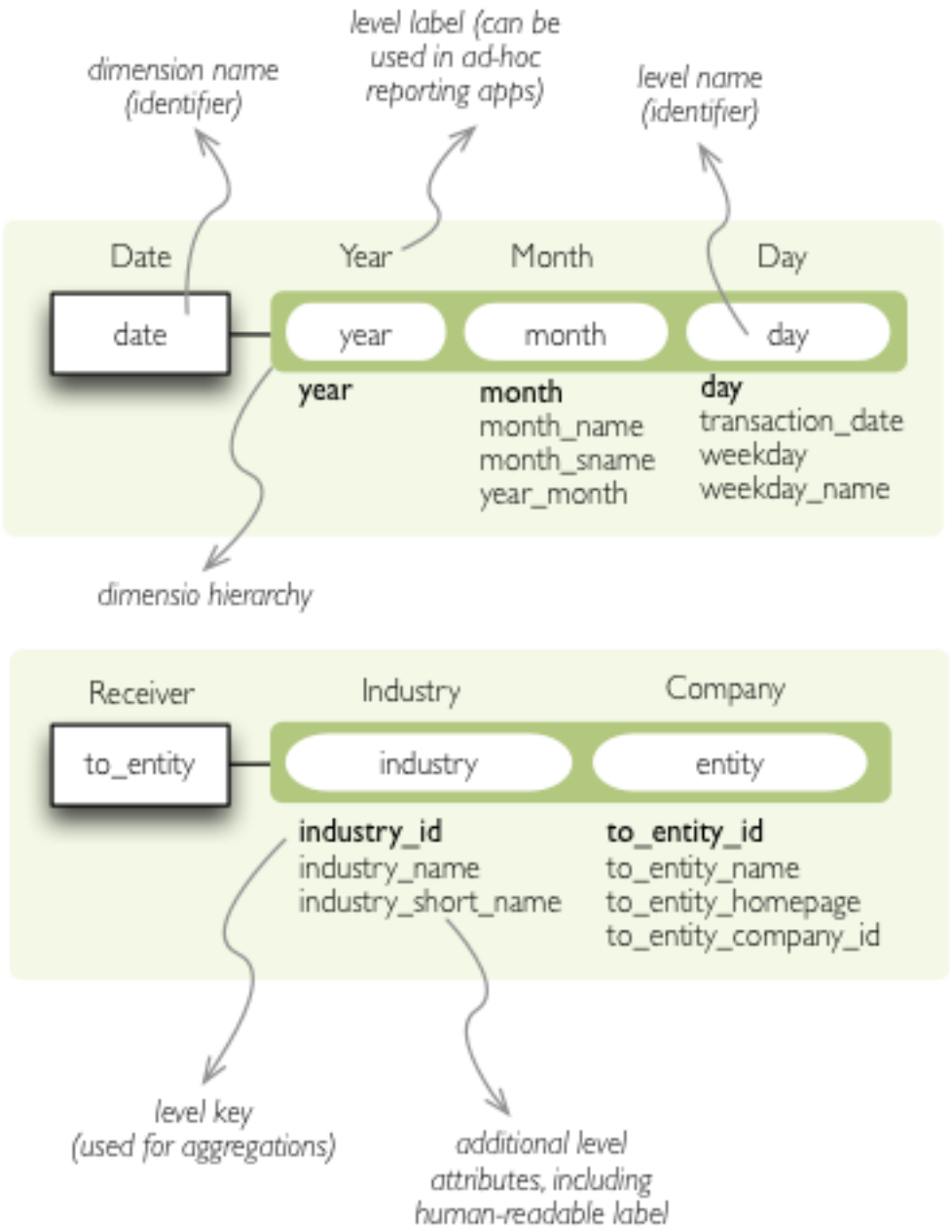


Fig. 2.4: Dimension description - attributes.

The dimension description contains keys:

Key	Description
Basic	
`name` *	dimension name, used as identifier
`label`	human readable name - can be used in an application
`description`	longer human-readable description of the dimension *(optional)*
`info`	custom info, such as formatting. Passed to the front-end.
`levels`	list of level descriptions
`hierarchies`	list of dimension hierarchies
`default_hierarchy_name`	name of a hierarchy that will be used as default
Advanced	
`cardinality`	dimension cardinality (see Level for more info)
`role`	dimension role
`category`	logical category (user oriented metadata)
`template`	name of a dimension that will be used as template

Fields marked with * are required.

If no levels are specified, then one default level will be created.

If no hierarchy is specified, then the default hierarchy will contain all levels of the dimension.

Example:

```
{
    "name": "date",
    "label": "Dátum",
    "levels": [ ... ]
    "hierarchies": [ ... ]
}
```

Use either `hierarchies` or `hierarchy`, using both results in an error.

Dimension Templates

If you are creating more dimensions with the same or similar structure, such as multiple dates or different types of organisational relationships, you might create a template dimension and then use it as base for the other dimensions:

```
"dimensions" = [
    {
        "name": "date",
        "levels": [...]
    },
    {
        "name": "creation_date",
        "template": "date"
    },
    {
        "name": "closing_date",
        "template": "date"
    }
]
```

All properties from the template dimension will be copied to the new dimension. Properties can be redefined in the new dimension. In that case, the old value is discarded. You might change levels, hierarchies or default hierarchy. There is no way how to add or drop a level from the template, all new levels have to be specified again if they are different than in the original template dimension. However, you might want to just redefine hierarchies to omit unnecessary levels.

Note: In mappings name of the new dimension should be used. The template dimension was used only to create the new dimension and the connection between the new dimension and the template is lost. In our example above, if cube uses the *creation_date* and *closing_date* dimensions and any mappings would be necessary, then they should be for those two dimensions, not for the *date* dimension.

Level

Dimension hierarchy levels are described as:

Key	Description
`name` *	level name, used as identifier
`label`	human readable name - can be used in an application
`attributes`	list of other additional attributes that are related to the level. The attributes are not being used for aggregations, they provide additional useful information.
`key`	key field of the level (customer number for customer level, region code for region level, year-month for month level). key will be used as a grouping field for aggregations. Key should be unique within level.
`label_attribute`	name of attribute containing label to be displayed (customer name for customer level, region name for region level, month name for month level)
`order_attribute`	name of attribute that is used for sorting, default is the first attribute (key)
`cardinality`	symbolic approximation of the number of level's members
`role`	Level role (see below)
`info`	custom info, such as formatting. Not used by cubes framework.

Fields marked with * are required.

If no attributes are specified then only one attribute is assumed with the same name as the level.

If no *key* is specified, then first attribute is assumed.

If no *label_attribute* is specified, then second attribute is assumed if level has more than one attribute, otherwise the first attribute is used.

Example of month level of date dimension:

```
{
    "month",
    "label": "Mesiac",
    "key": "month",
    "label_attribute": "month_name",
    "attributes": ["month", "month_name", "month_sname"]
},
```

Example of supplier level of supplier dimension:

```
{
    "name": "supplier",
    "label": "Dodávatel'",
    "key": "ico",
    "label_attribute": "name",
    "attributes": ["ico", "name", "address", "date_start", "date_end",
                   "legal_form", "ownership"]
}
```

See also:

`cubes.Dimension` Dimension class reference

`cubes.create_dimension()` Create a dimension object from a description dictionary.

`cubes.Level` Level class reference

cubes.create_level() Create level object from a description dictionary.

Note: Level attribute names have to be unique within a dimension that owns the level.

Cardinality

The *cardinality* property is used optionally by backends and front-ends for various purposes. The possible values are:

- `tiny` – few values, each value can have it's representation on the screen, recommended: up to 5.
- `low` – can be used in a list UI element, recommended 5 to 50 (if sorted)
- `medium` – UI element is a search/text field, recommended for more than 50 elements
- `high` – backends might refuse to yield results without explicit pagination or cut through this level.

Hierarchy

Hierarchies are described as:

Key	Description
`name`	hierarchy name, used as identifier
`label`	human readable name - can be used in an application
`levels`	ordered list of level names from top to bottom - from least detailed to most detailed (for example: from year to day, from country to city)

Required is only *name*.

Example:

```
"hierarchies": [
    {
        "name": "default",
        "levels": ["year", "month"]
    },
    {
        "name": "ymd",
        "levels": ["year", "month", "day"]
    },
    {
        "name": "yqmd",
        "levels": ["year", "quarter", "month", "day"]
    }
]
```

Attributes

Dimension level attributes can be specified either as rich metadata or just simply as strings. If only string is specified, then all attribute metadata will have default values, label will be equal to the attribute name.

Key	Description
name	attribute name (should be unique within a dimension)
label	human readable name - can be used in an application, localizable
order	natural order of the attribute (optional), can be `asc` or `desc`
format	application specific display format information
missing_value	Value to be substituted when there is no value (NULL) in the source (backend has to support this feature)
locales	list of locales in which the attribute values are available in (optional)
info	custom info, such as formatting. Not used by cubes framework.

The optional *order* is used in aggregation browsing and reporting. If specified, then all queries will have results sorted by this field in specified direction. Level hierarchy is used to order ordered attributes. Only one ordered attribute should be specified per dimension level, otherwise the behavior is unpredictable. This natural (or default) order can be later overridden in reports by explicitly specified another ordering direction or attribute. Explicit order takes precedence before natural order.

For example, you might want to specify that all dates should be ordered by default:

```
"attributes" = [
    {"name" = "year", "order": "asc"}
]
```

Locales is a list of locale names. Say we have a *CPV* dimension (common procurement vocabulary - EU procurement subject hierarchy) and we are reporting in Slovak, English and Hungarian. The attributes will be therefore specified as:

```
"attributes" = [
    {
        "name" = "group_code"
    },
    {
        "name" = "group_name",
        "order": "asc",
        "locales" = ["sk", "en", "hu"]
    }
]
```

group name is localized, but *group code* is not. Also you can see that the result will always be sorted by *group name* alphabetical in ascending order.

In reports you do not specify locale for each localized attribute, you specify locale for whole report or browsing session. Report queries remain the same for all languages.

Roles

Some dimensions and levels might have special, but well known, roles. One example of a role is *time*. There might be more recognized roles in the future, for example *geography*.

Front-ends that respect roles might provide different user interface elements, such as date and time pickers for selecting values of a date/time dimension. For the date picker to work, the front-end has to know, which dimension represents date and which levels of the dimension represent calendar units such as year, month or day.

The role of a dimension has to be explicitly stated. Front-ends are not required to assume a dimension named *date* is really a full date dimension.

The level roles do not have to be mentioned explicitly, if the level name can be recognized to match a particuliar role. For example, in a dimension with role *time* level with name *year* will have automatically role *year*.

Level roles have to be specified when level names are in different language or for any reason don't match english calendar unit names.

Currently there is only one recognized dimension role: `time`. Recognized level roles with their default assignment by level name are: `year`, `quarter`, `month`, `day`, `hour`, `minute`, `second`, `week`, `weeknum`, `dow`, `isoyear`, `isoweek`, `isoweekday`.

The key value of level with role `week` is expected to have format `YYYY-MM-DD`.

Schemas and Models

This section contains example database schemas and their respective models with description. The examples are for the SQL backend. Please refer to the backend documentation of your choice for more information about non-SQL setups.

See also:

Logical Model and Metadata Logical model description.

backends/index Backend references.

Model Reference Developer's reference of model classes and functions.

Basic Schemas

Simple Star Schema

Synopsis: Fact table has the same name as the cube, dimension tables have same names as dimensions.

Fact table is called *sales*, has one measure *amount* and two dimensions: *store* and *product*. Each dimension has two attributes.


```
"cubes": [
    {
        "name": "sales",
        "dimensions": ["product", "store"],
        "joins": [
            {"master":"product_id", "detail":"product.id"},
            {"master":"store_id", "detail":"store.id"}
        ]
    }
],
"dimensions": [
    { "name": "product", "attributes": ["code", "name"] },
    { "name": "store", "attributes": ["code", "address"] }
]
```

Simple Dimension

Synopsis: Dimension is represented only by one attribute, has no details, neither hierarchy.

Similar schema as *Simple Star Schema* Note the dimension *year* which is represented just by one numeric attribute.

It is important that no attributes are specified for the dimension. There dimension will be referenced just by its name and dimension label is going to be used as attribute label as well.


```
"cubes": [
    {
        "name": "sales",
        "dimensions": ["product", "store", "year"],
        "joins": [
            {"master":"product_id", "detail":"product.id"},
            {"master":"store_id", "detail":"store.id"}
        ]
    }
],
"dimensions": [
    { "name": "product", "attributes": ["code", "name"] },
    { "name": "store", "attributes": ["code", "address"] }
    { "name": "year" }
]
```

Table Prefix

Synopsis: dimension tables share a common prefix, fact tables share common prefix.

In our example the dimension tables have prefix `dim_` as in `dim_product` or `dim_store` and facts have prefix `fact_` as in `fact_sales`.

There is no need to change the model, only the data store configuration. In Python code we specify the prefix during the data store registration in `cubes.Workspace.register_store()`:

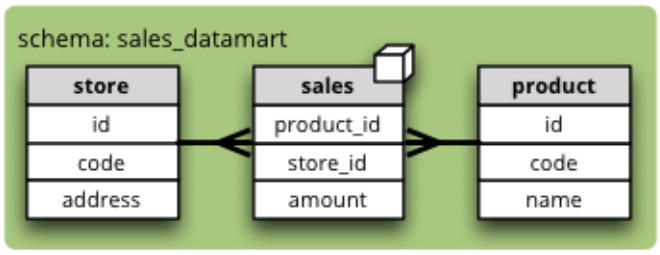
```
workspace = Workspace()
workspace.register_store("default", "sql",
                         url=DATABASE_URL,
                         dimension_prefix="dim_",
                         dimension_suffix="_dim",
                         fact_suffix="_fact",
                         fact_prefix="fact_")
```

When using the *OLAP Server* we specify the prefixes in the `[store]` section of the *slicer.ini* configuration file:

```
[store]
...
dimension_prefix="dim_"
fact_prefix="fact_"
```

Not Default Database Schema

Synopsis: all tables are stored in one common schema that is other than default database schema.



To specify database schema (in our example `sales_datamart`) in Python pass it in the *schema* argument of `cubes.Workspace.register_store()`:

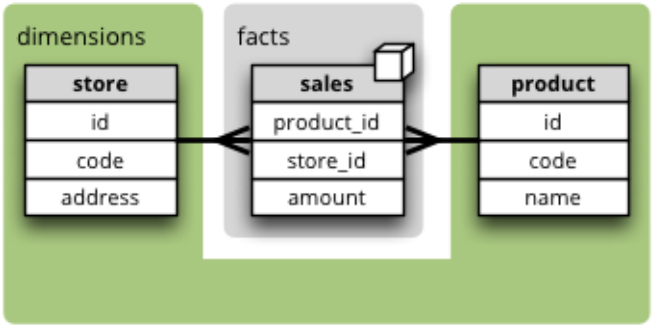
```
workspace = Workspace()
workspace.register_store("default", "sql",
                         url=DATABASE_URL,
                         schema="sales_datamart")
```

For the *OLAP Server* the schema is specified in in the `[store]` section of the *slicer.ini* configuration file:

```
[store]
...
schema="sales_datamart"
```

Separate Dimension Schema

Synopsis: dimension tables share one database schema and fact tables share another database schema



Dimensions can be stored in a different database schema than the fact table schema.

To specify database schema of dimensions (in our example `dimensions`) in Python pass it in the *dimension_schema* argument of `cubes.Workspace.register_store()`:

```
workspace = Workspace()
workspace.register_store("default", "sql",
                                    url=DATABASE_URL,
                                    schema="facts",
                                    dimension_schema="dimensions")
```

For the *OLAP Server* the dimension schema is specified in the `[store]` section of the *slicer.ini* configuration file:

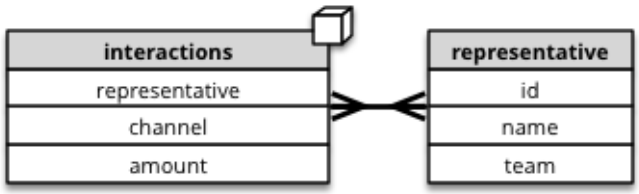
```
[store]
...
schema="facts"
dimension_schema="dimensions"
```

Many-to-Many Relationship

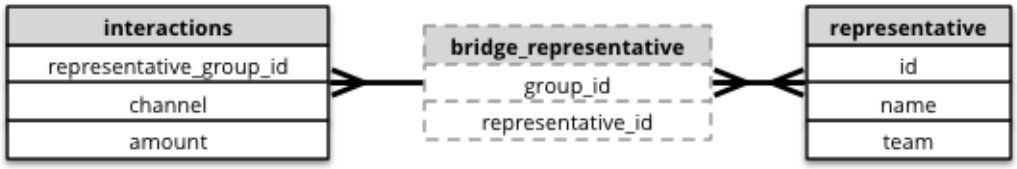
Synopsis: One fact might have multiple dimension members assigned

There are several options how the case of multiple dimension members per fact can be solved. Each has it advantages and disadvantages. Here is one of them: using a bridge table.

This is our logical intention: there might be multiple representatives involved in an interaction cases:



We can solve the problem with adding a bridge table and by creating artificial level *representative_group*. This group is unique combination of representatives that were involved in an interaction.



The model looks like:

```
"cubes": [
    {
        "dimensions": ["representative", ...],
        "joins": [
            {
                "master":"representative_group_id",
                "detail":"bridge_representative.group_id"
            },
            {
                "master":"bridge_representative.representative_id",
                "detail":"representative.id"
            }
        ]
    }
],
"dimensions": [
    {
        "name": "representative",
        "levels": [
```

```
            { "name":"team" },
            { "name":"name", "nonadditive": "any"}
        ]
    }
]
```

You might have noticed that the bridge table is hidden – you can't see it's contents anywhere in the cube.

There is one problem with aggregations when such dimension is involved: by aggregating over any level that is not the most detailed (deepest) we might get double (multiple) counting of the dimension members. For this reason it is important to specify all higher levels as *nonadditive* for `any` other dimension. It his case, backends that are aware of the issue, might handle it appropriately.

Some front-ends might not even allow to aggregate by levels that are marked as *nonadditivy*.

Mappings

Following patterns use the *Explicit Mapping*.

Basic Attribute Mapping

Synopsis: table column has different name than a dimension attribute or a measure.

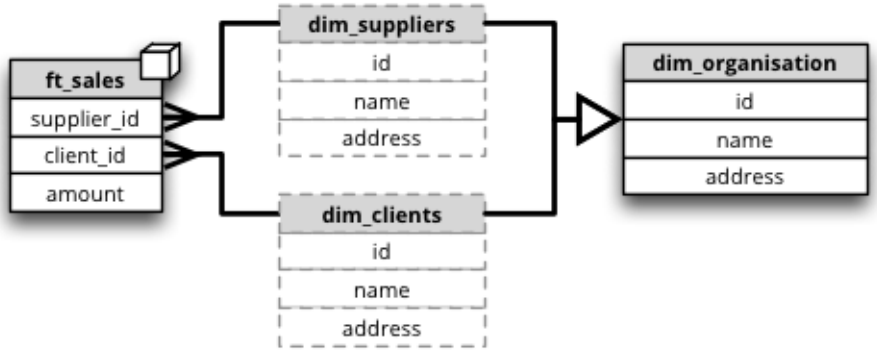
In our example we have a flat dimension called *year*, but the physical table column is "sales_year". In addition we have a measure *amount* however respective physical column is named *total_amount*.

We define the *mappings* within a cube:

```
"cubes": [
    {
        "dimensions": [..., "year"],
        "measures": ["amount"],
        "mappings": {
            "year":"sales_year",
            "amount":"total_amount"]
        }
    }
],
"dimensions": [
    ...
    { "name": "year" }
]
```

Shared Dimension Table

Synopsis: multiple dimensions share the same dimension table



Clients and suppliers might share one table with all organisations and companies. We have to specify a table alias in the *joins* part of the cube definition. The table aliases should follow the same naming pattern as the other tables – that is, if we are using dimension prefix, then the alias should include the prefix as well:

If the alias follows dimension naming convention, as in the example, then no mapping is required.

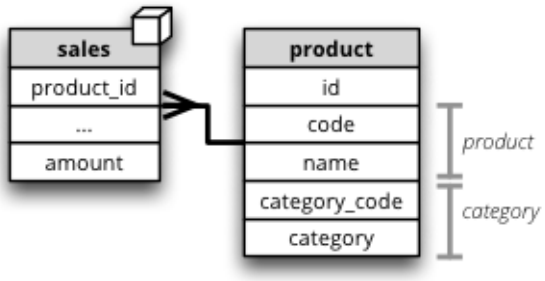
```
"cubes": [
    {
        "name": "sales"
        "dimensions": ["supplier", "client"],
        "measures": ["amount"],
        "joins": [
            {
                "master":"supplier_id",
                "detail":"dim_organisation.id",
                "alias":"dim_supplier"
            },
            {
                "master":"client_id",
                "detail":"dim_organisation.id",
                "alias":"dim_client"
            }
        ]
    }
],
"dimensions": [
    {
      "name": "supplier",
      "attributes": ["id", "name", "address"]
    },
    {
      "name": "client",
      "attributes": ["id", "name", "address"
    }
]
```

Hierarchies

Following patterns show how to specify one or multiple dimension hierarchies.

Simple Hierarchy

Synopsis: Dimension has more than one level.

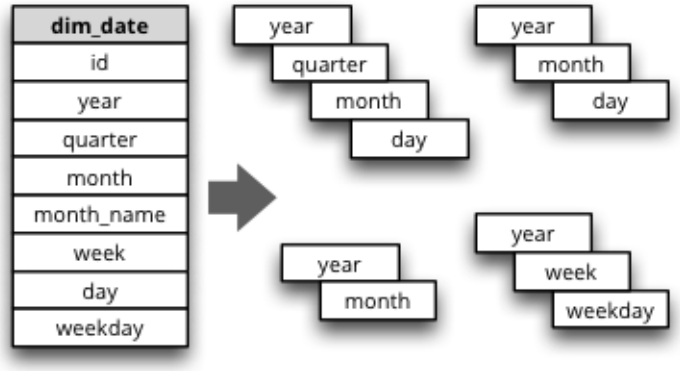


Product dimension has two levels: *product category* and *product*. The *product category* level is represented by two attributes `category_code` (as key) and `category`. The *product* has also two attributes: `product_code` and `name`.

```
"cubes": [
    {
        "dimensions": ["product", ...],
        "measures": ["amount"],
        "joins": [
            {"master":"product_id", "detail":"product.id"}
        ]
    }
],
"dimensions": [
    {
        "name": "product",
        "levels": [
            {
                "name":"category",
                "attributes": ["category_code", "category"]
            },
            {
                "name":"product",
                "attributes": ["code", "name"]
            }
        ]
    }
]
```

Multiple Hierarchies

Synopsis: Dimension has multiple ways how to organise levels into hierarchies.



Dimensions such as *date* (depicted below) or *geography* might have multiple ways of organizing their attributes into a hierarchy. The date can be composed of *year-month-day* or *year-quarter-month-day*.

To define multiple hierarchies, first define all possible levels. Then create list of hierarchies where you specify order of levels for that particular hierarchy.

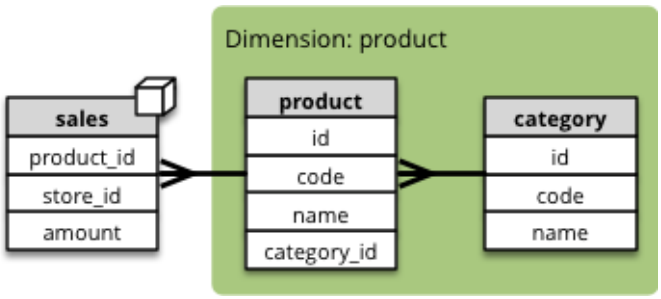
The code example below is in the "dimensions" section of the model:

```
{
    "name":"date",
    "levels": [
        { "name": "year", "attributes": ["year"] },
        { "name": "quarter", "attributes": ["quarter"] },
        { "name": "month", "attributes": ["month", "month_name"] },
        { "name": "week", "attributes": ["week"] },
        { "name": "weekday", "attributes": ["weekday"] },
        { "name": "day", "attributes": ["day"] }
    ],
    "hierarchies": [
        {"name": "ymd", "levels":["year", "month", "day"]},
        {"name": "ym", "levels":["year", "month"]},
        {"name": "yqmd", "levels":["year", "quarter", "month", "day"]},
        {"name": "ywd", "levels":["year", "week", "weekday"]}
    ],
    "default_hierarchy_name": "ymd"
}
```

The `default_hierarchy_name` specifies which hierarchy will be used if not mentioned explicitly.

Multiple Tables for Dimension Levels

Synopsis: Each dimension level has a separate table



We have to join additional tables and map the attributes that are not in the "main" dimension table (table with the same name as the dimension):

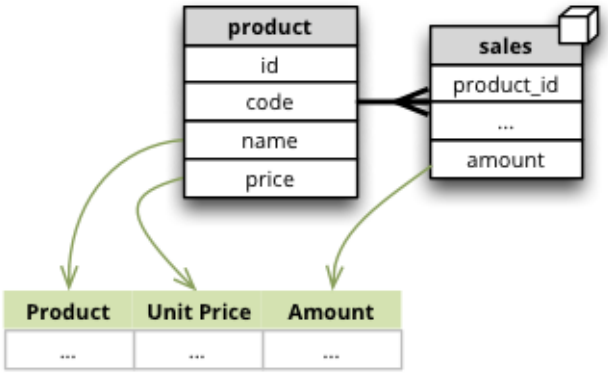
```
"cubes": [
    {
        "dimensions": ["product", ...],
        "measures": ["amount"],
        "joins": [
            {"master":"product_id", "detail":"product.id"},
            {"master":"product.category_id", "detail":"category.id"}
        ],
        "mappings": {
            "product.category_code": "category.code",
            "product.category": "category.name"
        }
    }
],
"dimensions": [
    {
        "name": "product",
        "levels": [
            {
                "name":"category",
                "attributes": ["category_code", "category"]
            },
            {
                "name":"product",
                "attributes": ["code", "name"]
            }
        ]
    }
]
```

Note: Joins should be ordered "from the master towards the details". That means that always join tables closer to the fact table before the other tables.

User-oriented Metadata

Model Labels

Synopsis: Labels for parts of model that are to be displayed to the user



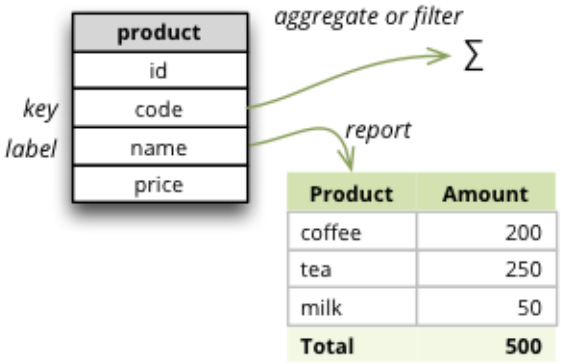
Labels are used in report tables as column headings or as filter descriptions. Attribute (and column) names should be used only for report creation and despite being readable and understandable, they should not be presented to the user in the raw form.

Labels can be specified for any model object (cube, dimension, level, attribute) with the *label* attribute:

```
"cubes": [
    {
        "name": "sales",
        "label": "Product Sales",
        "dimensions": ["product", ...]
    }
],
"dimensions": [
    {
        "name": "product",
        "label": "Product",
        "attributes": [
            {"name": "code", "label": "Code"},
            {"name": "name", "label": "Product"},
            {"name": "price", "label": "Unit Price"},
        ]
    }
]
```

Key and Label Attribute

Synopsis: specify which attributes are going to be used for filtering (keys) and which are going to be displayed in the user interface (labels)




```
"dimensions": [
    {
        "name": "product",
        "levels": [
            {
                "name": "product",
                "attributes": ["code", "name", "price"]
                "key": "code",
                "label_attribute": "name"
            }
        ]
    }
]
```

Example use:

```
result = browser.aggregate(drilldown=["product"])

for row in result.table_rows("product"):
   print "%s: %s" % (row.label, row.record["amount_sum"])
```

Localization

Localized Data

Synopsis: attributes might have values in multiple languages

Dimension attributes might have language-specific content. In cubes it can be achieved by providing one column per language (denormalized localization). The default column name should be the same as the localized attribute name with locale suffix, for example if the reported attribute is called *name* then the columns should be *name_en* for English localization and *name_hu* for Hungarian localization.

```
"dimensions": [
     {
          "name": "product",
          "label": "Product",
          "attributes": [
               {"name": "code", "label": "Code"},
               {
                    "name": "name",
                    "label": "Product",
                    "locales": ["en", "fr", "es"]
               }
          ]
     }
 ]
```

Use in Python:

```
browser = workspace.browser(cube, locale="fr")
```

The *browser* instance will now use only the French localization of attributes if available.

In slicer server requests language can be specified by the `lang=` parameter in the URL.

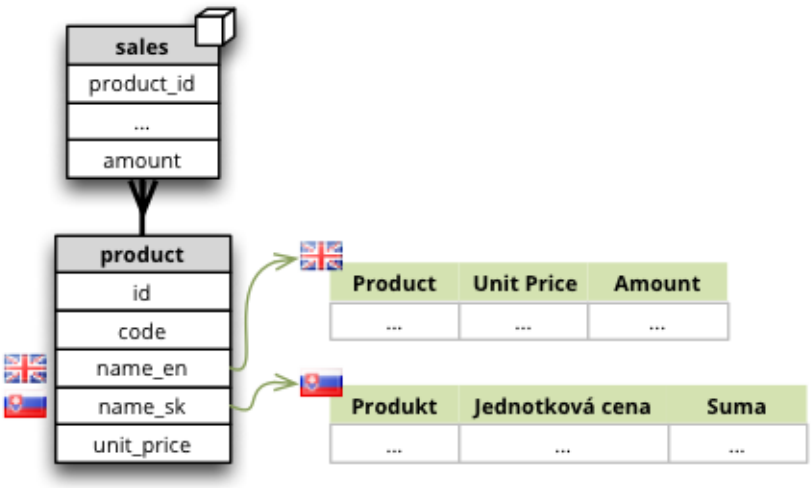
The dimension attributes are referred in the same way, regardless of localization. No change to reports is necessary when a new language is added.

Notes:

- only one locale per browser instance – either switch the locale or create another browser
- when non-existing locale is requested, then the default (first in the list of the localized attribute) locale is used

Localized Model Labels

Synopsis: Labels of model objects, such as dimensions, levels or attributes are localized.



Note: Way how model is localized is not yet decided, the current implementation might be changed.

We have a reporting site that uses two languages: English and Slovak. We want all labels to be available in both of the languages. Also we have a product name that has to be localized.

First we define the model and specify that the default locale of the model is English (for this case). Note the *locale* property of the model, the *label* attributes and the locales of *product.name* attribute:

```
{
    "locale": "en",
    "cubes": [
        {
            "name": "sales",
            "label": "Product Sales",
            "dimensions": ["product"],
            "measures": [
                {"name": "amount", "label": "Amount"}
            ]
        }
    ],
    "dimensions": [
        {
            "name": "product",
            "label": "Product",
            "attributes": [
                {
                  "name": "code",
                  "label": "Code"
                },
                {
                  "name": "name",
                  "label": "Product",
                  "locales": ["en", "sk"]
                },
                {
                  "name": "price",
                  "label": "Unit Price"
                }
            ]
        }
```

```
    ]
}
```

Next we create a separate translation dictionary for the other locale, in our case it is Slovak or `sk`. If we are translating only labels, no descriptions or any other information, we can use the simplified form:

```
{
   "locale": "sk",
   "dimensions":
   {
      "product”:
      {
           "levels":
           {
              "product" : "Produkt"
           },
           "attributes" :
           {
               "code": "Kód produktu",
               "name": "Produkt",
               "price": "Jednotková cena"
           }
      }
    },
    "cubes":
    {
        "sales":
        {
            "measures":
            {
                "amount": "Suma"
            }
        }
    }
}
```

Full localization with detailed dictionaries looks like this:

```
{
   "locale": "sk",
   "dimensions":
   {
      "product”:
      {
           "levels":
           {
              "product" : { "label" : "Produkt"}
           },
           "attributes" :
           {
               "code": {"label": "Kód produktu"},
               "name": {"label": "Produkt"},
               "price": {"label": "Jednotková cena"}
           }
      }
    },
    "cubes":
    {
        "sales":
        {
            "measures":
            {
```

```
                "amount": {"label": "Suma"}
            }
        }
    }
}
```

To create a model with translations:

```
translations = {"sk": "model-sk.json"}
model = create_model("model.json", translations)
```

The model created this way will be in the default locale. To get localized version of the master model:

```
localized_model = model.localize("sk")
```

Note: The `cubes.Workspace.browser()` method creates a browser with appropriate model localization, no explicit request for localization is needed.

Localization

Having origin in multi-lingual Europe one of the main features of the Cubes framework is ability to provide localizable results. There are three levels of localization in each analytical application:

1. Application level - such as buttons or menus
2. Metadata level - such as table header labels
3. Data level - table contents, such as names of categories or procurement types

Fig. 2.5: Localization levels.

The application level is out of scope of this framework and is covered in internationalization (i18n) libraries, such as *gettext*. What is covered in Cubes is metadata and data level.

Localization in cubes is very simple:

1. Create master model definition and specify locale the model is in
2. Specify attributes that are localized (see *Explicit Mapping*)
3. Create model translations for each required language
4. Make cubes function or a tool create translated versions the master model

To create localized report, just specify locale to the browser and create reports as if the model was not localized. See *Localized Reporting*.

Metadata Localization

The metadata are used to display report labels or provide attribute descriptions. Localizable metadata are mostly `label` and `description` metadata attributes, such as dimension label or attribute description.

Say we have three locales: Slovak, English, Hungarian with Slovak being the main language. The master model is described using Slovak language and we have to provide two model translation specifications: one for English and another for Hungarian.

The model translation file has the same structure as model definition file, but everything except localizable metadata attributes is ignored. That is, only `label` and `description` keys are considered in most cases. You can not change structure of mode in translation file. If structure does not match you will get warning or error, depending on structure change severity.

There is one major difference between master model file and model translations: all attribute lists, such as cube measures, cube details or dimension level attributes are dictionaries, not arrays. Keys are attribute names, values are metadata translations. Therefore in master model file you will have:

```
attributes = [
                { "name": "name", "label": "Name" },
                { "name": "cat", "label": "Category" }
            ]
```

in translation file you will have:

```
attributes = {
                "name": {"label": "Meno"},
                "cat": {"label": "Kategoria"}
            }
```

If a translation of a metadata attribute is missing, then the one in master model description is used.

In our case we have following files:

```
procurements.json
procurements_en.json
procurements_hu.json
```

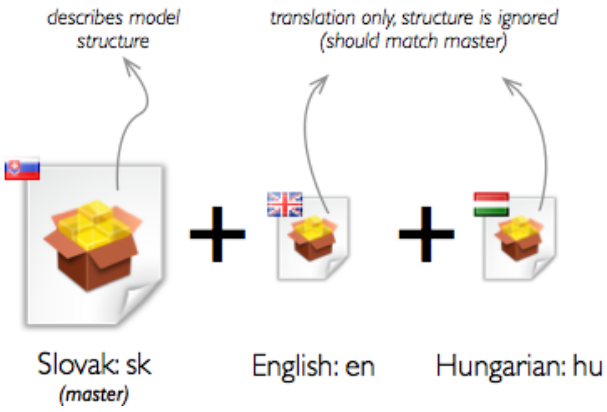


Fig. 2.6: Localization master model and translation files.

To add a model tranlsation:

```
workspace.add_translation("en", "procurements_en.json")
```

In the `slicer.ini`

```
[locale en]
default = procurements_en.json
```

```
[locale hu]
default = procurements_hu.json
```

To get translated version of a cube:

```
cube = workspace.cube("contracts", locale="en")
```

Localization is assigned to a model namespace.

Data Localization

If you have attributes that needs to be localized, specify the locales (languages) in the attribute definition in *Explicit Mapping*.

Note: Data localization is implemented only for Relational/SQL backend.

Localized Reporting

Main point of localized reporting is: *Create query once, reuse for any language*. Provide translated model and desired locale to the aggregation browser and you are set. The browser takes care of appropriate value selection.

Aggregating, drilling, getting list of facts - all methods return localized data based on locale provided to the browser. If you want to get multiple languages at the same time, you have to create one browser for each language you are reporting.

CHAPTER 3

Aggregation, Slicing and Dicing

Slicing and Dicing

Note: Examples are in Python and in Slicer HTTP requests.

Browser

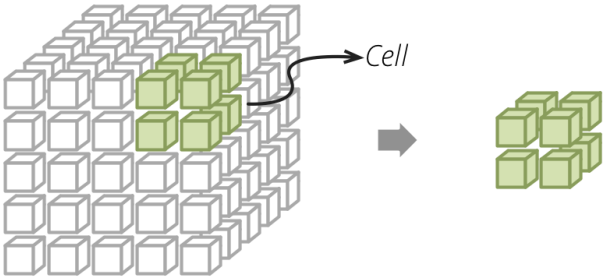
The aggregation, slicing, dicing, browsing of the multi-dimensional data is being done by an AggregationBrowser.

```
from cubes import Workspace

workspace = Workspace("slicer.ini")
browser = workspace.browser()
```

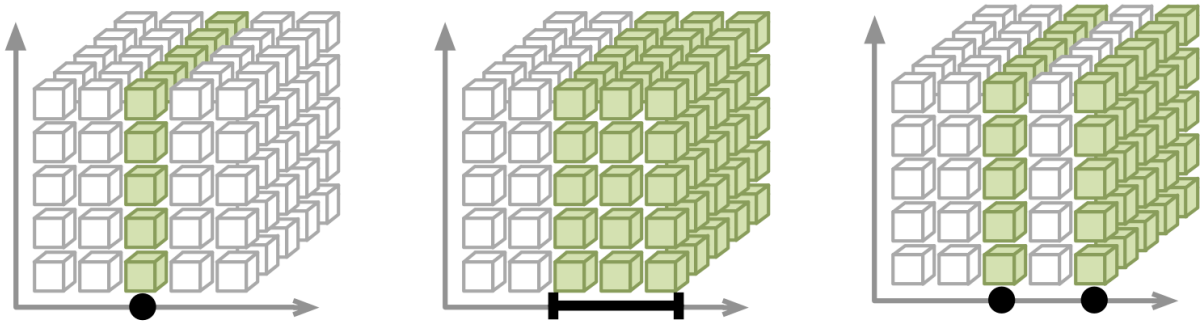
Cell and Cuts

Cell defines a point of interest – portion of the cube to be aggregated or browsed.



There are three types of cells: *point* – defines a single point in a dimension at a particular level; *range* – defines all points of an ordered dimension (such as date) within the range and *set* – collection of points:

Points are defined as dimension *paths* – list of dimension level keys. For example a date path for 24th of December 2010 would be: `[2010, 12, 24]`. For December 2010, regardless of day: `[2010, 12]` and for the whole



year: it would be a single item list `[2010]`. Similar for other dimensions: `["sk", "Bratislava"]` for city *Bratislava* in *Slovakia* (code `sk`).

In Python the cuts for "sales in Slovakia between June 2010 and June 2012" are defined as:

```
cuts = [
    PointCut("geography", ["sk"]),
    PointCut("date", [2010, 6], [2012, 6])
]
```

Same cuts for Slicer: `cut=geography:sk|date:2010,6-2012,6`.

If a different hierarchy than default is desired – "from the second quartal of 2010 to the second quartal of 2012":

```
cuts = [
    PointCut("date", [2010, 2], [2012, 2], hierarchy="yqmd")
]
```

Slicer: `cut=date@yqmd:2010,2-2012,2`.

Ranges and sets might have unequal depths: from `[2010]` to `[2012,12,24]` means "from the beginning of the year 2010 to December 24th 2012".

```
cuts = [
    PointCut("date", [2010], [2012, 12, 24])
]
```

Slicer: `cut=date:2010-2012,12,24`.

Ranges might be open, such as "everything until Dec 24 2012":

```
cuts = [
    PointCut("date", None, [2012, 12, 24])
]
```

Slicer: `cut=date:-2012,12,24`.

Aggregate

```
browser = workspace.browser("sales")
result = browser.aggregate()

print result.summary
```

Slicer: `/cube/sales/aggregate`

Aggregate of a cell:

```
cuts = [
    PointCut("geography", ["sk"])
    PointCut("date", [2010, 6], [2012, 6]),
]
cell = Cell(cube, cuts)
result = browser.aggregate(cell)
```

Slicer: `/cube/sales/aggregate?cut=geography:sk|date:2010,6-2012,6`

It is possible to select only specific aggregates to be aggregated:

```
result = browser.aggregate(cell, aggregates=["amount"])
```

Slicer: `/cube/sales/aggregate?aggregates=amount`

Drilldown

Drill-down – get more details, group the aggregation by dimension members.

For example "sales by month in 2010":

```
cut = PointCut("date", [2010])
cell = Cell(cube, [cut])
result = browser.aggregate(cell, drilldown=["date"])

for row in result:
    print "%s: %s" % (row["date.year"], row["amount_sum"])
```

Slicer: `/cube/sales/aggregate?cut=date:2010&drilldown=date`

Implicit

If not stated otherwise, the cubes drills-down to the next level of the drilled dimension. For example, if there is no cell constraint and the drilldown is *["date"]*, that means to use the first level of dimension date, usually *year*. If there is already a cut by year: *PointCut("date", [2010])* then the next level is by *month*.

The "next level" is determined as the next level after the deepest level used in a cut. Consider hierarchies for date: *year*, *month* and *day*, for geography: *region*, *country*, *city*. The implicit drilldown will be as follows:

Drilldown	Cut	Result levels
date	–	*date:year*
date	*date* point *[2010]*	*date:month*
date	*date* point *[2010, 4, 1]*	error
country, *date*	*date* range *[2010, 1] - [2010, 4]*	*date:day*, *geo:region*

If the cut is at its deepest level, then it is not possible to drill-down deeper which results in an error.

Explicit

If the implicit behavior is not satisfying, then the drill-down levels might be specified explicitly. In this case, the cut is not considered for the drilldown level.

You might want to specify drill-down levels explicitly for example if a cut range spans between multiple months and you don't want to have the next level to be *day*, but *month*. Another use is whe you want to use another hierarchy for drill-don than the default hierarchy.

Drilldown	Python	Server
by *year*	`("date", None, "year")`	`drilldown=date:year`
by *month* and *city*	`("date", None, "month"), ("geo", None, "city")`	`drilldown=date:month, geo:city`
by *month* but with quarter included	`("date", "yqmd", "month")`	`drilldown=date@yqmd:month`

Pagination

Results can be paginated by specifying *page* and *page_size* arguments:

```
result = browser.aggregate(cell, drilldown, page=0, page_size=10)
```

Server: `/cube/sales/aggregate?cell=...&drilldown=...&page=0&pagesize=10`

Split

Provisional:

- *aggregate(cell, drilldown, split)*

Facts

To get list of facts within a cell use `cubes.AggregationBrowser.facts()`:

```
facts = browser.facts(cell)
```

Server: `/cube/sales/facts?cell=...`

You can also paginate the result as in the aggregation.

Note that not all backends might support fact listing. Please refer to the backend's documentation for more information.

Fact

A single fact can be fetched using `cubes.AggregationBrowser.fact()` as in *fact(123)* or with the server as `/cube/sales/fact/123`.

Note that not all backends might support fact listing. Please refer to the backend's documentation for more information.

Members

Getting dimension members might be useful for example for populating drill-downs or for providing an information to the user what he can use for slicing and dicing. In python there is `cubes.AggregationBrowser.members()`.

For example to get all countries present in a cell:

```
members = browser.members(cell, "country")
```

Same query with the server would be: `/cube/sales/dimension/country?cut=...`

It is also possible to specify hierarchy and level depth for the dimension members.

Cell Details

When we are browsing a cube, the cell provides current browsing context. For aggregations and selections to happen, only keys and some other internal attributes are necessary. Those can not be presented to the user though. For example we have geography path (*country*, *region*) as `['sk', 'ba']`, however we want to display to the user *Slovakia* for the country and *Bratislava* for the region. We need to fetch those values from the data store. Cell details is basically a human readable description of the current cell.

For applications where it is possible to store state between aggregation calls, we can use values from previous aggregations or value listings. Problem is with web applications - sometimes it is not desirable or possible to store whole browsing context with all details. This is exact the situation where fetching cell details explicitly might come handy.

The cell details are provided by method `cubes.AggregationBrowser.cell_details()` which has Slicer HTTP equivalent `/cell` or `{"query":"detail", ...}` in `/report` request (see the *server documentation* for more information).

For point cuts, the detail is a list of dictionaries for each level. For example our previously mentioned path `['sk', 'ba']` would have details described as:

```
[
    {
        "geography.country_code": "sk",
        "geography.country_name": "Slovakia",
        "geography.something_more": "..."
        "_key": "sk",
        "_label": "Slovakia"
    },
    {
        "geography.region_code": "ba",
        "geography.region_name": "Bratislava",
        "geography.something_even_more": "...",
        "_key": "ba",
        "_label": "Bratislava"
    }
]
```

You might have noticed the two redundant keys: *_key* and *_label* - those contain values of a level key attribute and level label attribute respectively. It is there to simplify the use of the details in presentation layer, such as templates. Take for example doing only one-dimensional browsing and compare presentation of "breadcrumbs":

```
labels = [detail["_label"] for detail in cut_details]
```

Which is equivalent to:

```
levels = dimension.hierarchy().levels()
labels = []
for i, detail in enumerate(cut_details):
    labels.append(detail[levels[i].label_attribute.ref()])
```

Note that this might change a bit: either full detail will be returned or just key and label, depending on an option argument (not yet decided).

Supported Methods

Not all browsers might provide full functionality. For example a browser, such as Google Analytics, might provide aggregations, but might not provide fact details.

To learn what features are provided by the browser for particular cube use the `cubes.AggregationBrowser.features()` method which returns a dictionary with more detailed description of what can be done with the cube.

Data Formatters

Data and metadata from aggregation result can be transformed to one of multiple forms using formatters:

```
formatter = cubes.create_formatter("text_table")

result = browser.aggregate(cell, drilldown="date")

print formatter.format(result, "date")
```

Available formatters:

- *text_table* – text output for result of drilling down through one dimension
- *simple_data_table* – returns a dictionary with *header* and *rows*
- *simple_html_table* – returns a HTML table representation of result table cells
- *cross_table* – cross table structure with attributes *rows* – row headings, *columns* – column headings and *data* with rows of cells
- *html_cross_table* – HTML version of the *cross_table* formatter

See also:

Formatters Reference Formatter reference

CHAPTER 4

Analytical Workspace

Analytical Workspace

Analytical workspace is ... TODO: describe.

The analyital workspace manages cubes, shared (public) dimensions, data stores, model providers and model metadata. Provides aggregation browsers and maintains database connections.

Typical cubes session takes place in a workspace. Workspace is configured either through a `slicer.ini` file or programatically. Using the file:

```
from cubes import Workspace

workspace = Workspace(config="slicer.ini")
```

For more information about the configuration file options see *Configuration*

The manual workspace creation:

```
from cubes import Workspace

workspace = Workspace()
workspace.register_default_store("sql", url="postgresql://localhost/data")
workspace.import_model("model.json")
```

Stores

Cube data are stored somewhere or might be provided by a service. We call this data source a data *store*. A workspace might use multiple stores to get content of the cubes.

Built-in stores are:

- `sql` – relational database store (ROLAP) using star or snowflake schema
- `slicer` – connection to another Cubes server
- `mixpanel` – retrieves data from Mixpanel and makes it look like multidimensional cubes

Supported SQL dialects (by SQLAlchemy) are: Drizzle, Firebird, Informix, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, SQLite, Sybase

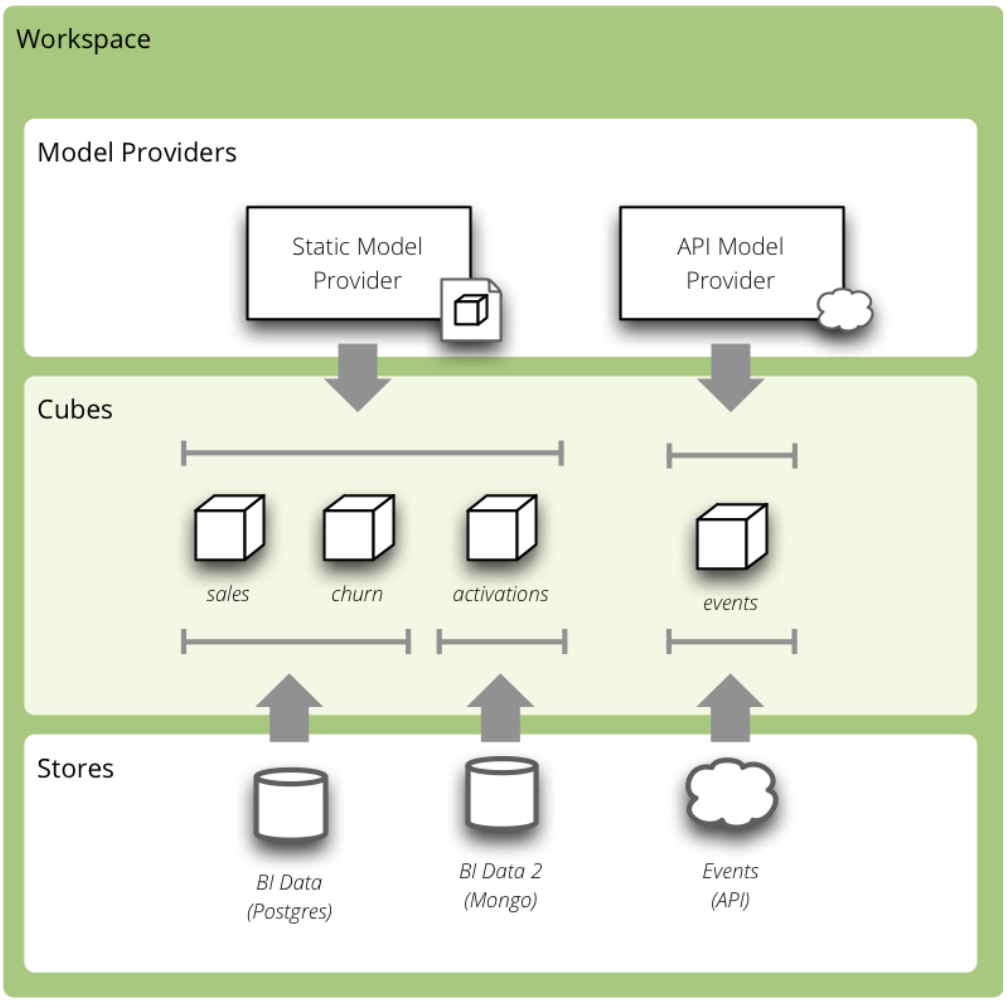


Fig. 4.1: Analytical Workspace

See *Configuration* for more information how to configure the stores.

Model Providers

Model provider creates models of cubes, dimensions and other analytical objects. The models can be created from a metadata, database or an external source, such as API.

Built-in model providers are:

- `static` (also aliased as `default`) – creates model objects from JSON data (files)
- `mixpanel` – describes cubes as Mixpanel events and dimensions as Mixpanel properties

To specify that the model is provided from other source than the metadata use the `provider` keyword in the model description:

```
{
    "provider": "mixpanel",
    "store": "mixpanel"
}
```

The store:

```
[store]
type: mixpanel
api_key: MY_MIXPANEL_API_KEY
api_secret: MY_MIXPANEL_API_SECRET
```

Authorization and Authentication

Cubes provides simple but extensible mechanism for authorization through an *Authorizer* and for authentication through an *Authenticator*.

Authentication in cubes: determining and confirirming the user's identity, for example using a user name and password, some secret key or using an external service.

Authorization: providing (or denying) access to cubes based on user's identity.

Authorization

The authorization principle in cubes is based on user's rights to a cube and restriction within a cube. If user has a "right to a cube" he can access the cube, the cube will be visible to him.

Restriction within a cube is cell based: users might have access only to a certain cell within a cube. For example a shop manager might have access only to sales cube and dimension point equal to his own shop.

Authorization is configured at the workspace level. In `slicer.ini` it is specified as:

```
[workspace]
authorization: simple

[authorization]
rights_file: access_rights.json
```

There is only one build-in authorizer called `simple`.

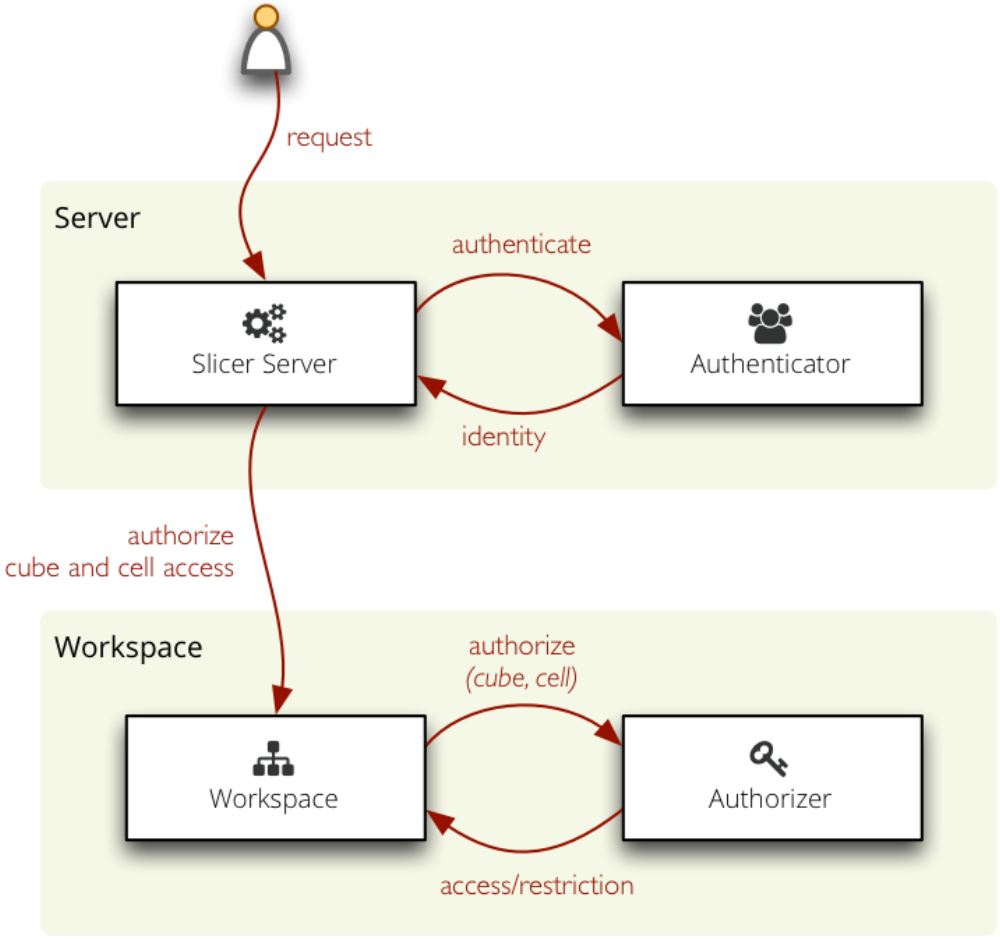


Fig. 4.2: Overview of authorization and authentication process in Slicer

Simple Authorization

Simple authorization based on JSON files: *rights* and *roles*. The *rights* file contains a dictionary with keys as user identities (user names, API keys, ...) and values as right descriptions.

The user right is described as:

- `roles` – list of of user's role – user inherits the restrictions from the role
- `allowed_cubes` – list of cubes that the user can access (and no other cubes)
- `denied_cubes` – list of cubes that the user can not access (he can access the rest of cubes)
- `cube_restrictions` – a dictionary where keys are cube names and values are lists of cuts

The roles file has the same structure as the rights file, instead of users it defines inheritable roles. The roles can inherit properties from other roles.

Example of roles file:

```
{
    "retail": {
        "allowed_cubes": ["sales"]
    }
}
```

Rights file:

```
{
    "martin": {
        "roles": ["retail"],
    }
}
```

The rights file of the simple authorization method might contain a special guest role which will be used when no other identity is found. See the configuration documentation for more information.

Authentication

Authentication is handled at the server level.

Built-in authentication methods:

- `none` – no authentication
- `pass_parameter` – permissive authentication that just passes an URL parameter to the authorizer. Default parameter name is `api_key`
- `http_basic_proxy` – permissive authentication using HTTP Basic method. Assumes that the slicer is behind a proxy and that the password was already verified. Passes the user name to the authorizer.

Configuration

Cubes workspace configuration is stored in a `.ini` file with sections:

- `[server]` - server related configuration, such as host, port
- `[workspace]` – Cubes workspace configuration
- `[model]` - model and cube configuration
- `[models]` - list of models to be loaded (deprecated)
- `[naming]` - naming conventions

- `[store]` – default datastore configuration
- `[store NAME]` – configuration for store with name *NAME*
- `[locale NAME]` - model translations. See *Localization* for more information.
- `[info]` - optional section for user presentable info about your project

Note: The configuration has changed with version 1.0. Since Cubes supports multiple data stores, their type (backend) is specified in the store configuration as `type` property, for example `type=sql`.

Quick Start

Simple configuration might look like this:

```
[workspace]
model = model.json

[store]
type = sql
url = postgresql://localhost/database
```

Server

`json_record_limit`

Number of rows to limit when generating JSON output with iterable objects, such as facts. Default is 1000. It is recommended to use alternate response format, such as CSV, to get more records.

`modules`

Space separated list of modules to be loaded. This is only used if run by the *slicer command*.

`prettyprint`

If set to `true`, JSON is serialized with indentation of 4 spaces. Set to `true` for demonstration purposes, omit or comment out option for production use.

`host`

Host or IP address where the server binds, defaults to `localhost`.

`port`

Port on which the server listens, defaults to `5000`.

`reload`

Suitable for development only. Set to `yes` to enable Werkzeug reloader.

`allow_cors_origin`

Cross-origin resource sharing header. Other related headers are added as well, if this option is present.

authentication

Authentication method, see *Authentication and Authorization* below for more information.

pid_file

Path to a file where PID of the running server will be written. If not provided, no PID file is created.

Workspace

This section covers the Workspace configuration, such as file locations, logging, namespaces and localization.

Authorization

authorization

Authorization method to be used on the workspace side. If omitted, no authorization is required. For details see *Authentication and Authorization* below.

Localization configuration

timezone

Name of the default time zone, for example `Europe/Berlin`. Used in date and time operations, such as *named relative time*.

first_weekday

First day of the week in english weekday name. Can also be specified as number, where 0 is Monday and 6 is Sunday.

File Locations

root_directory

Workspace root path: all paths, such as `models_directory` or `info_file` are considered relative to the `root_directory` it they are not specified as absolute.

models_directory

Path to a directory containing models. If this is set to non-empty value, then all model paths specified in `[models]` are prefixed with this path.

stores_file

Path to a file (with *.ini* config syntax) containing store descriptions – every section is a store with same name as the section.

info_file

Path to a file containing user info metadata. See more in *Info*.

Logging configuration

log

Path to log file.

log_level

Level of log details, from least to most: `error`, `warn`, `info`, `debug`.

Namespaces

If not specified otherwise, all cubes share the same default namespace. Their names within namespace should be unique.

Model

path

Path to model .json file. See *Logical Model and Metadata* for more on model definition.

Models

Warning: This section is deprecated in favor of section `[model]`.

Section `[models]` contains list of models. The property names are model identifiers within the configuration (see `[translations]` for example) and the values are paths to model files.

Example:

```
[models]
main = model.json
mixpanel = mixpanel.json
```

If *models_directory* is specified in *Workspace* then the relative model paths are combined with the *models_directory*. Example:

```
[workspace]
models_directory = /dwh/cubes/models

[models]
main = model.json
events = events.json
```

The models are loaded from `/dwh/cubes/models/model.json` and `/dwh/cubes/models/events.json`.

Note: If the *root_directory* is set, then the `models_directory` is relative to the `root_directory`. For example if the workspace root is `/var/lib/cubes` and `models_directory` is `models` then the search

path for models will be `/var/lib/cubes/models`. If the `models_directory` is absolute, for example `/cubes/models` then the absolute path will be used regardless of the workspace root directory settings.

Data stores

There might be one or more store configured. The section `[store]` of the `cubes.ini` file describes the default store. Multiple stores are configured in a separate `stores.ini` file referenced by the *stores_file* configuration option in `[workspace]` section.

Data store properties

type

Defines the data store backend module used, eg. `sql`. Required.

For list of available types see backends/index.

model

Model related to the datastore.

namespace

Namespace where the store's cubes will be registered.

model_provider

Model provider type for the datastore. For more on model providers, see chapter *Model Provider and External Models*.

Example data store configurations

Example SQL store:

```
[store]
type = sql
url = postgresql://localhost/data
schema = cubes
```

For more information and configuration on SQL store options see *SQL Backend*.

Example mixpanel store:

```
[store]
type = mixpanel
model = mixpanel.json
api_key = 123456abcd
api_secret = 12345abcd
```

Multiple *Slicer* stores:

```
[store slicer1]
type = slicer
url = http://some.host:5000

[store slicer2]
type = slicer
url = http://other.host:5000
```

The cubes will be named *slicer1.** and *slicer2.**. To use specific namespace, different from the store name:

```
[store slicer3]
type = slicer
namespace = external
url = http://some.host:5000
```

Cubes will be named *external.**

To specify default namespace:

```
[store slicer4]
type = slicer
namespace = default.
url = http://some.host:5000
```

Cubes will be named without namespace prefix.

Naming

Todo

Write the naming section.

```
[naming]
dimension_prefix = dim_
fact_prefix = ft_
```

See respective backend documentation for more information about naming conventions in the `[naming]` section.

Authentication and Authorization

Cubes provides mechanisms for authentication at the server side and authorization at the workspace side.

Authorization

To configure authorization, you need to enable authorization in workspace section.

```
[workspace]
authorization = simple

[authorization]
rights_file = /path/to/access_rights.json
```

authorization

This option goes in the `[workspace]` section.

Valid options are

- `none` – no authorization
- `simple` – uses a JSON file with per-user access rights

Simple authorization

The simple authorization has following configuration options:

rights_file

Path to the JSON configuration file with access rights.

roles_file

Path to the JSON configuration file with roles.

identity_dimension

Name of a flat dimension that will be used for cell restriction. Key of that dimension should match the identity.

order

Access control. Valid is `allow_deny` or `deny_allow` (default).

guest

Name of a guest role. If specified, then this role will be used for all unknown (not specified in the file) roles.

Authentication

Example authentication via parameter passing:

```
[server]
authentication = pass_parameter

[authentication]
# additional authentication parameters
parameter = token
```

This configures server to expect a GET parameter `token` which will be passed on to authorization.

authentication

Built-in server authentication methods:

`none`

No authentication.

`http_basic_proxy`

HTTP basic authentication will pass the *username* to the authorizer. This assumes the server is behind a proxy and that the proxy authenticated the user.

`pass_parameter`

Authentication without verification, just a way of passing an URL parameter to the authorizer. Parameter name can be specified via `parameter` option, default `api_key`.

For more on how this works, see *Authorization and Authentication*.

Note: When you have authorization method specified and is based on an users's indentity, then you have to specify the authentication method in the server. Otherwise the authorizer will not receive any identity and might refuse any access.

Localization sections

Model localizations are specified in the configuration with `[locale XX]` where `XX` is the two letter ISO 639-1 locale code. Option names are namespace names and option keys are paths to translation files. For example:

```
[locale sk]
default = translation_sk.json

[locale hu]
default = translation_hu.json
```

Info

This section contains user supplied and front-end presentable information such as description or license. This can be included in main .ini configuration or as a separate JSON file.

The info JSON file might contain:

- `label` – server's name or label
- `description` – description of the served data
- `copyright` – copyright of the data, if any
- `license` – data license
- `maintainer` – name of the data maintainer, might be in format *Name Surname <namesurname@domain.org>*
- `contributors` - list of contributors
- `keywords` – list of keywords that describe the data
- `related` – list of related or "friendly" Slicer servers with other open data – a dictionary with keys `label` and `url`.
- `visualizers` – list of links to prepared visualisations of the server's data – a dictionary with keys `label` and `url`.

Server Query Logging

Sections, prefixed with *query_log* configure query logging. All sections with this prefix (including section named as the prefix) are collected and chained into a list of logging handlers. Required option is *type*. You might have multiple handlers at the same time.

Configuration options are:

type

Type of query log. Required.

Valid options are:

`default`

Log using Cubes logger via Python logging module.

`csv_file`

Log into a CSV file. Specify the file name via `path` option.

`json`

Log into file as quasi-JSON file - each log record is valid JSON and records are separated by newlines. Specify the file name via `path` option.

`sql`

Log into a SQL table. SQL request logger options are:

- *url* – database URL
- *table* – database table
- *dimensions_table* – table with dimension use (optional)

If tables do not exist, they are created automatically.

Example query log configuration

This configuration will create three query loggers, all at once. *query_log_one* will emit to Python logging and will show in console if *log_level* is set to `info` or more verbose. *query_log_two* will log queries into CSV file /var/log/cubes/queries.csv. *query_log_three* will insert query log into table *cubes_query_log* in a PostgreSQL database named *cubes_log* located on a remote host named *log_host*.

```
[query_log_one]
type = default

[query_log_two]
type = csv
path = /var/log/cubes/queries.csv

[query_log_three]
type = sql
url = postgresql://log_host/cubes_log
table = cubes_query_log
```

Examples

Simple configuration:

```
[workspace]
model = model.json

[store]
type = sql
url = postgresql://localhost/cubes
```

Multiple models, one store:

```
[models]
finance = finance.cubesmodel
customer = customer.cubesmodel

[store]
type = sql
url = postgresql://localhost/cubes
```

Multiple stores:

```
[store finance]
type = sql
url = postgresql://localhost/finance
model = finance.cubesmodel

[store customer]
type = sql
url = postgresql://otherhost/customer
model = customer.cubesmodel
```

Example of a whole configuration file:

```
[workspace]
model = ~/models/contracts_model.json

[server]
log = /var/log/cubes.log
log_level = info

[store]
type = sql
url = postgresql://localhost/data
schema = cubes
```

SQL Backend

The SQL backend is using the SQLAlchemy which supports following SQL database dialects:

- Drizzle
- Firebird
- Informix
- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL
- SQLite
- Sybase

Supported aggregate functions:

- *sum*
- *count* – equivalend to `COUNT(1)`
- *count_nonempty* – equivalent to `COUNT(measure)`
- *count_distinct* – equivalent to `COUNT(DISTINCT measure)`

- *min*
- *max*
- *avg*
- *stddev*
- *variance*

Store Configuration

Data store:

- `url` *(required)* – database URL in form: `adapter://user:password@host:port/database`, for example: `postgresql://stefan:secret@localhost:5432/datawarehouse`. Empty values can be ommited, as in `postgresql://localhost/datawarehouse`.
- `schema` *(optional)* – schema containing denormalized views for relational DB cubes

Database Connection

(advanced topic)

To fine-tune the SQLAlchemy database connection some of the *create_engine()* parameters can be specified as `sqlalchemy_PARAMETER`:

- `sqlalchemy_case_sensitive`
- `sqlalchemy_convert_unicode`
- `sqlalchemy_pool_size`
- `sqlalchemy_pool_recycle`
- `sqlalchemy_pool_timeout`
- `sqlalchemy_...` ...

Please refer to the create_engine documentation for more information.

Naming

The following configuration settings might be used in the naming conventions configuration:

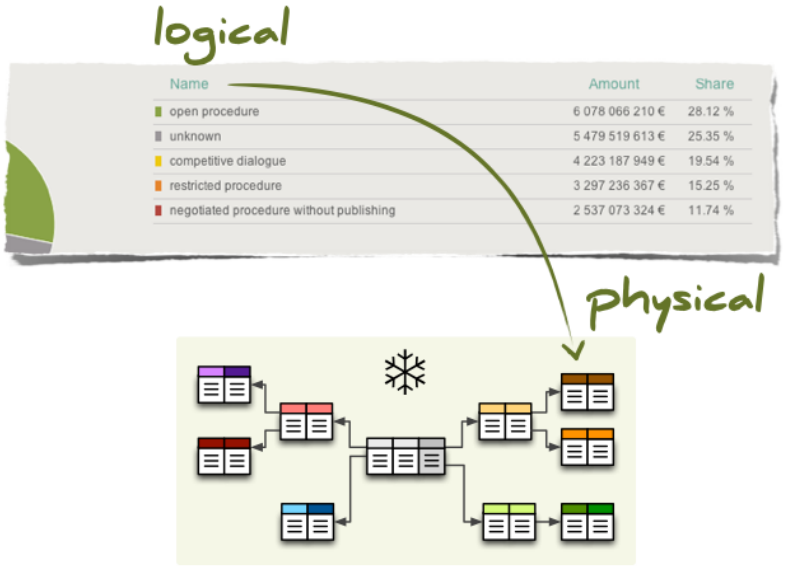
- `dimension_prefix` *(optional)* – used by snowflake mapper to find dimension tables when no explicit mapping is specified
- `dimension_suffix` *(optional)* – used by snowflake mapper to find dimension tables when no explicit mapping is specified
- `dimension_schema` – use this option when dimension tables are stored in different schema than the fact tables
- `fact_prefix` *(optional)* – used by the snowflake mapper to find fact table for a cube, when no explicit fact table name is specified
- `fact_suffix` *(optional)* – used by the snowflake mapper to find fact table for a cube, when no explicit fact table name is specified
- `use_denormalization` *(optional)* – browser will use dernormalized view instead of snowflake
- `denormalized_view_prefix` *(optional, advanced)* – if denormalization is used, then this prefix is added for cube name to find corresponding cube view
- `denormalized_view_schema` *(optional, advanced)* – schema wehere denormalized views are located (use this if the views are in different schema than fact tables, otherwise default schema is going to be used)

Model Requirements

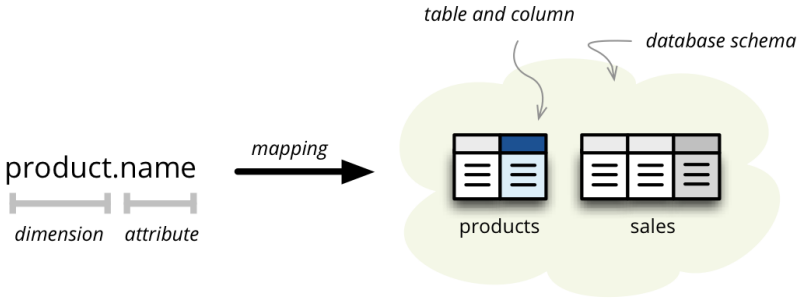
Cube has to have `key` property set to the fact table key column to be able to provide list of facts. Default key is `id`.

Mappings

One of the important parts of proper OLAP on top of the relational database is the mapping of logical attributes to their physical counterparts. In SQL database the physical attribute is stored in a column, which belongs to a table, which might be part of a database schema.



For example, take a reference to an attribute *name* in a dimension *product*. What is the column of what table in which schema that contains the value of this dimension attribute?



For data browsing, the Cubes framework has to know where those logical (reported) attributes are physically stored. It needs to know which tables are related to the cube and how they are joined together so we get whole view of a fact.

The process is done in two steps:

1. joining relevant star/snowflake tables
2. mapping logical attribute to table + column

There are two ways how the mapping is being done: *implicit* and *explicit*. The simplest, straightforward and most customizable is the explicit way, where the actual column reference is provided in a mapping dictionary of the cube description.

Implicit Mapping

With implicit mapping one can match a database schema with logical model and does not have to specify additional mapping metadata. Expected structure is star schema with one table per (denormalized) dimension.

Facts

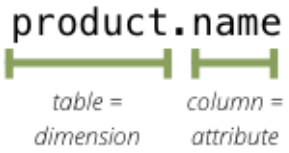
Cubes looks for fact table with the same name as cube name. You might specify prefix for every fact table with `fact_table_prefix`. Example:

- Cube is named *contracts*, framework looks for a table named *contracts*.
- Cubes are named *contracts*, *invoices* and fact table prefix is `fact_` then framework looks for tables named `fact_contracts` and `fact_invoices` respectively.

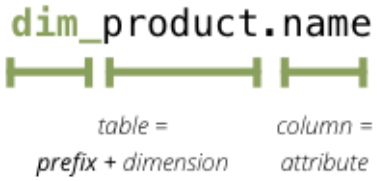
Dimensions

In short: a dimension attribute *customer.name* maps to table *customer* and column *name* by default. A dimension without details and with just a single level such as *is_hungry* maps to the *is_hungry* column of the fact table.

By default, dimension tables are expected to have same name as dimensions and dimension table columns are expected to have same name as dimension attributes:



It is quite common practice that dimension tables have a prefix such as `dim_` or `dm_`. Such prefix can be specified with `dimension_prefix` option.



The rules are:

- fact table should have same name as represented cube: *fact table name = fact table prefix + fact table name*
- dimension table should have same name as the represented dimension, for example: *product* (singular): *dimension table name = dimension prefix + dimension name*
- column name should have same name as dimension attribute: *name*, *code*, *description*
- references without dimension name in them are expected to be in the fact table, for example: *amount*, *discount* (see note below for simple flat dimensions)
- if attribute is localized, then there should be one column per localization and should have locale suffix: *description_en*, *description_sk*, *description_fr* (see below for more information)

Flat dimension without details

What about dimensions that have only one attribute, like one would not have a full date but just a *year*? In this case it is kept in the fact table without need of separate dimension table. The attribute is treated in by the same rule as measure and is referenced by simple *year*. This is applied to all dimensions that have only one attribute (representing key as well). This dimension is referred to as *flat and without details*.

Note: The simplification of the flat references can be disabled by setting `simplify_dimension_references` to `False` in the mapper. In that case you will have to have separate table for the dimension attribute and you will have to reference the attribute by full name. This might be useful when you know that your dimension will be more detailed.

Database Schemas

For databases that support schemas, such as PostgreSQL, option `schema` can be used to specify default database schema where all tables are going to be looked for.

In case you have dimensions stored in separate schema than fact table, you can specify that in `dimension_schema`. All dimension tables are going to be searched in that schema.

Explicit Mapping

If the schema does not match expectations of cubes, it is possible to explicitly specify how logical attributes are going to be mapped to their physical tables and columns. *Mapping dictionary* is a dictionary of logical attributes as keys and physical attributes (columns, fields) as values. The logical attributes references look like:

- *dimensions_name.attribute_name*, for example: `geography.country_name` or `category.code`
- *fact_attribute_name*, for example: `amount` or `discount`

Following mapping maps attribute *name* of dimension *product* to the column *product_name* of table *dm_products*.

```
"mappings": {
    "product.name": "dm_products.product_name"
}
```

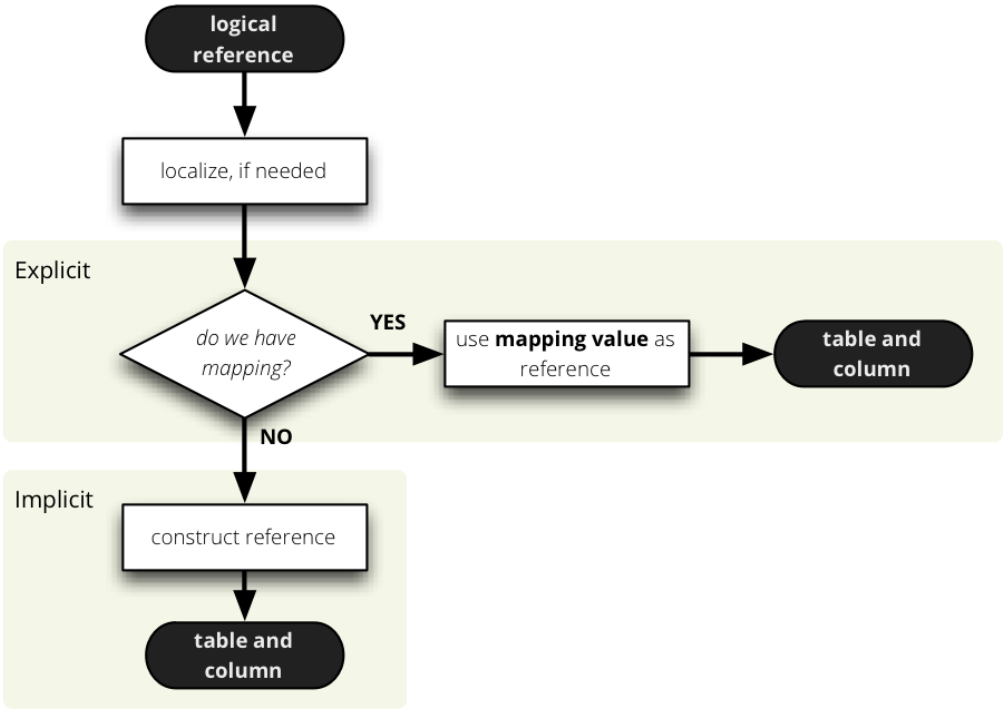
Note: Note that in the mappings the table names should be spelled as they are in the database even the table prefix is specified.

If it is in different schema or any part of the reference contains a dot:

```
"mappings": {
    "product.name": {
            "schema": "sales",
            "table": "dm_products",
            "column": "product_name"
        }
}
```

Both, explicit and implicit mappings have ability to specify default database schema (if you are using Oracle, PostgreSQL or any other DB which supports schemas).

The mapping process process is like this:



Date Data Type

Date datatype column can be turned into a date dimension by extracting date parts in the mapping. To do so, for each date attribute specify a `column` name and part to be extracted with value for `extract` key.

```
"mappings": {
  "date.year": {"column":"date", "extract":"year"},
  "date.month": {"column":"date", "extract":"month"},
  "date.day": {"column":"date", "extract":"day"}
}
```

According to SQLAlchemy, you can extract in most of the databases: `month`, `day`, `year`, `second`, `hour`, `doy` (day of the year), `minute`, `quarter`, `dow` (day of the week), `week`, `epoch`, `milliseconds`, `microseconds`, `timezone_hour`, `timezone_minute`. Please refer to your database engine documentation for more information.

Note: It is still recommended to have a date dimension table.

Localization

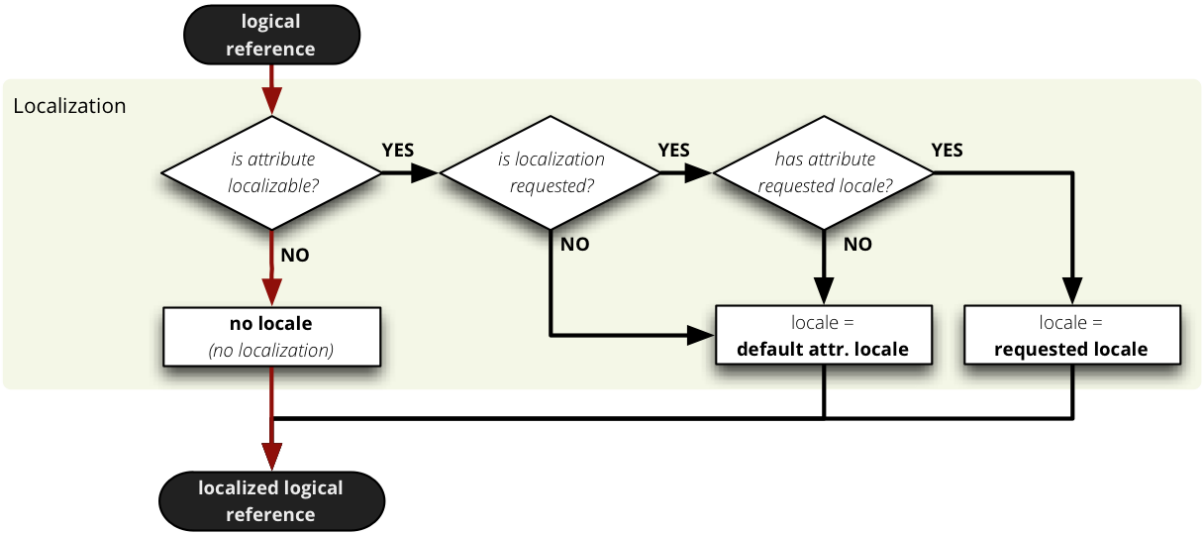
From physical point of view, the data localization is very trivial and requires language denormalization - that means that each language has to have its own column for each attribute.

Localizable attributes are those attributes that have `locales` specified in their definition. To map logical attributes which are localizable, use locale suffix for each locale. For example attribute *name* in dimension *category* has two locales: Slovak (`sk`) and English (`en`). Or for example product category can be in English, Slovak or German. It is specified in the model like this:

```
attributes = [
    {
        "name" = "category",
        "locales" = ["en", "sk", "de"]
```

```
    }
]
```

During the mapping process, localized logical reference is created first:



In short: if attribute is localizable and locale is requested, then locale suffix is added. If no such localization exists then default locale is used. Nothing happens to non-localizable attributes.

For such attribute, three columns should exist in the physical model. There are two ways how the columns should be named. They should have attribute name with locale suffix such as `category_sk` and `category_en` (_underscore_ because it is more common in table column names), if implicit mapping is used. You can name the columns as you like, but you have to provide explicit mapping in the mapping dictionary. The key for the localized logical attribute should have `.locale` suffix, such as `product.category.sk` for Slovak version of category attribute of dimension product. Here the _dot_ is used because dots separate logical reference parts.

Note: Current implementation of Cubes framework requires a star or snowflake schema that can be joined into fully denormalized normalized form just by simple one-key based joins. Therefore all localized attributes have to be stored in their own columns. In other words, you have to denormalize the localized data before using them in Cubes.

Read more about *Localization*.

Mapping Process Summary

Following diagram describes how the mapping of logical to physical attributes is done in the star SQL browser (see `cubes.backends.sql.StarBrowser`):

The "red path" shows the most common scenario where defaults are used.

Joins

The SQL backend supports a star:

and a snowflake database schema:

If you are using either of the two schemas (star or snowflake) in relational database, Cubes requires information on how to join the tables. Tables are joined by matching single-column – surrogate keys. The framework needs the join information to be able to transform following snowflake:

to appear as a denormalized table with all cube attributes:

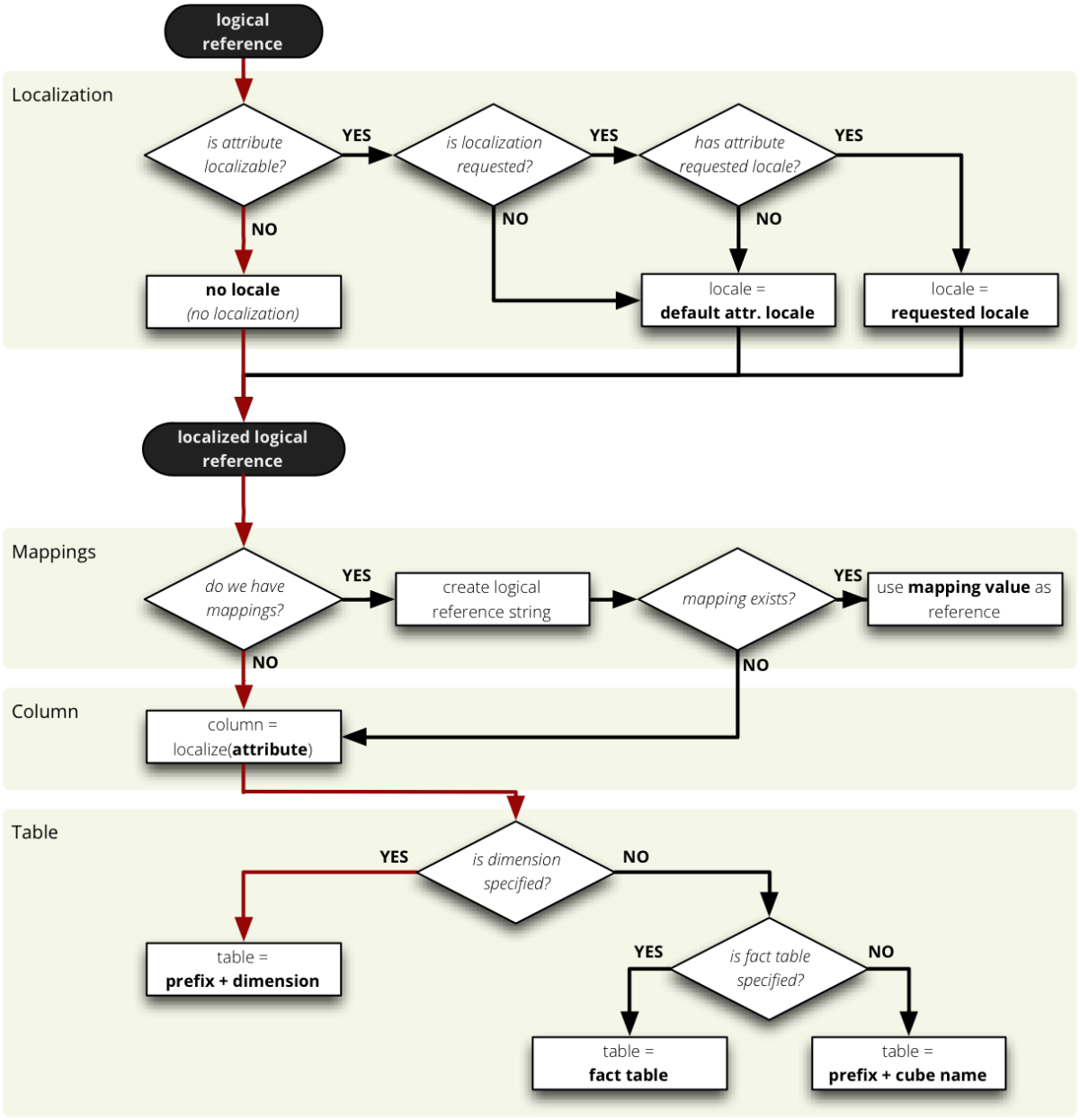
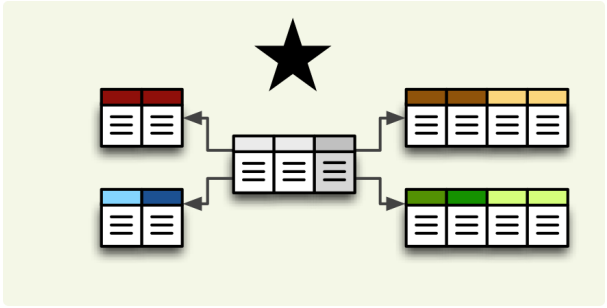
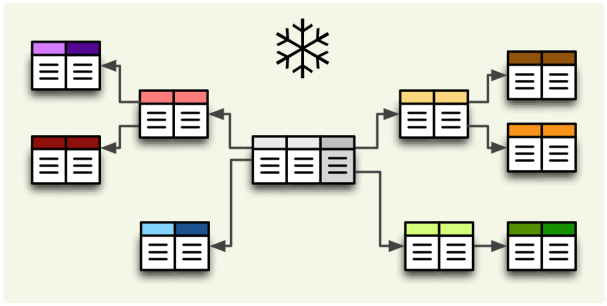
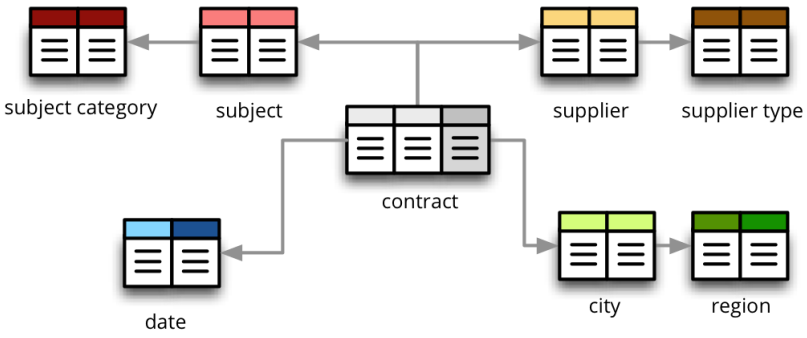


Fig. 4.3: logical to physical attribute mapping







Note: The SQL backend performs only joins that are relevant to the given query. If no attributes from a table are used, then the table is not joined.

Join Description

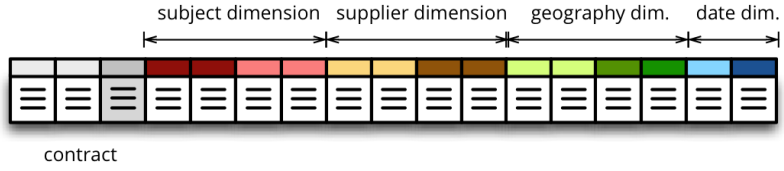
Joins are defined as an ordered list (order is important) for every cube separately. The join description consists of reference to the *master* table and a table with *details*. Fact table is example of master table, dimension is example of a detail table (in a star schema).

The join specification is very simple, you define column reference for both: master and detail. The table reference is in the form *table*.'column':

```
"joins" = [
    {
        "master": "fact_sales.product_key",
        "detail": "dim_product.key"
    }
]
```

As in mappings, if you have specific needs for explicitly mentioning database schema or any other reason where *table.column* reference is not enough, you might write:

```
"joins" = [
    {
        "master": "fact_sales.product_id",
        "detail": {
            "schema": "sales",
            "table": "dim_products",
```



```
            "column": "id"
        }
]
```

To specify a compound join key, the `column` value of a join specified as a dictionary can be an array denoting multiple keys. The above join would be specified as:

```
{
    "master": {
        "table": "fact_table",
        "column": ["dimension_id", "partition"]
    },
    "detail": {
        "table": "dimension",
        "column": ["id", "partition"]
    }
}
```

This will generate the following join:

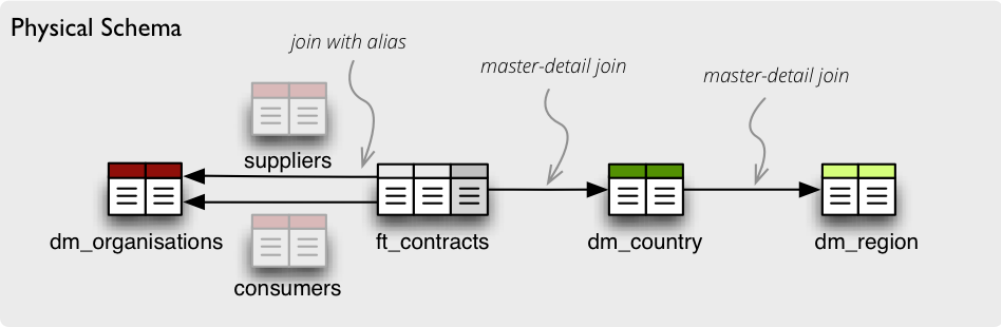
```
FROM fact_table
INNER JOIN fact_table ON (
   fact_table.dimension_id = dimension_table.id
  AND fact_table.partition = dimension.partition
)
```

Join Order

Order of joins has to be from the master tables towards the details.

Aliases

What if you need to join same table twice or more times? For example, you have list of organizations and you want to use it as both: supplier and service consumer.



It can be done by specifying alias in the joins:

```
"joins" = [
    {
        "master": "contracts.supplier_id",
        "detail": "organisations.id",
        "alias": "suppliers"
    },
    {
        "master": "contracts.consumer_id",
        "detail": "organisations.id",
        "alias": "consumers"
```

```
    }
]
```

Note that with aliases, in the mappings you refer to the table by alias specified in the joins, not by real table name. So after aliasing tables with previous join specification, the mapping should look like:

```
"mappings": {
    "supplier.name": "suppliers.org_name",
    "consumer.name": "consumers.org_name"
}
```

For example, we have a fact table named `fact_contracts` and dimension table with categories named `dm_categories`. To join them we define following join specification:

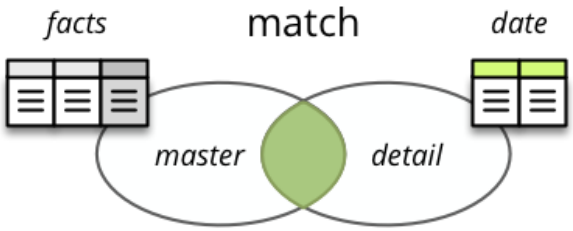
```
"joins" = [
    {
        "master": "fact_contracts.category_id",
        "detail": "dm_categories.id"
     }
]
```

Join Methods and Outer Joins

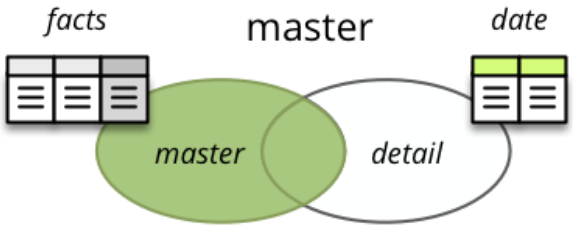
(advanced topic)

Cubes supports three join methods `match`, `detail` and `master`.

match (default) – the keys from both master and detail tables have to match – INNER JOIN



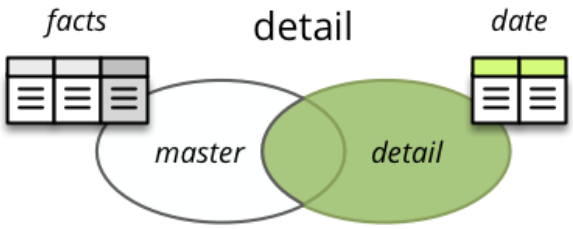
master – the master might contain more keys than the detail, for example the fact table (as a master) might contain unknown or new dimension entries not in the dimension table yet. This is also known as LEFT OUTER JOIN.



detail – every member of the detail table will be always present. For example every date from a date dimension table. Alskoknown as RIGHT OUTER JOIN.

To join a date dimension table so that every date will be present in the output reports, regardless whether there are any facts or not for given date dimension member:

```
"joins" = [
    {
        "master": "fact_contracts.contract_date_id",
        "detail": "dim_date.id",
```




```
        "method": "detail"
    }
]
```

The *detail* Method and its Limitations

(advanced topic)

When at least one table is joined using the outer *detail* method during aggregation, the statement is composed from two nested statements or two join zones: *master fact* and *outer detail*.

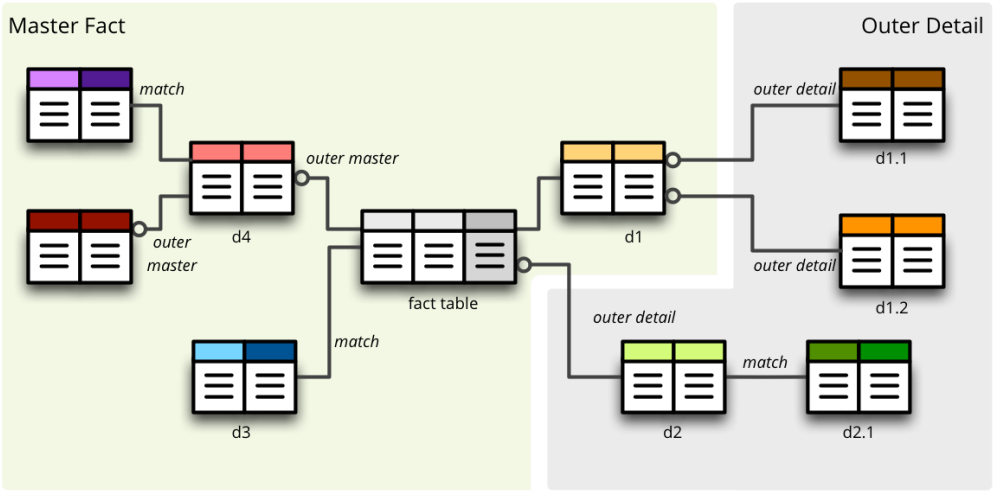


Fig. 4.4: Aggregate statement composition

The query builder analyses the schema and assigns a relationship of a table towards the fact. If a table is joined as *detail* or is behind a *detail* join it is considered to have a *detail* relationship towards the fact. Otherwise it has *master/match* relationship.

When this composed setting is used, then:

- aggregate functions are wrapped using `COALESCE()` to always return non-NULL values
- `count` aggregates are changed to count non-empty facts instead of all rows

---

Note: There should be no cut (path) that has some attributes in tables joined as *master* and others in a table joined as *detail*. Every cut (all the cut's attributes) should fall into one of the two table zones: either the master or the outer detail. There might be cuts from different join zones, though.

Take this into account when designing the dimension hierarchies.

Named Join Templates

If multiple cubes share the same kinds of joins, for example with a dimension table, it is possible to define such joins at the model level. They will be considered as templates:

```
"joins": [
    { "name": "date", "detail": "dim_date.id" },
    { "name": "company", "detail": "dim_company.id" }
]
```

Then use the join in a cube:

```
"cubes": [
    {
        "name": "events",
        "joins": [
            { "name": "date", "master": "event_date_id" },
            { "name": "company", "master": "company_id" }
        ]
    }
]
```

Any property defined in the cube join will replace the model join template. You can also use the same named join multiple times in a cube, just give it different alias:

```
"cubes": [
    {
        "name": "contracts",
        "joins": [
            {
                "name": "date",
                "master": "contract_start_date_id",
                "alias": "dim_contract_start"
            },
            {
                "name": "date",
                "master": "contract_end_date_id",
                "alias": "dim_contract_end"
            }
        ]
    }
]
```

Slicer Server

It is possible to plug-in cubes from other slicer servers using the Slicer Server backend.

Note: If the server has a JSON record limit set, then the backend will receive only limited number of facts.

Store Configuration and Model

Type is `slicer`

- `url` – Slicer URL
- `authentication` – authentication method of the source server (supported only `none` and `pass_parameter`)

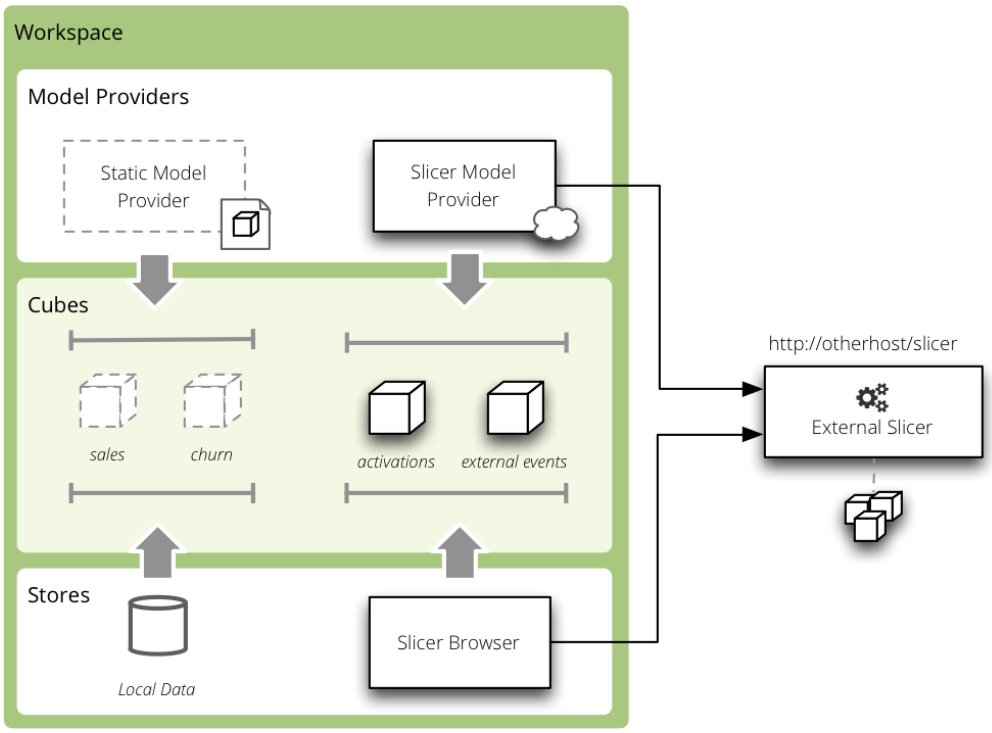


Fig. 4.5: Slicer backend

- `auth_identity` – authentication identity (or API key) for `pass_parameter` authentication.

Example:

```
[store]
type: slicer
url: http://slicer.databrewery.org/webshop-example
```

For more than one slicer define one datastore per source Slicer server.

Model

Slicer backend generates the model on-the-fly from the source server. You have to specify that the provider is `slicer`:

```
{
    "provider": "slicer"
}
```

For more than one slicer, create one file per source Slicer server and specify the data store:

```
{
    "provider": "slicer",
    "store": "slicer_2"
}
```

Example

Create a `model.json`:

```
{
    "provider": "slicer"
}
```

Create a `slicer.ini`:

```
[workspace]
model: slicer_model.json

[store]
type: slicer
url: http://slicer.databrewery.org/webshop-example

[server]
prettyprint: true
```

Run the server:

```
slicer serve slicer.ini
```

Get a list of cubes:

```
curl "http://localhost:5000/cubes"
```

CHAPTER 5

Slicer Server and Tool

OLAP Server

Cubes framework provides easy to install web service WSGI server with API that covers most of the Cubes logical model metadata and aggregation browsing functionality.

See also:

Configuration, *Server Deployment*

Server Requests

Version

Request: `GET /version`

Return a server version.

```
{
    "version": "1.0"
}
```

Info

Request: `GET /info`

Return an information about the server and server's data.

Content related keys:

- `label` – server's name or label
- `description` – description of the served data
- `copyright` – copyright of the data, if any
- `license` – data license
- `maintainer` – name of the data maintainer, might be in format `Name Surname <namesurname@domain.org>`

- `contributors` - list of contributors
- `keywords` – list of keywords that describe the data
- `related` – list of related or "friendly" Slicer servers with other open data – a dictionary with keys `label` and `url`.
- `visualizers` – list of links to prepared visualisations of the server's data – a dictionary with keys `label` and `url`.

Server related keys:

- `authentication` – authentication method, might be `none`, `pass_parameter`, `http_basic_proxy` or other. See *Authorization and Authentication* for more information
- `json_record_limit` - maximum number of records yielded for JSON responses
- `cubes_version` – Cubes framework version

Example:

```
{
    "description": "Some Open Data",
    "license": "Public Domain",
    "keywords": ["budget", "financial"],
    "authentication": "none",
    "json_record_limit": 1000,
    "cubes_version": "0.11.2"
}
```

Model

List of Cubes

Request: `GET /cubes`

Get list of basic information about served cubes. The cube description dictionaries contain keys: *name*, *label*, *description* and *category*.

```
[
    {
        "name": "contracts",
        "label": "Contracts",
        "description": "...",
        "category": "..."
    }
]
```

Cube Model

Request: `GET /cube/<name>/model`

Get model of a cube *name*. Returned structure is a dictionary with keys:

- `name` – cube name – used as server-wide cube identifier
- `label` – human readable name of the cube – to be displayed to the users (localized)
- `description` – optional textual cube description (localized)
- `dimensions` – list of dimension description dictionaries (see below)
- **`aggregates` – list of measures aggregates (mostly computed values) that** can be computed. Each item is a dictionary.

- **`measures` – list of measure attributes (properties of facts). Each** item is a dictionary. Example of a measure is: *amount*, *price*.
- `details` – list of attributes that contain fact details. Those attributes are provided only when getting a fact or a list of facts.
- **`info` – a dictionary with additional metadata that can be used in the** front-end. The contents of this dictionary is defined by the model creator and interpretation of values is left to the consumer.
- `features` (advanced) – a dictionary with features of the browser, such as available actions for the cube ("is fact listing possible?")

Aggregate is the key numerical property of the cube from reporting perspective. It is described as a dictionary with keys:

- `name` – aggregate identifier, such as: *amount_sum*, *price_avg*, *total*, *record_count*
- `label` – human readable label to be displayed (localized)
- `measure` – measure the aggregate is derived from, if it exists or it is known. Might be empty.
- `function` - name of an aggregate function applied to the *measure*, if known. For example: *sum*, *min*, *max*.
- `window_size` – number of elements within a window for window functions such as moving average
- `info` – additional custom information (unspecified)

Aggregate names are used in the `aggregates` parameter of the `/aggregate` request.

Measure dictionary contains:

- `name` – measure identifier
- `label` – human readable name to be displayed (localized)
- `aggregates` – list of aggregate functions that are provided for this measure
- `window_size` – number of elements within a window for window functions such as moving average
- `info` – additional custom information (unspecified)

Note: Compared to previous versions of Cubes, the clients do not have to construct aggregate names (as it used to be `amount``+``_sum`). Clients just get the aggrergate `name` property and use it right away.

See more information about measures and aggregates in the `/aggregate` request description.

Example cube:

```
{
    "name": "contracts",
    "info": {},
    "label": "Contracts",
    "aggregates": [
        {
            "name": "amount_sum",
            "label": "Amount sum",
            "info": {},
            "function": "sum"
        },
        {
            "name": "record_count",
            "label": "Record count",
            "info": {},
            "function": "count"
        }
    ],

    "measures": [
```

```
        {
            "name": "amount",
            "label": "Amount",
            "info": {},
            "aggregates": [ "sum" ]
        }
    ],

    "details": [...],

    "dimensions": [...]
}
```

The dimension description dictionary:

- `name` – dimension identifier
- `label` – human readable dimension name (localized)
- `is_flat` – *True* if the dimension has only one level, otherwise *False*
- `has_details` – *True* if the dimension has more than one attribute
- `default_hierarchy_name` - name of default dimension hierarchy
- `levels` – list of level descriptions
- `hierarchies` – list of dimension hierarchies
- `info` – additional custom information (unspecified)
- `cardinality` – dimension cardinality
- `role` – dimension role (special treatment, for example `time`)
- `category` – dimension category

The level description:

- `name` – level identifier (within dimension context)
- `label` – human readable level name (localized)
- `attributes` – list of level's attributes
- `key` – name of level's key attribute (mostly the first attribute)
- `label_attribute` – name of an attribute that contains label for the level's members (mostly the second attribute, if present)
- `order_attribute` – name of an attribute that the level should be ordered by (optional)
- `order` – order direction `asc`, `desc` or none.
- `cardinality` – symbolic approximation of the number of level's members
- `role` – level role (special treatment)
- `info` – additional custom information (unspecified)

Cardinality values and their meaning:

- `tiny` – few values, each value can have it's representation on the screen, recommended: up to 5.
- `low` – can be used in a list UI element, recommended 5 to 50 (if sorted)
- `medium` – UI element is a search/text field, recommended for more than 50 elements
- `high` – backends might refuse to yield results without explicit pagination or cut through this level.

Note: Use `attribute["ref"]` to access aggreegation result records. Each level (dimension) attribute description contains two properties: *name* and *ref*. *name* is identifier within the dimension context. The key reference *ref* is used for retrieving aggregation or browing results.

It is not recommended to create any dependency by parsing or constructing the *ref* property at the client's side.

Aggregation and Browsing

The core data and analytical functionality is accessed through the following requests:

- `/cube/<name>/aggregate` – aggregate measures, provide summary, generate drill-down, slice&dice, ...
- `/cube/<name>/members/<dim>` – list dimension members
- `/cube/<name>/facts` – list facts within a cell
- `/cube/<name>/fact` – return a single fact
- `/cube/<name>/cell` – describe the cell

If the model contains only one cube or default cube name is specified in the configuration, then the `/cube/<name>` part might be omitted and you can write only requests like `/aggregate`.

Cells and Cuts

The cell - part of the cube we are aggregating or we are interested in - is specified by cuts. The cut in URL are given as single parameter `cut` which has following format:

Examples:

```
date:2004
date:2004,1
date:2004,1|class:5
date:2004,1,1|category:5,10,12|class:5
```

To specify a range where keys are sortable:

```
date:2004-2005
date:2004,1-2005,5
```

Open range:

```
date:2004,1,1-
date:-2005,5,10
```

Set cuts:

```
date:2005;2007
```

Dimension name is followed by colon `:`, each dimension cut is separated by `|`, and path for dimension levels is separated by a comma `,`. Set cuts are separated by semicolons `;`.

To specify other than default hierarchy use format *dimension@hierarchy*, the path then should contain values for specified hierarchy levels:

```
date@ywd:2004,25
```

Following image contains examples of cuts in URLs and how they change by browsing cube aggregates:

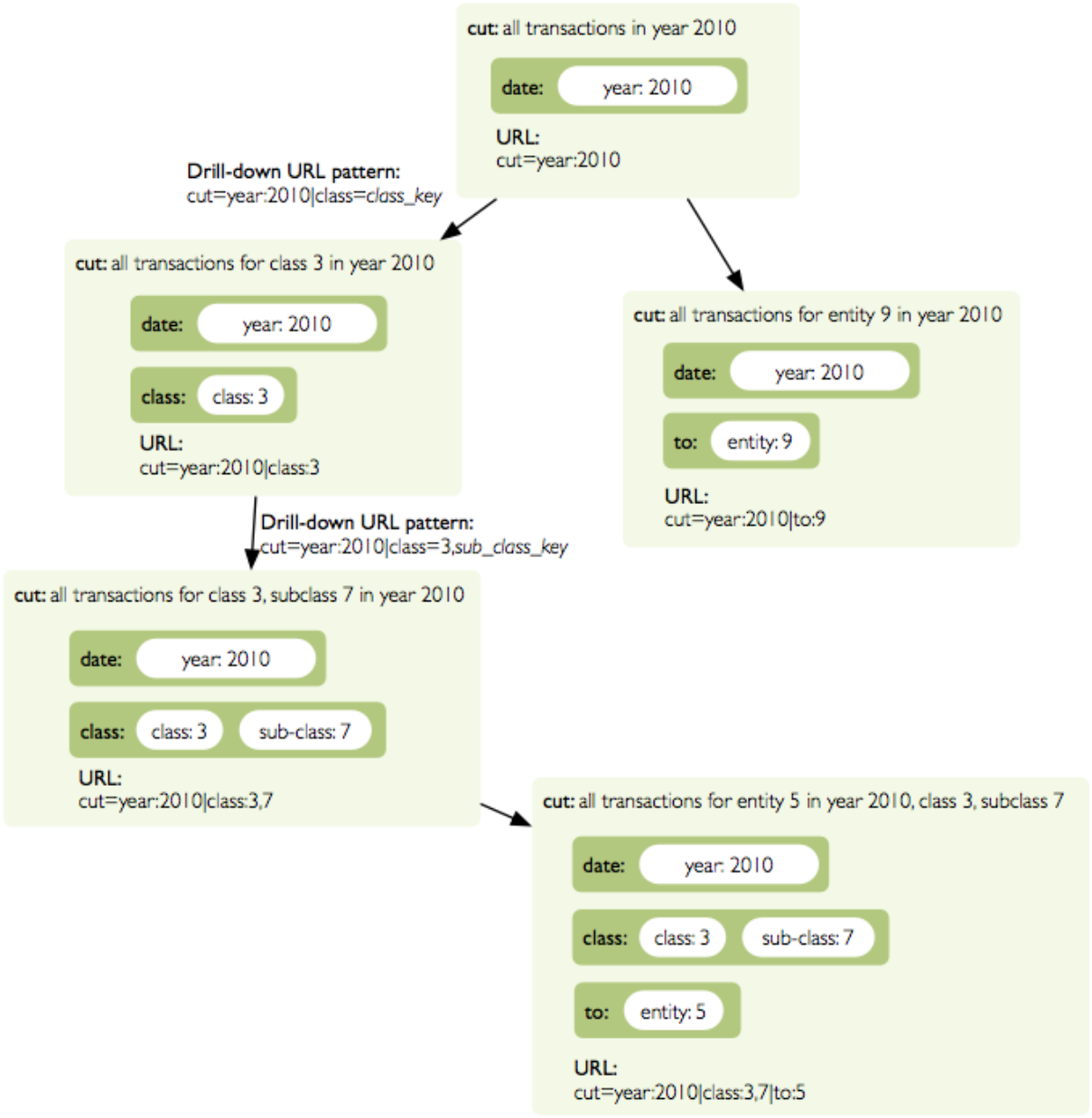


Fig. 5.1: Example of how cuts in URL work and how they should be used in application view templates.

Special Characters

To pass reserved characters as a dimension member path value escape it with the backslash `\` character:

- `category:10\-24` is a point cut for *category* with value `10-24`, not a range cut
- `city:Nové\ Mesto\ nad\ Váhom` is a city `Nové Mesto nad Váhom`

Calendar and Relative Time

If a dimension is a date or time dimension (the dimension role is `time`) the members can be specified by a name referring to a relative time. For example:

- `date:yesterday`
- `date:90daysago-today` – get cell for last 90 days
- `expiration_date:lastmonth-next2months` – all facts with *expiration date* within last month (whole) and next 2 months (whole)
- `date:yearago` – all facts since the same day of the year last year

The keywords and patterns are:

- `today`, `yesterday` and `tomorrow`
- `...ago` and `...forward` as in `3weeksago` (current day minus 3 weeks) and `2monthsforward` (current day plus 2 months) – relative offset with fine granularity
- `last...` and `next...` as in `last3months` (beginning of the third month before current month) and `nextyear` (end of next year) – relative offset of specific (more coarse) granularity.

Aggregate

Request: `GET /cube/<cube>/aggregate`

Return aggregation result as JSON. The result will contain keys: *summary* and *drilldown*. The summary contains one row and represents aggregation of whole cell specified in the cut. The *drilldown* contains rows for each value of drilled-down dimension.

If no arguments are given, then whole cube is aggregated.

Parameters:

- *cut* - specification of cell, for example: `cut=date:2004,1|category:2|entity:12345`
- *drilldown* - dimension to be drilled down. For example `drilldown=date` will give rows for each value of next level of dimension date. You can explicitly specify level to drill down in form: `dimension:level`, such as: `drilldown=date:month`. To specify a hierarchy use `dimension@hierarchy` as in `drilldown=date@ywd` for implicit level or `drilldown=date@ywd:week` to explicitly specify level.
- *aggregates* – list of aggregates to be computed, separated by `|`, for example: `aggregates=amount_sum|discount_avg|count`
- *measures* – list of measures for which their respecive aggregates will be computed (see below). Separated by `|`, for example: `aggregates=proce|discount`
- *page* - page number for paginated results
- *pagesize* - size of a page for paginated results
- *order* - list of attributes to be ordered by
- *split* – split cell, same syntax as the *cut*, defines virtual binary (flag) dimension that inticates whether a cell belongs to the *split* cut (*true*) or not (*false*). The dimension attribute is called *__within_split__*. Consult the backend you are using for more information, whether this feature is supported or not.

Note: You can specify either *aggregates* or *measures*. *aggregates* is a concrete list of computed values. *measures* yields their respective aggregates. For example: `measures=amount` might yield `amount_sum` and `amount_avg` if defined in the model.

Use of *aggregates* is preferred, as it is more explicit and the result is well defined.

Response:

A dictionary with keys:

- `summary` - dictionary of fields/values for summary aggregation
- `cells` - list of drilled-down cells with aggregated results
- `total_cell_count` - number of total cells in drilldown (after *limit*, before pagination). This value might not be present if it is disabled for computation on the server side.
- `aggregates` – list of aggregate names that were considered in the aggragation query
- `cell` - list of dictionaries describing the cell cuts
- `levels` – a dictionary where keys are dimension names and values is a list of levels the dimension was drilled-down to

Example for request `/aggregate?drilldown=date&cut=item:a`:

```
{
    "summary": {
        "count": 32,
        "amount_sum": 558430
    }
    "cells": [
        {
            "count": 16,
            "amount_sum": 275420,
            "date.year": 2009
        },
        {
            "count": 16,
            "amount_sum": 283010,
            "date.year": 2010
        }
    ],
    "aggregates": [
        "amount_sum",
        "count"
    ],
    "total_cell_count": 2,
    "cell": [
        {
            "path": [ "a" ],
            "type": "point",
            "dimension": "item",
            "invert": false,
            "level_depth": 1
        }
    ],
    "levels": { "date": [ "year" ] }
}
```

If pagination is used, then `drilldown` will not contain more than `pagesize` cells.

Note that not all backengs might implement `total_cell_count` or providing this information can be configurable therefore might be disabled (for example for performance reasons).

Facts

Request: `GET /cube/<cube>/facts`

Return all facts within a cell.

Parameters:

- *cut* - see `/aggregate`
- *page*, *pagesize* - paginate results
- *order* - order results
- *format* - result format: `json` (default; see note below), `csv` or `json_lines`.
- *fields* - comma separated list of fact fields, by default all fields are returned
- *header* – specify what kind of headers should be present in the `csv` output: `names` – raw field names (default), `labels` – human readable labels or `none`

The JSON response is a list of dictionaries where keys are attribute references (*ref* property of an attribute).

To use JSON formatted repsonse but don't have the record limit `json_lines` format can be used. The result is one fact record in JSON format per line – JSON dictionaries separated by newline *n* character.

Note: Number of facts in JSON is limited to configuration value of `json_record_limit`, which is 1000 by default. To get more records, either use pages with size less than record limit or use alternate result format, such as `csv`.

Single Fact

Request: `GET /cube/<cube>/fact/<id>`

Get single fact with specified *id*. For example: `/fact/1024`.

The response is a dictionary where keys are attribute references (*ref* property of an attribute).

Dimension members

Request: `GET /cube/<cube>/members/<dimension>`

Get *dimension* members used in *cube*.

Parameters:

- *cut* - see `/aggregate`
- ***depth* - specify depth (number of levels) to retrieve. If not** specified, then all levels are returned. Use this or *level*.
- *level* - deepest level to be retrieved – use this or *depth*.
- ***hierarchy* – name of hierarchy to be considered, if not specified, then** dimension's default hierarchy is used
- *page*, *pagesize* - paginate results
- *order* - order results

Response: dictionary with keys `dimension` – dimension name, `depth` – level depth and `data` – list of records.

Example for `/cube/facts/members/item?depth=1`:

```
{
    "dimension": "item"
    "depth": 1,
    "hierarchy": "default",
    "data": [
        {
            "item.category": "a",
            "item.category_label": "Assets"
        },
        {
            "item.category": "e",
            "item.category_label": "Equity"
        },
        {
            "item.category": "l",
            "item.category_label": "Liabilities"
        }
    ],
}
```

Cell

Get details for a cell.

Request: `GET /cube/<cube>/cell`

Parameters:

- *cut* - see `/aggregate`

Response: a dictionary representation of a `cell` (see `cubes.Cell.as_dict()`) with keys `cube` and `cuts`. *cube* is cube name and `cuts` is a list of dictionary representations of cuts.

Each cut is represented as:

```
{
    // Cut type is one of: "point", "range" or "set"
    "type": cut_type,

    "dimension": cut_dimension_name,
    "level_depth": maximal_depth_of_the_cut,

    // Cut type specific keys.

    // Point cut:
    "path": [ ... ],
    "details": [ ... ]

    // Range cut:
    "from": [ ... ],
    "to": [ ... ],
    "details": { "from": [...], "to": [...] }

    // Set cut:
    "paths": [ [...], [...], ... ],
    "details": [ [...], [...], ... ]
}
```

Each element of the `details` path contains dimension attributes for the corresponding level. In addition in contains two more keys: `_key` and `_label` which (redundantly) contain values of key attribute and label attribute respectively.

Example for `/cell?cut=item:a` in the `hello_world` example:

```
{
    "cube": "irbd_balance",
    "cuts": [
        {
            "type": "point",
            "dimension": "item",
            "level_depth": 1
            "path": ["a"],
            "details": [
                {
                    "item.category": "a",
                    "item.category_label": "Assets",
                    "_key": "a",
                    "_label": "Assets"
                }
            ],
        }
    ]
}
```

Report

Request: `GET /cube/<cube>/report`

Process multiple request within one API call. The data should be a JSON containing report specification where keys are names of queries and values are dictionaries describing the queries.

`report` expects `Content-type` header to be set to `application/json`.

See *Report* for more information.

Search

Warning: Experimental feature.

Note: Requires a search backend to be installed.

Request: `GET /cube/<cube>/search/dimension/<dimension>/<query>`

Search values of *dimensions* for *query*. If *dimension* is `_all` then all dimensions are searched. Returns search results as list of dictionaries with attributes:

Search result

- *dimension* - dimension name
- *level* - level name
- *depth* - level depth
- *level_key* - value of key attribute for level
- *attribute* - dimension attribute name where searched value was found
- *value* - value of dimension attribute that matches search query
- *path* - dimension hierarchy path to the found value
- ***level_label* - label for dimension level (value of label_attribute** for level)

Parameters that can be used in any request:

- *prettyprint* - if set to `true`, space indentation is added to the JSON output

Reports

Report queries are done either by specifying a report name in the request URL or using HTTP `GET` request where posted data are JSON with report specification.

Keys:

- *queries* - dictionary of named queries

Query specification should contain at least one key: *query* - which is query type: `aggregate`, `cell_details`, `values` (for dimension values), `facts` or `fact` (for multiple or single fact respectively). The rest of keys are query dependent. For more information see AggregationBrowser documentation.

Note: Note that you have to set the content type to `application/json`.

Result is a dictionary where keys are the query names specified in report specification and values are result values from each query call.

Example report JSON file with two queries:

```
{
    "queries": {
        "summary": {
            "query": "aggregate"
        },
        "by_year": {
            "query": "aggregate",
            "drilldown": ["date"],
            "rollup": "date"
        }
    }
}
```

Request:

```
curl -H "Content-Type: application/json" --data-binary "@report.json" \
    "http://localhost:5000/cube/contracts/report?prettyprint=true&cut=date:2004"
```

Reply:

```
{
    "by_year": {
        "total_cell_count": 6,
        "drilldown": [
            {
                "record_count": 4390,
                "requested_amount_sum": 2394804837.56,
                "received_amount_sum": 399136450.0,
                "date.year": "2004"
            },
            ...
            {
                "record_count": 265,
                "requested_amount_sum": 17963333.75,
                "received_amount_sum": 6901530.0,
                "date.year": "2010"
            }
        ],
        "summary": {
```

```
            "record_count": 33038,
            "requested_amount_sum": 2412768171.31,
            "received_amount_sum": 2166280591.0
        }
    },
    "summary": {
        "total_cell_count": null,
        "drilldown": {},
        "summary": {
            "date.year": "2004",
            "requested_amount_sum": 2394804837.56,
            "received_amount_sum": 399136450.0,
            "record_count": 4390
        }
    }
}
```

Explicit specification of a cell (the cuts in the URL parameters are going to be ignored):

```
{
    "cell": [
        {
            "dimension": "date",
            "type": "range",
            "from": [2010,9],
            "to": [2011,9]
        }
    ],
    "queries": {
        "report": {
            "query": "aggregate",
            "drilldown": {"date":"year"}
        }
    }
}
```

Roll-up

Report queries might contain `rollup` specification which will result in "rolling-up" one or more dimensions to desired level. This functionality is provided for cases when you would like to report at higher level of aggregation than the cell you provided is in. It works in similar way as drill down in `/aggregate` but in the opposite direction (it is like `cd ..` in a UNIX shell).

Example: You are reporting for year 2010, but you want to have a bar chart with all years. You specify rollup:

```
...
"rollup": "date",
...
```

Roll-up can be:

- a string - single dimension to be rolled up one level
- an array - list of dimension names to be rolled-up one level
- a dictionary where keys are dimension names and values are levels to be rolled up-to

Local Server

To run your local server, prepare server *Configuration* and run the server using the Slicer tool (see *slicer - Command Line Tool*):

```
slicer serve slicer.ini
```

Server requests

Example server request to get aggregate for whole cube:

```
$ curl http://localhost:5000/cube/procurements/aggregate?cut=date:2004
```

Reply:

```
{
    "drilldown": {},
    "summary": {
        "received_amount_sum": 399136450.0,
        "requested_amount_sum": 2394804837.56,
        "record_count": 4390
    }
}
```

Server Deployment

Apache mod_wsgi deployment

Deploying Cubes OLAP Web service server (for analytical API) can be done in four very simple steps:

1. Create slicer server *Configuration* file
2. Create WSGI script
3. Prepare apache site configuration
4. Reload apache configuration

Note: The model paths have to be full paths to the model, not relative paths to the configuration file.

Place the file in the same directory as the following WSGI script (for convenience).

Create a WSGI script `/var/www/wsgi/olap/procurements.wsgi`:

```
import os.path
from cubes.server import create_server

CURRENT_DIR = os.path.dirname(os.path.abspath(__file__))

# Set the configuration file name (and possibly whole path) here
CONFIG_PATH = os.path.join(CURRENT_DIR, "slicer.ini")

application = create_server(CONFIG_PATH)
```

Apache site configuration (for example in `/etc/apache2/sites-enabled/`):

```
<VirtualHost *:80>
    ServerName olap.democracyfarm.org

    WSGIScriptAlias /vvo /var/www/wsgi/olap/procurements.wsgi

    <Directory /var/www/wsgi/olap>
        WSGIProcessGroup olap
```

```
        WSGIApplicationGroup %{GLOBAL}
        Order deny,allow
        Allow from all
    </Directory>

    ErrorLog /var/log/apache2/olap.democracyfarm.org.error.log
    CustomLog /var/log/apache2/olap.democracyfarm.org.log combined

</VirtualHost>
```

Reload apache configuration:

```
sudo /etc/init.d/apache2 reload
```

UWSGI

Configuration file `uwsgi.ini`:

```
[uwsgi]
http = 127.0.0.1:5000
module = cubes.server.app
callable = application
```

Run `uwsgi uwsgi.ini`.

You can set environment variables:

- `SLICER_CONFIG` – full path to the slicer configuration file
- `SLICER_DEBUG` – set to true boolean value if you want to enable Flask server debugging

Heroku and UWSGI

To deploy the slicer in Heroku, prepare a directory with following files:

- `slicer.ini` – main slicer configuration file
- `uwsgi.ini` – UWSGI configuration
- `Procfile`

The `Procfile`:

```
web: uwsgi uwsgi.ini
```

The `uwsgi.ini`:

```
[uwsgi]
http-socket = :$(PORT)
master = true
processes = 4
die-on-term = true
memory-report = true
module = cubes.server.app
```

The `requirements.txt`:

```
Flask
SQLAlchemy
-e git+git://github.com/DataBrewery/cubes.git@master#egg=cubes
jsonschema
python-dateutil
```

```
expressions
grako
uwsgi
```

Add any packages that you might need for your Slicer server installation.

slicer - Command Line Tool

Cubes comes with a command line tool that can:

- run OLAP server
- build and compute cubes
- validate and translate models

Usage:

```
slicer command [command_options]
```

or:

```
slicer command sub_command [sub_command_options]
```

Commands are:

Command	Description
`list`	List available cubes in current workspace
`serve`	Start OLAP server
`model validate`	Validates logical model for OLAP cubes
`model convert`	Convert between model formats
`test`	Test the configuration and model against backends
`sql aggregate`	Create aggregated table
`sql denormalize`	Create denormalized table

serve

Run Cubes OLAP HTTP server.

Example server configuration file `slicer.ini`:

```
[server]
host: localhost
port: 5000
reload: yes
log_level: info

[workspace]
url: sqlite:///tutorial.sqlite
view_prefix: vft_

[model]
path: models/model_04.json
```

To run local server:

```
slicer serve slicer.ini
```

In the `[server]` section, space separated list of modules to be imported can be specified under option `modules`:

```
[server]
modules=cutom_backend
...
```

Note: Use *–debug* option if you would like to see more detailed error messages in the browser (generated by Flask).

For more information about OLAP HTTP server see *OLAP Server*

model convert

Usage:

```
slicer model convert --format bundle model.json model.cubesmodel
slicer model convert model.cubesmodel > model.json
```

Optional arguments:

```
--format              model format: json or bundle
--force               replace the target if exists
```

model validate

Usage:

```
slicer model validate /path/to/model/directory
slicer model validate model.json
slicer model validate http://somesite.com/model.json
```

Optional arguments:

```
-d, --defaults        show defaults
-w, --no-warnings     disable warnings
-t TRANSLATION, --translation TRANSLATION
                      model translation file
```

For more information see Model Validation in *Logical Model and Metadata*

Example output:

```
loading model wdmmg_model.json
--------------------------
cubes: 1
    wdmmg
dimensions: 5
    date
    pog
    region
    cofog
    from
--------------------------
found 3 issues
validation results:
warning: No hierarchies in dimension 'date', flat level 'year' will be used
warning: Level 'year' in dimension 'date' has no key attribute specified
warning: Level 'from' in dimension 'from' has no key attribute specified
0 errors, 3 warnings
```

The tool output contains recommendation whether the model can be used:

- *model can be used* - if there are no errors, no warnings and no defaults used, mostly when the model is explicitly described in every detail
- *model can be used, make sure that defaults reflect reality* - there are no errors, no warnings, but the model might be not complete and default assumptions are applied
- *not recommended to use the model, some issues might emerge* - there are just warnings, no validation errors. Some queries or any other operations might produce invalid or unexpected output
- *model can not be used* - model contain errors and it is unusable

test

Every cube in the model is tested through the backend whether it can be accessed and whether the mappings reflect reality.

Usage:

```
slicer test [-h] [-E EXCLUDE_STORES] [config] [cubes]
```

Positional arguments:

```
config                server configuration .ini file
cubes                 list of cubes to be tested
```

Optional arguments:

```
--aggregate               Test aggregate of whole cube
-E, --exclude-store TEXT
--store TEXT
--help                    Show this message and exit.
```

sql denormalize

Usage:

```
slicer sql denormalize [-h] [-f] [-m] [-i] [-s SCHEMA] [--config config]
                       [CUBE] [TARGET]
```

positional arguments:

```
CUBE                  cube to be denormalized
TARGET                target table name
```

optional arguments:

```
--force               replace existing views
-m, --materialize     create materialized view (table)
--index / --no-index  create index for key attributes
-s, --schema TEXT     target view schema (overrides default fact schema
--help                Show this message and exit.
--store TEXT   Name of the store to use other than default. Must be SQL.
--config TEXT  Name of slicer.ini configuration file
```

If no cube is specified then all cubes are denormalized according to the naming conventions in the configuration file.

Examples

If you plan to use denormalized views, you have to specify it in the configuration in the `[workspace]` section:

```
[workspace]
denormalized_view_prefix = mft_
denormalized_view_schema = denorm_views

# This switch is used by the browser:
use_denormalized = yes
```

The denormalization will create tables like `denorm_views.mft_contracts` for a cube named `contracts`. The browser will use the view if option `use_denormalization` is set to a true value.

Denormalize all cubes in the model:

```
slicer sql denormalize
```

Denormalize only one cube:

```
slicer sql denormalize contracts
```

Create materialized denormalized view with indexes:

```
slicer denormalize --materialize --index slicer.ini
```

Replace existing denormalized view of a cube:

```
slicer denormalize --force -c contracts slicer.ini
```

Schema

Schema where denormalized view is created is schema specified in the configuration file. Schema is shared with fact tables and views. If you want to have views in separate schema, specify `denormalized_schema` option in the configuration.

If for any specific reason you would like to denormalize into a completely different schema than specified in the configuration, you can specify it with the `--schema` option.

View name

By default, a view name is the same as corresponding cube name. If there is `denormalized_prefix` option in the configuration, then the prefix is prepended to the cube name. Or it is possible to override the option with command line argument `--prefix`.

Note: The tool will not allow to create view if it's name is the same as fact table name and is in the same schema. It is not even possible to `--force` it. A view prefix or different schema has to be specified.

---

sql aggregate

Create pre-aggregated table from cube(s). If no cube is specified, then all cubes are aggregated. Target table can be specified only for one cube, for multiple cubes naming convention is used.

Usage:

```
slicer sql aggregate [OPTIONS] [CUBE] [TARGET]
```

positional arguments:

```
CUBE                  cube to be denormalized
TARGET                target table name
```

optional arguments:

```
--force               replace existing views
--index / --no-index  create index for key attributes
-s, --schema TEXT     target view schema (overrides default fact schema
-d, --dimension TEXT  dimension to be used for aggregation
--help                Show this message and exit.
--store TEXT   Name of the store to use other than default. Must be SQL.
--config TEXT  Name of slicer.ini configuration file
```

If no cube is specified then all cubes are denormalized according to the naming conventions in the configuration file.

CHAPTER 6

Recipes

Recipes

How-to guides with code snippets for various use-cases.

Integration With Flask Application

Objective: Add Cubes Slicer to your application to provide raw analytical data.

Cubes Slicer Server can be integrated with your application very easily. The Slicer is provided as a flask Blueprint – a module that can be plugged-in.

The following code will add all Slicer's end-points to your application:

```
from flask import Flask
from cubes.server import slicer

app = Flask(__name__)
app.register_blueprint(slicer, config="slicer.ini")
```

To have a separate sub-path for Slicer add *url_prefix*:

```
app.register_blueprint(slicer, url_prefix="/slicer", config="slicer.ini")
```

See also:

Flask – Modular Applications with Blueprints

HTTP WSGI OLAP Server Reference

Publishing Open Data with Cubes

Cubes and Slicer were built with Open Data or rather Open Analytical Data in mind.

Read more about Open Data:

- Open Data (Wikipedia)
- Defining Open Data (OKFN)

- What is Open Data (Open Data Handbook)

With Cubes you can have a server that provides raw detailed data (denormalized facts) and grouped and aggregated data (aggregates). It is possible to serve multiple datasets which might share properties (dimensions).

Serving Open Data

Just create a public *Slicer server*. To provide more metadata add a `info.json` file with the following contents:

- `label` – server's name or label
- `description` – description of the served data
- `copyright` – copyright of the data, if any
- `license` – data license, such as Creative Commons, Public Domain or other
- `maintainer` – name of the data maintainer, might be in format `Name Surname <namesurname@domain.org>`
- `contributors` - list of contributors (if any)
- `keywords` – list of keywords that describe the data
- `related` – list of related or "friendly" Slicer servers with other open data
- `visualizations` – list of links to prepared visualisations of the server's data

Create a `info.json` file:

```
{
    "description": "Some Open Data",
    "license": "Public Domain",
    "keywords": ["budget", "financial"],
}
```

Include *info* option in the slicer configuration:

```
[workspace]
info: info.json
```

Related Servers

For better open data discoverability you might add links to other servers:

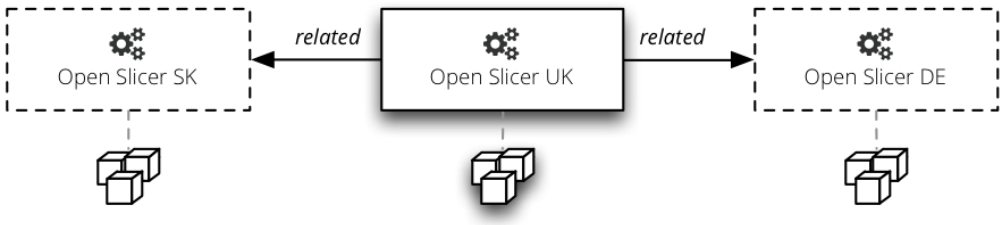


Fig. 6.1: Related slicers.

```
{
    "related": [
        {
            "label": "Slicer - Germany",
            "url": "http://slicer.somewhere.de",
        },
```

```
        {
            "label": "Slicer - Slovakia",
            "url": "http://slicer.somewhere.sk",
        }
    ]
}
```

Drill-down Tree

Goal: Create a tree by aggregating every level of a dimension.

Level: Advanced.

Drill-down

Drill-down is an action that will provide more details about data. Drilling down through a dimension hierarchy will expand next level of the dimension. It can be compared to browsing through your directory structure.

We create a function that will recursively traverse a dimension hierarchy and will print-out aggregations (count of records in this example) at the actual browsed location.

Attributes

- cell - cube cell to drill-down
- dimension - dimension to be traversed through all levels
- path - current path of the *dimension*

Path is list of dimension points (keys) at each level. It is like file-system path.

```
def drill_down(cell, dimension, path=[]):
```

Get dimension's default hierarchy. Cubes supports multiple hierarchies, for example for date you might have year-month-day or year-quarter-month-day. Most dimensions will have one hierarchy, thought.

```
hierarchy = dimension.hierarchy()
```

Base path is path to the most detailed element, to the leaf of a tree, to the fact. Can we go deeper in the hierarchy?

```
if hierarchy.path_is_base(path):
    return
```

Get the next level in the hierarchy. *levels_for_path* returns list of levels according to provided path. When *drilldown* is set to *True* then one more level is returned.

```
levels = hierarchy.levels_for_path(path,drilldown=True)
current_level = levels[-1]
```

We need to know name of the level key attribute which contains a path component. If the model does not explicitly specify key attribute for the level, then first attribute will be used:

```
level_key = dimension.attribute_reference(current_level.key)
```

For prettier display, we get name of attribute which contains label to be displayed for the current level. If there is no label attribute, then key attribute is used.

```
level_label = dimension.attribute_reference(current_level.label_attribute)
```

We do the aggregation of the cell...

Note: Shell analogy: Think of `ls $CELL` command in commandline, where `$CELL` is a directory name. In this function we can think of `$CELL` to be same as current working directory (`pwd`)

```
result = browser.aggregate(cell, drilldown=[dimension])

for record in result.drilldown:
    print "%s%s: %d" % (indent, record[level_label], record["record_count"])
    ...
```

And now the drill-down magic. First, construct new path by key attribute value appended to the current path:

```
drill_path = path[:] + [record[level_key]]
```

Then get a new cell slice for current path:

```
drill_down_cell = cell.slice(dimension, drill_path)
```

And do recursive drill-down:

```
drill_down(drill_down_cell, dimension, drill_path)
```

The whole recursive drill down function looks like this:

Whole working example can be found in the `tutorial` sources.

Get the full cube (or any part of the cube you like):

```
cell = browser.full_cube()
```

And do the drill-down through the item dimension:

```
drill_down(cell, cube.dimension("item"))
```

The output should look like this:

```
a: 32
    da: 8
        Borrowings: 2
        Client operations: 2
        Investments: 2
        Other: 2
    dfb: 4
        Currencies subject to restriction: 2
        Unrestricted currencies: 2
    i: 2
        Trading: 2
    lo: 2
        Net loans outstanding: 2
    nn: 2
        Nonnegotiable, nonintrest-bearing demand obligations on account of␣
↪subscribed capital: 2
    oa: 6
        Assets under retirement benefit plans: 2
        Miscellaneous: 2
        Premises and equipment (net): 2
```

Note that because we have changed our source data, we see level codes instead of level names. We will fix that later. Now focus on the drill-down.

See that nice hierarchy tree?

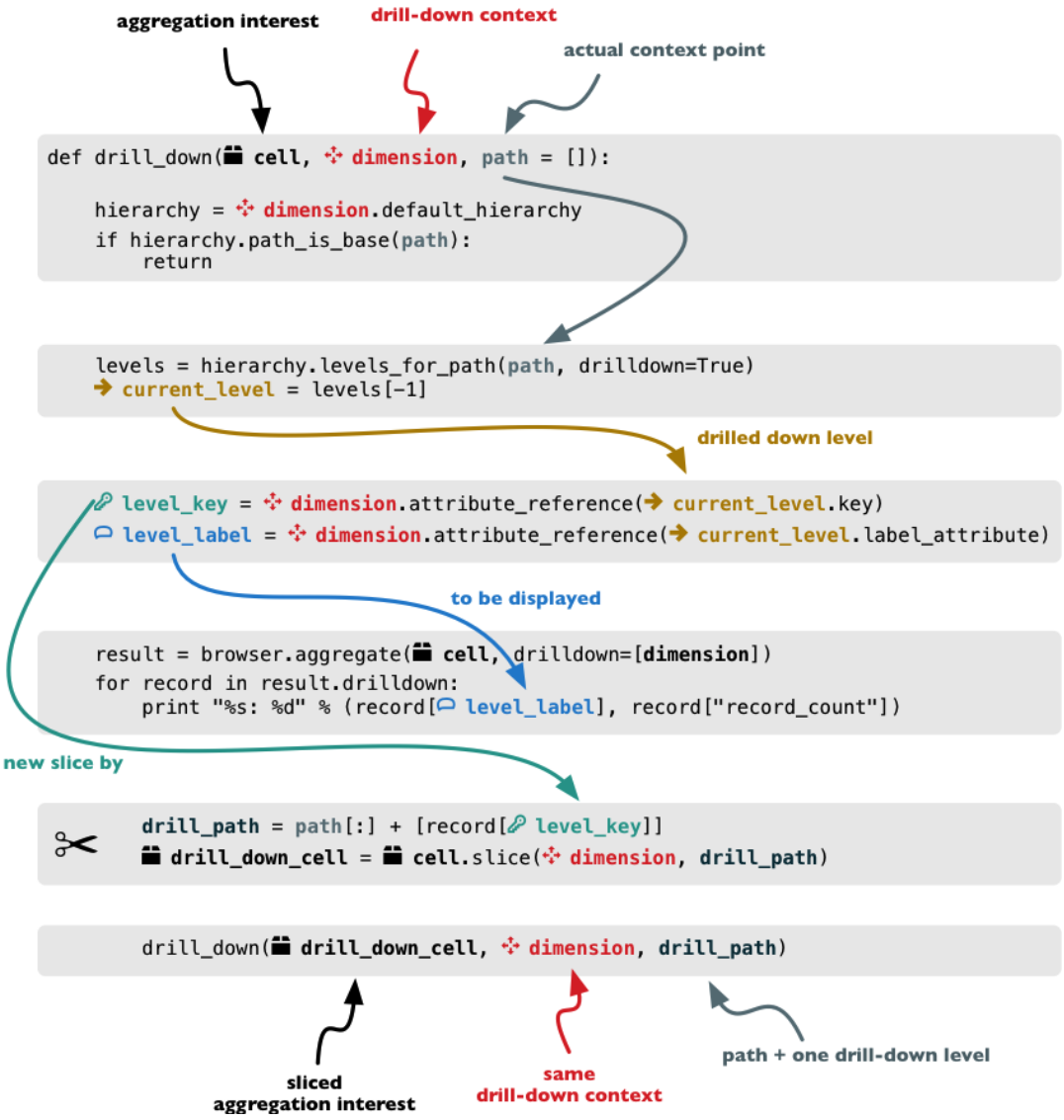


Fig. 6.2: Recursive drill-down explained

Now if you slice the cell through year 2010 and do the exact same drill-down:

```
cell = cell.slice("year", [2010])
drill_down(cell, cube.dimension("item"))
```

you will get similar tree, but only for year 2010 (obviously).

Level Labels and Details

Codes and ids are good for machines and programmers, they are short, might follow some scheme, easy to handle in scripts. Report users have no much use of them, as they look cryptic and have no meaning for the first sight.

Our source data contains two columns for category and for subcategory: column with code and column with label for user interfaces. Both columns belong to the same dimension and to the same level. The key column is used by the analytical system to refer to the dimension point and the label is just decoration.

Levels can have any number of detail attributes. The detail attributes have no analytical meaning and are just ignored during aggregations. If you want to do analysis based on an attribute, make it a separate dimension instead.

So now we fix our model by specifying detail attributes for the levels:

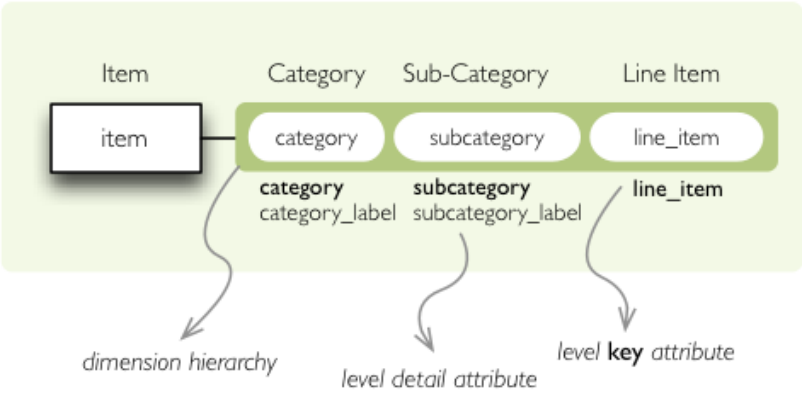


Fig. 6.3: Attribute details.

The model description is:

```
"levels": [
        {
            "name":"category",
            "label":"Category",
            "label_attribute": "category_label",
            "attributes": ["category", "category_label"]
        },
        {
            "name":"subcategory",
            "label":"Sub-category",
            "label_attribute": "subcategory_label",
            "attributes": ["subcategory", "subcategory_label"]
        },
        {
            "name":"line_item",
            "label":"Line Item",
            "attributes": ["line_item"]
        }
    ]
}
```

Note the *label_attribute* keys. They specify which attribute contains label to be displayed. Key attribute is by-default the first attribute in the list. If one wants to use some other attribute it can be specified in *key_attribute*.

Because we added two new attributes, we have to add mappings for them:

```
"mappings": { "item.line_item": "line_item",
              "item.subcategory": "subcategory",
              "item.subcategory_label": "subcategory_label",
              "item.category": "category",
              "item.category_label": "category_label"
             }
```

Now the result will be with labels instead of codes:

```
Assets: 32
    Derivative Assets: 8
        Borrowings: 2
        Client operations: 2
        Investments: 2
        Other: 2
    Due from Banks: 4
        Currencies subject to restriction: 2
        Unrestricted currencies: 2
    Investments: 2
        Trading: 2
    Loans Outstanding: 2
        Net loans outstanding: 2
    Nonnegotiable: 2
        Nonnegotiable, nonintrest-bearing demand obligations on account of␣
↪subscribed capital: 2
    Other Assets: 6
        Assets under retirement benefit plans: 2
        Miscellaneous: 2
        Premises and equipment (net): 2
```

Hierarchies, levels and drilling-down

Goals:

- how to create a hierarchical dimension
- how to do drill-down through a hierarchy
- detailed level description

Level: basic.

We are going to use very similar data as in the previous examples. Difference is in two added columns: category code and sub-category code. They are simple letter codes for the categories and subcategories. Download `this example file`.

Hierarchy

Some `dimensions` can have multiple `levels` forming a `hierarchy`. For example dates have year, month, day; geography has country, region, city; product might have category, subcategory and the product.

In our example we have the *item* dimension with three levels of hierarchy: *category*, *subcategory* and *line item*:

The levels are defined in the model:

```
"levels": [
    {
        "name":"category",
```

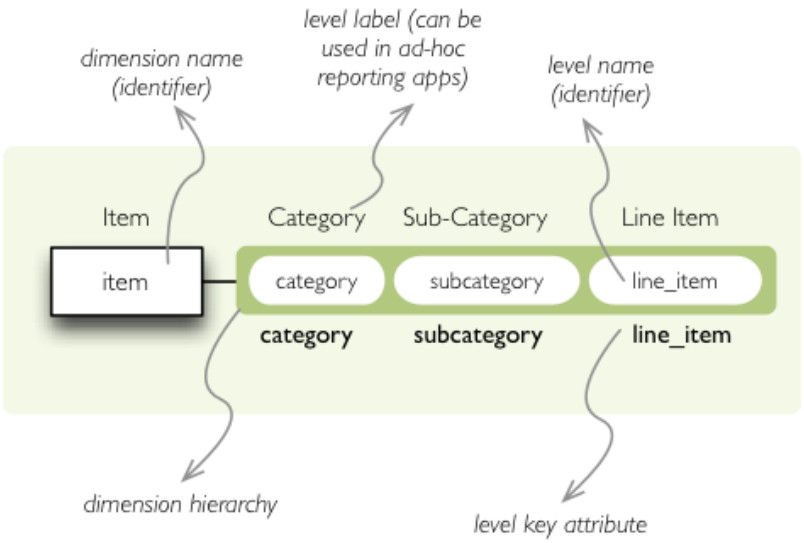


Fig. 6.4: *Item* dimension hierarchy.

```
        "label":"Category",
        "attributes": ["category"]
    },
    {
        "name":"subcategory",
        "label":"Sub-category",
        "attributes": ["subcategory"]
    },
    {
        "name":"line_item",
        "label":"Line Item",
        "attributes": ["line_item"]
    }
]
```

You can see a slight difference between this model description and the previous one: we didn't just specify level names and didn't let cubes to fill-in the defaults. Here we used explicit description of each level. *name* is level identifier, *label* is human-readable label of the level that can be used in end-user applications and *attributes* is list of attributes that belong to the level. The first attribute, if not specified otherwise, is the key attribute of the level.

Other level description attributes are *key* and *label_attribute*'. The *key* specifies attribute name which contains key for the level. Key is an id number, code or anything that uniquely identifies the dimension level. *label_attribute* is name of an attribute that contains human-readable value that can be displayed in user-interface elements such as tables or charts.

Preparation

Again, in short we need:

- data in a database
- logical model (see `model file`) prepared with appropriate mappings
- denormalized view for aggregated browsing (optional)

Implicit hierarchy

Try to remove the last level *line_item* from the model file and see what happens. Code still works, but displays only two levels. What does that mean? If metadata - logical model - is used properly in an application, then

application can handle most of the model changes without any application modifications. That is, if you add new level or remove a level, there is no need to change your reporting application.

Summary

- hierarchies can have multiple levels
- a hierarchy level is identifier by a key attribute
- a hierarchy level can have multiple detail attributes and there is one special detail attribute: label attribute used for display in user interfaces

Multiple Hierarchies

Dimension can have multiple hierarchies defined. To use specific hierarchy for drilling down:

```
result = browser.aggregate(cell, drilldown = [("date", "dmy", None)])
```

The *drilldown* argument takes list of three element tuples in form: (*dimension*, *hierarchy*, *level*). The *hierarchy* and *level* are optional. If *level* is `None`, as in our example, then next level is used. If *hierarchy* is `None` then default hierarchy is used.

To sepcify hierarchy in cell cuts just pass *hierarchy* argument during cut construction. For example to specify cut through week 15 in year 2010:

```
cut = cubes.PointCut("date", [2010, 15], hierarchy="ywd")
```

Note: If drilling down a hierarchy and asking cubes for next implicit level the cuts should be using same hierarchy as drilldown. Otherwise exception is raised. For example: if cutting through year-month-day and asking for next level after year in year-week-day hierarchy, exception is raised.

CHAPTER 7

Extension Development

Plugin Reference

Cubes has a plug-in based architecture. The objects that can be provided through external plug-ins are: *authenticators*, *authorizers*, *browsers*, *formatters*, *model_providers* and *stores*.

Plugins are classes providing an interface respective for the plug-in class. They are advertised throgh `setup.py` as follows:

```
setup(
    name = "my_package",

    # ... regular module setup here

    # Cubes Plugin Advertisment
    #
    entry_points={
        'cubes.stores': [
            'my = my_package.MyStore',
        ],
        'cubes.authorizers': [
            'my = my_package.MyAuthorizer',
        ]
    }
)
```

For more information see Python Packaging User Guide

Backends

Two objects play major role in Cubes backends:

- *aggregation browser* – responsible for aggregations, fact listing, dimension member listing
- *store* – represents a database connection, shared by multiple browsers

See also:

Plugin Reference

Store

Data for cubes are provided by a *data store* – every cube has one. Stores have to be subclasses of *Store* for cubes to be able to find them.

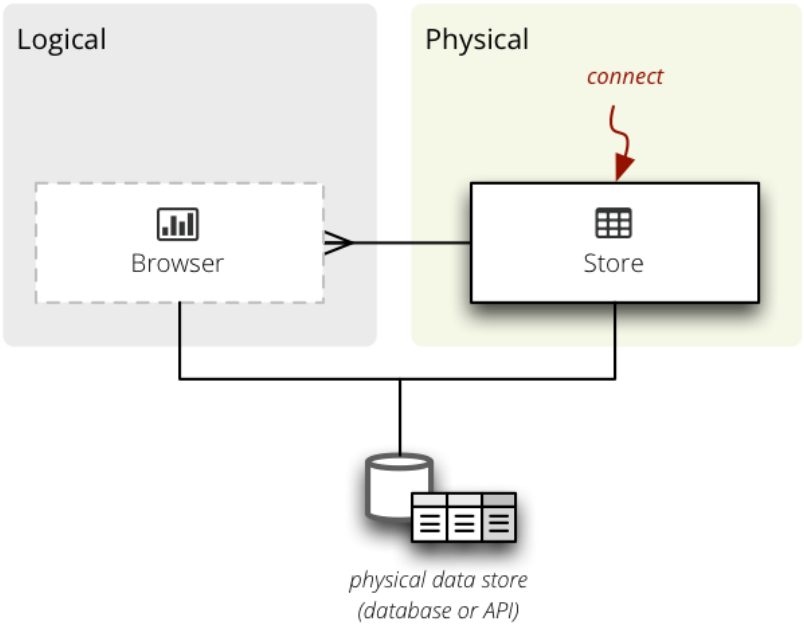


Fig. 7.1: Backend data store.

Required methods:

- *__init__(**options)* – initialize the store with *options*. Even if you use named arguments, you have to include the ***options*.
- *close()* – release all resources associated with the store, close database connections
- *default_browser_name* – a class variable with browser name that will be created for a cube, if not specified otherwise

A *Store* class:

```
from cubes import Store

class MyStore(Store):
    default_browser_name = "my"

    def __init__(self, **options):
        super(MyStore, self).__init__(**options)
        # configure the store here ...
```

Note: The custom store has to be a subclass of *Store* so Cubes can find it. The name will be derived from the class name: *MyStore* will become *my*, *AnotherSQLStore* will become *another_sql*. To explicitly specify a store name, set the *__extension_name__* class variable.

Configuration

The store is configured from a *slicer.ini* file. The store instance receives all options from it's configuration file section as arguments to the *__init__()* method.

It is highly recommended that the store provides a class variable named *__options__* which is a list of parameter description dictionaries. The list is used for properly configuring the store from end-user tools, such as Slicer. It also provides information about how to convert options into appropriate data types. Example:

```
class MyStore(Store):
    default_browser_name = "my"

    __options__ = [
        {
            "name": "collection",
            "type": "string",
            "description": "Name of data collection"
        },
        {
            "name": "unfold",
            "type": "bool",
            "description": "Unfold nested structures"
        }
    ]

    def __init__(self, collection=None, unfold=Flase, **options):
        super(MyStore, self).__init__(**options)

        self.collection = collection
        self.unfold = unfold
```

An example configuration for this store would look like:

```
[store]
type: my
collection: data
unfold: true
```

Aggregation Browser

Browser retrieves data from a *store* and works in a context of a *cube* and *locale*.

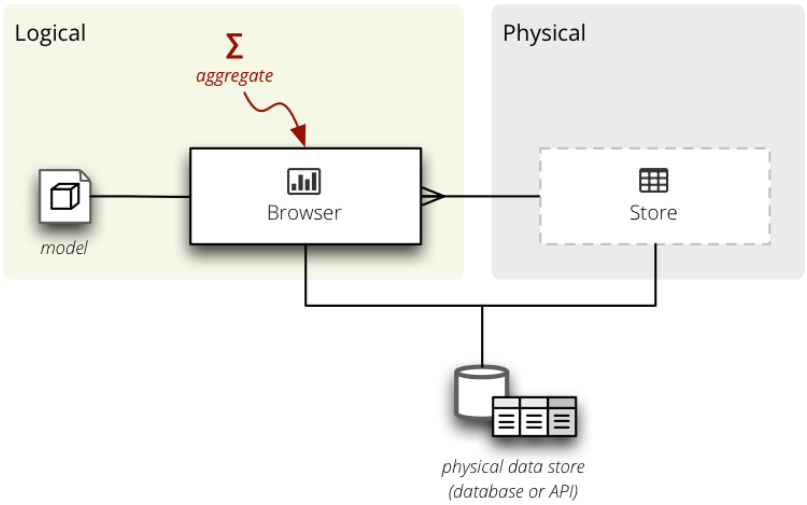


Fig. 7.2: Backend data store.

Methods to be implemented:

- *__init__(cube, store, locale)* – initialize the browser for *cube* stored in a *store* and use model and data *locale*.

- *features()* – return a dictionary with browser's features
- *aggregate()*, *facts()*, *fact()*, *members()* – all basic browser actions that take a cell as first argument. See `AggregationBrowser` for more information.

For example:

```
class SnowflakeBrowser(AggregationBrowser):

    def __init__(self, cube, store, locale=None, **options):
        super(SnowflakeBrowser, self).__init__(cube, store, locale)
        # browser initialization...
```

Name of the example store will be `snowflake`. To explicitly set the browser name set the *__extension_name__* class property:

```
class SnowflakeBrowser(AggregationBrowser):
    __extension_name__ = "sql"
```

In this case, the browser will be known by the name `sql`.

Note: The current *AggregationBrowser* API towards the extension development is provisional and will verylikely change. The change will mostly involve removal of requirements for preparation of arguments and return value.

Aggregate

Implement the *provide_aggregate()* method with the following arguments:

- *cell* – cube cell to be aggregated, alwas a `cubes.Cell` instance
- *aggregates* – list of aggregates to be considered
- *drilldown* – `cubes.Drilldown` instance (already prepared)
- *split* (optional browser feature) – virtual cell-based dimension to split the aggregation cell into two: within the split cell or outside of the split cell. Can be either *None* or a `cubes.Cell` instance
- *page*, *page_size* – page number and size of the page for paginated results
- *order* – order specification: list of two-item tuples (*attribute*, *order*)

```
def provide_aggregate(self, cell, aggregates, drilldown, split, order,
                      page, page_size, **options):

    #
    # ... do the aggregation here ...
    #

    result = AggregationResult(cell=cell, aggregates=aggregates)

    # Set the result cells iterator (required)
    result.cells = ...
    result.labels = ...

    # Optional:
    result.total_cell_count = ...
    result.summary = ...

    return result
```

Note: Don't override the *aggregate()* method – it takes care of proper argument conversions and set-up.

See also:

`cubes.AggregationResult`, `cubes.Drilldown`, `cubes.Cell`

Facts

```
def facts(self, cell=None, fields=None, order=None, page=None,
          page_size=None):

    cell = cell or Cell(self.cube)
    attributes = self.cube.get_attributes(fields)
    order = self.prepare_order(order, is_aggregate=False)

    #
    # ... fetch the facts here ...
    #
    # facts = ... an iterable ...
    #

    result = Facts(facts, attributes)

    return result
```

Browser and Cube Features

The browser features for all or a particuliar cube (if there are differences) are returned by the `cubes.AggregationBrowser.features()` method. The method is expected to return at least one key in the dictionary: `actions` with list of browser actions that the browser supports.

Browser actions are: `aggregate`, `fact`, `facts`, `members` and `cell`.

Optional but recommended is setting the list of `aggregate_functions` – functions for measures computed in the browser's engine. The other is `post_aggregate_functions` – list of fucntions used as post-aggregation outside of the browser.

Configuration

The browser is configured by merging:

- model's *options* property
- cube's *options* property
- store's configuration options (from `slicer.ini`)

The browser instance receives the options as parameters to the *__init__()* method.

Model Providers

Model providers create `cubes.Cube` and `cubes.Dimension` objects from a metadata or an external description.

To implement a custom model provider subclass the `cubes.ModelProvider` class. It is required that the *__init__* method calls the super's *__init__* with the *metadata* argument.

Required methods to be implemented:

- *list_cubes()* – return a list of cubes that the provider provides. Return value should be a dictionary with keys: `name`, `label`, `description` and `info`.

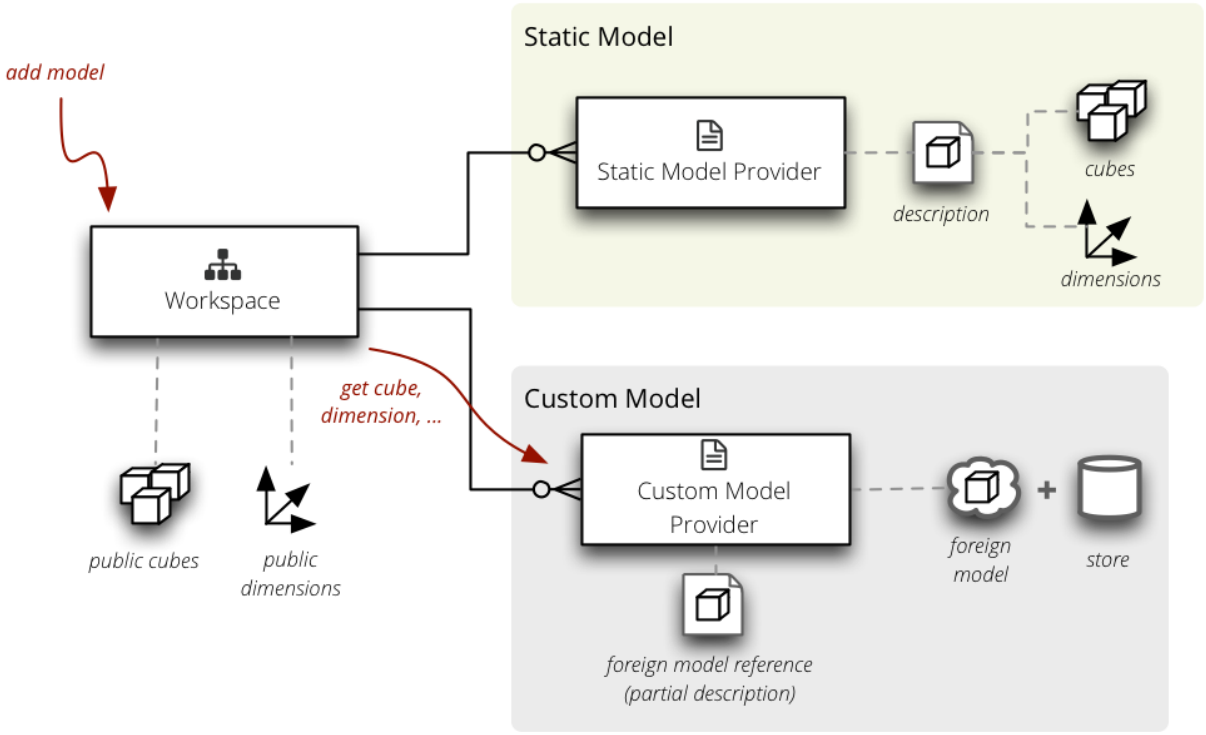


Fig. 7.3: Context of Model Providers.

- *cube(name)* – return a `cubes.Cube` object
- *dimension(name, dimensions)* – return a `cubes.Dimension` object. *dimensions* is a dictionary of public dimensions that can be used as templates. If a template is missing the method should raise *TemplateRequired(template)* error.

Optional:

- *requires_store()* – return *True* in this method if the provider requires a data store (database connection, API credentials, ...).

See also:

Model Reference, *Model Providers Reference*, `cubes.ModelProvider`, `cubes.StaticModelProvider`, `cubes.create_cube()`, `cubes.create_dimension()`

Cube

To provide a cube implement *cube(name)* method. The method should raise *NoSuchCubeError* when a cube is not provided by the provider.

To set cube's dimension you can either set dimension's name in *linked_dimensions* or directly a *Dimension* object in *dimensions*. The rule is:

- *linked_dimensions* – shared dimensions, might be defined in external model, might be even own dimension that is considered public
- *dimensions* – private dimensions, dimensions with public name conflicts

---

Note: It is recommended to use the *linked_dimensions* name list. The *dimensions* is considered an advanced feature.

Example of a provider which provides just a simple cube with date dimension and a measure *amount* and two aggregates *amount_sum* and *record_count*. Knows three cubes: *activations*, *churn* and *sales*:

```
from cubes import ModelProvider, create_cube

class SimpleModelProvider(ModelProvider):
    def __init__(self, metadata=None):
        super(DatabaseModelProvider, self).__init__(metadata)

        self.known_cubes = ["activations", "churn", "sales"]

    def list_cubes(self):

        cubes = []
        for name in self.known_cubes:
            info = {"name": name}
            cubes.append(info)

        return cubes

    def cube(self, name):
        if not name in self.known_cubes:
            raise NoSuchCubeError("Unknown cube '%s'" % name, name)

        metadata = {
            "name": name,
            "linked_dimensions": ["date"],
            "measures": ["amount"],
            "aggregats": [
                {"name": "amount_sum", "measure": "amount", "function": "sum"},
                {"name": "record_count", "function": "count"}
            ]
        }

        return create_cube(metadata)
```

The above provider assumes that some other object providers the *date* dimension.

Store

Some providers might require a database connection or an API credentials that might be shared by the data store containing the actual cube data. In this case the model provider should implement method *requires_store()* and return `True`. The provider's *initialize_from_store()* will be called back at some point before first cube is retrieved. The provider will have *store* instance variable available with `cubes.Store` object instance.

Example:

```
from cubes import ModelProvider, create_cube
from sqlalchemy import sql
import json

class DatabaseModelProvider(ModelProvider):
    def requires_store(self):
        return True

    def initialize_from_store(self):
        self.table = self.store.table("cubes_metadata")
        self.engine = self.store.engine

    def cube(self, name):
        self.engine.execute(select)

        # Let's assume that we have a SQLalchemy table with a JSON string
        # with cube metadata and columns: name, metadata
```

```
        condition = self.table.c.name == name

        statement = sql.expression.select(self.table.c.metadata,
                                          from_obj=self.table,
                                          where=condition)

        result = list(self.engine.execute(statement))

        if not result:
            raise NoSuchCubeError("Unknown cube '%s'" % name, name)

        cube = json.loads(result[0])

        return create_cube(cube)
```

See also:

Plugin Reference

Authenticators and Authorizers

See also:

Plugin Reference

Authorizer

Authorizers gives or denies access to cubes and restricts access to a portion of a cube.

Custom authorizers should be subclasses of `cubes.Authorizer` (to be findable) and should have the following methods:

- *authorize(identity, cubes)* – return list of cube names (from the *cubes*) that the *identity* is allowed to acces. Might return an empty list if no cubes are allowed.
- *restricted_cell(identity, cube, cell)* – return a cell derived from *cell* with restrictions for *identity*

Custom authorizer example: an authorizer that uses some HTTP service that accepts list of cubes in the `cubes=` paramter and returns a comma separated list of authorized cubes.

```
class CustomAuthorizer(Authorizer):
    def __init__(self, url=None, user_dimension=None, **options):
        super(DatabaseAuthorizer, self).__init__(self, **options)

        self.url = url
        self.user_dimension = user_dimension or "user"

    def authorize(self, cubes):
        params = {
            "cubes": ",".join(cubes)
        }

        response = Request(url, params=params)

        return response.data.split(",")
```

Note: The custom authorizer has to be a subclass of *Authorizer* so Cubes can find it. The name will be derived from the class name: *CustomAuthorizer* will become *custom*, *DatabaseACLAuthorizer* will become *database_acl*.

To explicitly specify an authorizer name, set the *__extension_name__* class variable.

The cell restrictions are handled by *restricted_cell()* method which receives the identity, cube object (not just a name) and optionaly the cell to be restricted.

```
class CustomAuthorizer(Authorizer):
    def __init__(self, url=None, table=None, **options):
        # ... initialization goes here ...

    def authorize(self, cubes):
        # ... authorization goes here
        return cubes

    def restricted_cell(self, identity, cube, cell):

        # If the cube has no dimension "user", we can't restrict
        # and we assume that the cube can be seen by anyone

        try:
            cube.dimension(self.user_dimension)
        except NoSuchDimensionError:
            return cell

        # Find the user ID based on identity
        user_id = self.find_user(identity)

        # Assume a flat "user" dimension for every cube
        cut = PointCut(self.user_dimension, [user_id])
        restriction = Cell(cube, [cut])

        if cell:
            return cell & restriction
        else:
            return restriction
```

Configuration

The authorizer is configured from the `[authorization]` section in the *slicer.ini* file. The authorizer instance receives all options from the section as arguments to the *__init__()* method.

To use the above authorizer, add the following to the `slicer.ini`:

```
[workspace]
authorization: custom

[authorization]
url: http://localhost/authorization_service
user_dimension: user
```

Authenticator

Authentication takes place at the server level right before a request is processed.

Custom authenticator has to be a subclass of `slicer.server.Authenticator` and has to have at least *authenticate(request)* method defined. Another optional method is *logout(request, identity)*.

Example authenticator which authenticates against a database table with two columns: *user* and *password* with a clear-text password (don't do that).

```
from cubes.server import Authenticator, NotAuthenticated
from sqlalchemy import create_engine, MetaData, Table

class DatabaseAuthenticator(Authenticator):
    def __init__(self, url=None, table=None, **options):

        self.engine = create_engine(url)
        metadata = MetaData(bind=engine)
        self.users = Table(table, metadata, autoload=True)

    def authenticate(self, request):
        user = request.values.get("user")
        password = request.values.get("password")

        select = self.users.select(self.users.c.password)
        select = select.where(self.users.c.user == user)

        row = self.engine.execute(select).fetchone()

        if row["password"] == password:
            return user
        else:
            raise NotAuthenticated
```

The *authenticate(request)* method should return the identity that will be later passed to the authorizer (it does not have to be the same value as a user name). The identity might even be *None* which might be interpreted by some authorizers guest or not-logged-in visitor. The method should raise *NotAuthenticated* when the credetials don't match.

CHAPTER 8

Developer's Reference

Workspace Reference

Workspace manages all cubes, their data stores and model providers.

Model Reference

Model - Cubes meta-data objects and functionality for working with them. *Logical Model and Metadata*

Note: All model objects: *Cube*, *Dimension*, *Hierarchy*, *Level* and attribute objects should be considered immutable once created. Any changes to the object attributes might result in unexpected behavior.

See also:

Model Providers Reference Model providers – objects for constructing model objects from other kinds of sources, even during run-time.

Model Utility Functions

Model components

Cube

Dimension, Hierarchy and Level

Attributes, Measures and Aggregates

exception `ModelError`
Exception raised when there is an error with model and its structure, mostly during model construction.

exception `ModelIncosistencyError`
Raised when there is incosistency in model structure, mostly when model was created programatically in a wrong way by mismatching classes or misonfiguration.

exception NoSuchDimensionError
Raised when a dimension is requested that does not exist in the model.

exception NoSuchAttributeError
Raised when an unknown attribute, measure or detail requested.

Model Providers Reference

See also:

Model Reference

Model Providers

Model Metadata

Aggregation Browser Reference

Abstraction for aggregated browsing (concrete implementation is provided by one of the backends in package `backend` or a custom backend).

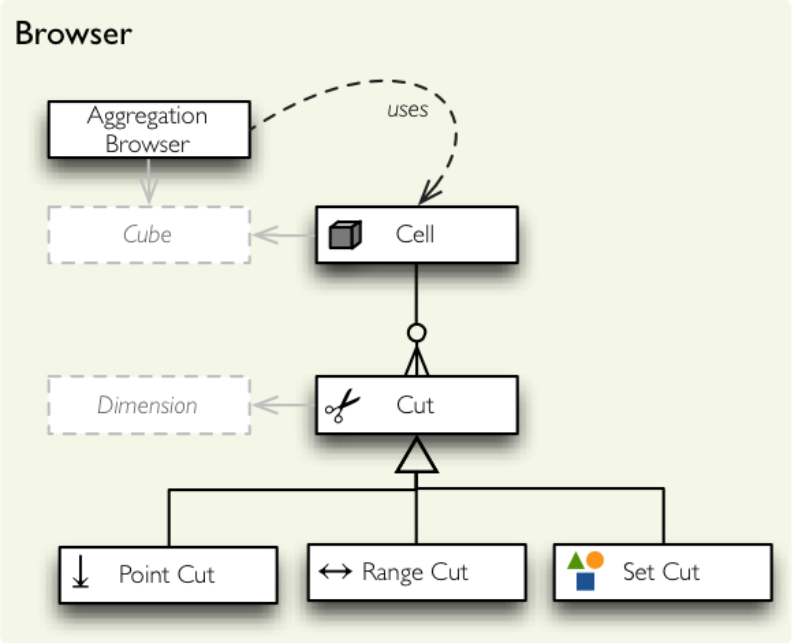


Fig. 8.1: Browser package classes.

Aggregate browsing

Result

The result of aggregated browsing is returned as object:

Facts

Slicing and Dicing

Cuts

Drilldown

String conversions

In applications where slicing and dicing can be specified in form of a string, such as arguments of HTTP requests of an web application, there are couple helper methods that do the string-to-object conversion:

Mapper

Formatters Reference

Formatters

See also:

Data Formatters Formatters documentation.

Aggregation Browsing Backends

Built-in backends for browsing aggregates of various data sources.

Other backends can be found at https://github.com/DataBrewery.

SQL

SQL backend uses SQLAlchemy for generating queries. It supports all databases that the SQLAlchemy supports such as:

- Drizzle
- Firebird
- Informix
- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL
- SQLite
- Sybase

Browser

Slicer

HTTP WSGI OLAP Server Reference

Light-weight HTTP WSGI server based on the Flask framework. For more information about using the server see *OLAP Server*.

`cubes.server.`**`slicer`**
Flask Blueprint instance.

See *Integration With Flask Application* for a use example.

`cubes.server.`**`workspace`**
Flask *Local* object referring to current application's workspace.

Authentication and Authorization

See also:

Authorization and Authentication

Authentication

Authorization

Utility functions

CHAPTER 9

Release Notes

Cubes Release Notes

Cubes 2.0 release notes

Moved to Python 3.6.

- SQL Alchemy is now required dependency, as the focus is now SQL query generator.

Overview

Major change is full move to Python 3.6 and dropping compatibility with lesser versions of Python.

Naming Conventions

The naming conventions were moved from the `[server]` section of the config file and moved to a separate `[naming]` section.

Migration from 1.x to 2.0

Model Changes:

- All joins in the model must be specified as dictionaries, not as tuples

Configuration Changes:

- Naming conventions (dimension prefix, fact prefix, etc.) should be moved from the `[server]` to the `[naming]` section
- There must be no unknown configuration settings in the *.ini* file that are not recognized by the library. (Note: If you think the option should be accepted, please file an issue in the Cubes issue tracker)

Cubes 1.1 release notes

These release notes cover the new features and changes (some of them backward incompatible).

Overview

This release brings major refactoring and complexity reduction of the SQL backend. Other notable changes:

- implementation of arithmetic expressions
- removal of all backends but SQL and Slicer into a separate packages
- removal of all non-essential modules as extensions in separate packages

New Features

Model

- changed all *create_** methods into a model object class initializers *from_metadata* such as *Cube.from_metadtata()* or *Dimension.from_metadata()*

Cube:

- `Cube.base_attributes()` - returns all attributes that don't have expressions and are very likely represented by a physical column
- `Cube.attribute_dependencies()` - returns a dictionary saying which attribute directly depends on which other attributes
- `Cube.collect_dependencies()` - dictionary of all, deep dependencies (whole attribute dependency tree is expanded)

Attributes

Expressions

Attributes can now carry an arithmetic expression. Attributes used in the expressions must be other logical attributes. Only base attributes (those without expressions) require to have physical column mappings.

Example:

```
{"name": "price_with_vat", "expression": "price * 1.2"}
{"name": "price_with_discount", "expression": "price * (1 - discount / 100)"}
```

The expressions currently support basic arithmetics and few SQL functions. The expression language and operators are inspired (and will very likely follow) the Postgres SQL dialect, but is not going to be 100% compatible. Language will be extended gently, with regard to other backends or SQL dialects. (Note that the expression language is meant to be shared with other, non-Cubes tools).

Plugins

New plugin system. Packages can now advertise in their `setup.py` plugins:

..code-block:: python

entry_points={

'cubes.stores': ['my = my_package.MyStore',

], 'cubes.authorizers': [

'my = my_package.MyAuthorizer',

]

}

Extensible obects: *authenticators*, *authorizers*, *browsers*, *formatters*, *model_providers* and *stores*.

Major Changes

Modules and Packages

The modules were restructured. The *backend* package was removed, it's content was separated into external packages. *sql* became a top-level package, yet maintaining it's optional status. It should stay in the Cubes package as it is the most used backend.

browser was split into two separate packages *browser* and *cells*.

New external packages:

- *cubes-ga*
- *cubes-mongo*
- *cubes-mixpanel*
- **important**: No longer generate implicit aggregates by default. Override in model

Model

- *Cube.all_attributes* was changed to return actually all attributes of the Cube instead of just attributes for a fact table (non-aggregates). There are now three methods: `Cubes.all_attributes()`, *Cubes.all_fact_attributes* and *Cubes.all_aggregation_attributes*.

Model Attributes:

- string representation of attributes now returns attribute reference instead of attribute name
- *ref* is now a property of all attributes (originally it was a function *ref(locale, simplify)*)
- attribute reference is now opinionated without ability to have alternative way: all dimensions are simplified if they are flat and have no details, otherwise attribute reference is *dimension.attribute*
- removed public_dimensions()

Other

- removed `store_name` in *Store*
- added *Drilldown.natural_order*

SQL

Now a top-level package as it will receive more attention in the near future. Simplified, made code more understandable and maintainable.

- new SQL schema object holding information about the star/snowflake schema
- topological sort of joins - joins are now ordered automagically, no longer cryptic exceptions about *to-fact relationships*
- new QueryContext – replaces QueryBuilder
- support for SQL Alchemy selectables as star/snowflake schema tables
- removed simple vs. composed aggregation statement (which was required due to unpredictability of low-level mapping expressions), now every statement is just "simple" statement

Other:

- *find_dimension()* and *link_cube()* are now global functions. Cube linking has been moved into the provider.
- added *naming* convention dicitonary to the SQL mapper

- added SQLSchemaInspector
- SQLStore accepts metadata object
- added compound keys (multiple columns) in joins

Fix:

- if fact table schema is explicitly specified, use it in the joins as default schema

Slicer

The `slicer` command has been rewritten using Click. There are new commands and refreshed commands:

- `ext-info` – list extensions and give more details about particuliar extension
- `materialize` and `aggregate` – brought back under new `sql` command group
- `list` – list cubes

The configuration `slicer.ini` is now as default and does not have to be explicitly provided if not necessary.

Removed

- Dropped support for experimental "nonadditive" measures (temporarily)
- Dropped support for experimental periods-to-date (requires specification)
- Dropped support of experimental `expr` mapping (permanently)

Cubes 1.0 release notes

These release notes cover the new features and changes (some of them backward incompatible).

Overview

The biggest new feature in cubes is the "pluggable" model. You are no longer limited to one one model, one type of data store (database) and one set of cubes. The new *Workspace* is now framework-level controller object that manages models (model sources), cubes and datastores. To the future more features will be added to the workspace.

New Workspace related objects:

- model provider – creates model objects from a model source (might be a foreign API/service or custom database)
- store – provides access and connection to cube's data

For more information see the *Workspace* documentation.

Other notable new features in Cubes 1.0 are:

- Rewritten Slicer server in Flask as a reusable Blueprint.
- New *server API*.
- support for *outer joins* in the *SQL backend*.
- Distinction between *measures and aggregates*
- Extensible *authorization and authentication*
- Visualizer

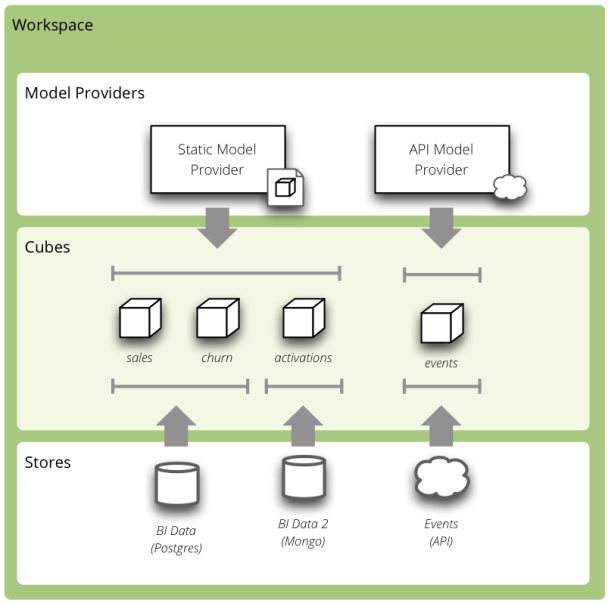


Fig. 9.1: Analytical Workspace Overview

Python Versions

Cubes 1.0 supports Python >= 2.7 for Python 2 series and Python >= 3.4.1 for Python 3 series.

Analytical Workspace

The old backend architecture was limiting. It allowed only one store to be used, the model had to be known before the server started, it was not possible to get the model from a remote source.

For more details about the new workspace see the *Analytical Workspace* documentation.

Configuration

The *slicer.ini* configuration has changed to reflect new features.

The section `[workspace]` now contains global configuration of a cubes workspace session. The database connection has moved into `[store]` (or similar, if there are more).

The database connection is specified either in the `[store]` section or in a separate `stores.ini` file where one section is one store, section name is store name (as referenced from cube models).

If there is only one model, it can be specified either in the `[workspace]` section as `model`. Multiple models are specified in the `[models]` section.

To sum it up:

- `[server] backend` is now `[store] type` for every store
- `[server] log` and `log_level` has moved to `[workspace]`
- `[model]` is now either `model` option of `[workspace]` or list of multiple models in the `[models]` section

The old configuration:

```
[server]
host: localhost
port: 5000
```

```
reload: yes
log_level: info

[workspace]
url: postgres://localhost/mydata"

[model]
path: grants_model.json
```

Is now:

```
[workspace]
log_level: info
model: grants_model.json

[server]
host: localhost
port: 5000
reload: yes

[store]
type: sql
url: postgres://localhost/mydata
```

Check your configuration files.

See also:

Configuration

Server

Slicer server is now a Flask application and a reusable Blueprint. It is possible to include the Slicer in your application at an end-point of your choice.

For more information, see the *recipe*.

Other server changes:

- do not expose internal exceptions, only user exceptions
- added simple authentication methods: HTTP Basic (behind a proxy) and parameter-based identity. Both are permissive and serve just for passing an identity to the authorizer.

HTTP Server API

Server end-points have changed.

New end-points:

- `/version`
- `/info`
- `/cubes`
- `/cube/<cube>/model`
- `/cube/<cube>/aggregate`
- `/cube/<cube>/facts`
- `/cube/<cube>/fact`
- `/cube/<cube>/members/<dimension>`

- `/cube/<cube>/cell`
- `/cube/<cube>/report`

Removed end-points:

- `/model` – without replacement doe to the new concepts of workspace. Alternative is to get list of basic cube info using `/cubes`.
- `/model/cubes` – without replacement, use `/cubes`
- `/model/cube/<cube>` – use `/cube/<cube>/model` instead
- `/model/dimension/*` – without replacement due to the new concepts of workspace
- all top-level browser actions such as `/aggregate` – now the cube name has to be explicit

Parameter changes:

- `/aggregate` uses `aggregates=`, does not accept `measure=` any more
- `/aggregate` now accepts `format=` to generate CSV output
- new parameter `headers=` for CSV output: with headers as attribute names, headers as attribute labels (human readable) or no headers at all
- it is now possible to specify multiple drilldowns, separated by `|` in one `drilldown=` parameter
- cuts for date dimension accepts named relative time references such as `cut=date:90daysago-today`. See the *server documentation* for more information.
- dimension path elements can contain special characters if they are escaped by a backslash `\` such as `cut=city:Nové\ Mesto`

Many actions now accept `format=` parameter, which can be `json`, `csv` or `json_lines` (new-line separated JSON).

Response changes:

- `/cubes` (replacement for `/model`) returns a list of basic cubes info: *name*, *label*, *description* and *category*. It does not return full cube description with dimensions.
- `/cube/<cube>/model` has new keys: `aggregates` and `features`

See also:

OLAP Server

Outer Joins

Support for thee types of joins was added to the SQL backend: *match* (inner), *master* (left outer) and *detail* (right outer).

The *outer joins* allows for example to use whole `date` dimension table and have "empty cells" for dates where there are no facts.

When an right outer join (`detail` method) is present, then aggregate values are coalesced to zero (based on the function either the values or the result is coalesced). For example: AVG coalesces values: `AVG(COALESCE(c, 0))`, SUM coalesces result: `COALESCE(SUM(c), 0)`.

See also:

SQL Backend – Outer Joins Documentation

Statutils

Module with statistical aggregate functions such as simple moving average or weighted moving average.

Provided functions:

- `wma` – weighted moving average
- `sma` – simple moving average
- `sms` – simple moving sum
- `smstd` – simple moving st. deviation
- `smrsd` – simple moving relative st. deviation
- `smvar` – simple moving variance

The function are applied on the already computed aggregation results. Backends migh handle the function internally if they can.

Window functions respect *window_size* property of aggregates.

Browser

- cuts now have an *invert* flag (might not be supported by all backends)
- *aggregate()* has new argument *split* which is a cell that defines artificial flag-like dimension with two values: *0* – aggergated cell is outside of the split cell, *1* – aggregated cell is within the split cell

Both *invert* and *split* features are still provisional, their interface might change.

Slicer

- added `slicer model convert` to convert between json and directory bundle

Model

Model and modeling related changes are:

- new concept of model providers (see *details below*)
- measure aggregates (see *details below*)
- cardinality of dimensions and dimension levels
- dimension and level roles
- attribute missing values
- *format* property of a measure and aggregate
- namespaces

Note: `cubes`, `dimensions`, `levels` and `hierarchies` can no longer be dictionaries, they should be lists of dictionaries and the dictionaries should have a `name` property set. This was depreciated long ago.

Model Providers

The models of cubes are now being created by the *model providers*. Model provider is an object that creates *Cubes* and *Dimension* instances from it's source. Built-in model provider is `cubes.StaticModelProvider` which creates cubes objects from JSON files and dictionaries.

See also:

Model Providers, *Model Providers Reference*

Namespaces

Cubes from stores can be wrapped in a model namespace. By-default, the namespace is the same as the name of the store. The cubes are referenced as *NAMESPACE.CUBE* such as *foreign.sales*. For backward compatibility reasons and for simplicity there are two cube lookup methods: *recursive* and *global*.

Measures and Aggregates

Cubes now distinguishes between *measures* and *aggregates*. *measure* represents a numerical fact property, *aggregate* represents aggregated value (applied aggregate function on a property, or provided natively by the backend).

This new approach of *aggregates* makes development of backends and clients much easier. There is no need to construct and guess aggregate measures or splitting the names from the functions. Backends receive concrete objects with sufficient information to perform the aggregation (either by a function or fetch already computed value).

Functionality additions and changes:

- New model objects: `cubes.Attribute` (for dimension or detail), `cubes.Measure` and `cubes.MeasureAggregate`.
- New model creation/helper functions: `cubes.create_measure_aggregate()`, `cubes.create_measure()`
- `cubes.create_cube()` is back
- `cubes.Cube.aggregates_for_measure()` – return all aggregates referring the measure
- `cubes.Cube.get_aggregates()` – get a list of aggregates according to names
- `cubes.Measure.default_aggregates()` – create a list of default aggregates for the measure
- `calculators_for_aggregates()` in statutils – returns post-aggregation calculators
- Added a cube metadata flag to control creation of default aggregates: *implicit_aggregates*. Default is `True`
- Cube initialization has no creation of defaults – it should belong to the model provider or `create_cube()` function
- If there is no function specified, we consider the aggregate to be specified in the mappings

record_count

Implicit aggregate *record_count* is no longer provided for every cube. It has to be explicitly defined as an aggregate:

```
"aggregates": [
    {
        "name": "item_count",
        "label": "Total Items",
        "function": "count"
    }
]
```

It can be named and labelled in any way.

If cube has no aggregates, then new default aggregate named *fact_count* is created.

See also:

Measures and Aggregates Documentation, *Logical Model and Metadata*

Dimension Links

Linking of dimensions to cubes can be fine-tuned by specifying multiple properties of the dimension in the cube's context:

- *hierarchies* – cube's dimension can have only certain hierarchies from the original dimension
- *detault_hierarchy_name* – it is possible to specify different default hierarchy
- *nonadditive* – override the dimensions' non-additive property
- *cardinality* – use if dimension might have different cardinality in the new context
- *alias* – reuse dimensions in a cube but give them different names

Backends

- Backends should now implement *provide_aggregate()* method instead of *aggregate()* – the later takes care of argument conversion and preparation. See *Backends* for more information.

SQL Backend

- New module `functions` with new AggregationFunction objects
- Added get_aggregate_function() and available_aggregate_functions()
- Renamed `star` module to `browser`
- Updated the code to use the new aggregates instead of old measures. Affected parts of the code are now cleaner and more understandable
- Moved calculated_aggregations_for_measure to library-level statutils module as calculators_for_aggregates
- function dictionary is no longer used

New Backends

- *Mixpanel*: ../backends/mixpanel
- *Slicer*: *Slicer Server*
- *Mongo*: ../backends/mongo
- *Google Analytics*: ../backends/google_analytics

See also:

How to Write a Backend Extension

Visualizer

There is a cubes visualizer included in the Cubes that can connect to any cubes slicer server over HTTP. It is purely HTML/JavaScript application.

Other Minor Changes

- Cell.contains_level(dim, level, hierarhy) – returns `True` when the cell contains level `level` of dimension `dim`
- renamed *AggregationBrowser.values()* to `cubes.AggregationBrowser.members()`
- *AggregationResult.measures* changed to *AggregationResult.aggregates* (see `AggregationResult`)

- browser's *__init__* signature has changed to include the store
- changed the exception hierarchy. Now has two branches: `UserError` and `InternalError` – the `UserError` can be returned to the client, the `InternalError` should remain privade on the server side.
- `to_dict()` of model objects returns an ordered dictionary for nicer JSON output
- New class `cubes.Facts` that should be returned by `cubes.AggregationBrowser.facts()`
- `cubes.cuts_from_string()` has two new arguments *member_converters* and *role_member_converters*
- New class `cubes.Drilldown` to get more information about the drilldown

Migration to 1.0

Checklists for migrating a Cubes project from pre-1.0 to 1.0:

The `slicer.ini`

1. Rename `[workspace]` to `[store]`
2. Create new empty `[workspace]`
3. Move `[server] backend` to `[store] type`
4. Move `[server] log`, `log_level` to the new `[workspace]`
5. Rename `[model] path` to `[models] main` and remove all non-model references (such as `locales`).

The minimal configuration looks like:

```
[store]
type: sql
url: sqlite:///data.sqlite

[models]
main: model.json
```

See *configuration changes* for an example and *configuration documentation* for more information.

The Model

There are not many model changes, mostly measures and aggregates related.

1. Make sure that `dimensions`, `cubes`, `levels` and `hierarchies` are not dictionaries but lists of dictionaries with `name` property.
2. Create the explicit `record_count` aggregate, if you are using it. Note that you can name and label the aggregate as you like.

```
"aggregates": [
    {
        "name": "record_count",
        "label": "Total Items",
        "function": "count"
    }
]
```

3. In `measures` rename `aggregations` to `aggregates` or even better: create explicit, full aggregate definitions.

See *Aggregates* for more information.

Slicer Front-end

The biggest change in the front-ends is the removal of the `/model` end-point without equivalend replacement. Use `/cubes` to get list of provided cubes. The cube definition contains whole dimension descriptions.

1. Change from `/model` to `/cubes`
2. Change from `/model/cube/<name>` to `/cube/<name>/model`
3. Cube has to be explicit in every request, therefore `/aggregate` does not work any more, use `/cube/<name>/aggregate`
4. Change `aggregate` parameter `measure` to `aggregates`

Refer to the *OLAP Server* documentation for the new response structures. There were minor changes, mostly additions.

Additional and Optional Considerations for Migration

- if your model is too big, split it into multiple models and add them to the `[models]` section. Note that the dimensions can be shared between models.
- put all your models into a separate directory and use the `[workspace] models_path` property. The paths in `[models]` are relative to the `models_path`
- if you have muliple stores, create a separate `stores.ini` file where the section names are store names. Set the `[workspace] stores` to the `stores.ini` path if it is different than default.
- Add `"role"="time"` to a *date* dimension – you might benefit from new date-related additions and special dimension handling in the available front-ends
- Review `joins` and set appropriate join method if desired, for example `detail` for a date dimension.
- Add `cardinality` metadata to dimension levels if appropriate.
- Look at the cube's model `features` property to learn what the front-end can expect from the backend for that cube
- Look at the `/info` response

v1.0.1 Changes

- [feature] Added SimpleAuthorizer.expand_roles
- [feature] create indexes for aggregated table
- [change] make workspace optional
- [change] Allow user to supply an external workspace to the slicer
- [change] modified create_cube_aggregate
- [fix] correct physical attribute schema handling in SQL backend - fact details were getting dimension schema
- [fix] increase debug level in hello_world example
- [fix] more descriptive error messages in browser/backend
- [fix] Use store instead of datastore (remaining places)
- various documentation fixes
- various example fixes

Contributors:

- Dmitriy Trochshenko
- Friedrich Lindenberg

- Lucas Taylor
- Michal Skop
- Gasper Zejn
- jerry dumblauskas

Cubes 0.6 to 0.10.2 Release Notes

0.10.2

Summary:

- many improvements in handling multiple hierarchies
- more support of multiple hierarchies in the slicer server either as parameter or with syntax `dimension@hierarchy`:
  - dimension values: `GET /dimension/date?hierarchy=dqmy`
  - cut: get first quarter of 2012 `?cut=date@dqmy:2012,1`
  - drill-down on hierarchy with week on implicit (next) level: `?drilldown=date@ywd`
  - drill-down on hierarchy with week with exlpicitly specified week level: `?drilldown=date@ywd:week`
- order and order attribute can now be specified for a Level
- optional safe column aliases (see docs for more info) for databases that have non-standard requirements for column labels even when quoted

Thanks:

- Jose Juan Montes (@jjmontesl)
- Andrew Zeneski
- Reinier Reisy Quevedo Batista (@rquevedo)

New Features

- added *order* to Level object - can be `asc`, `desc` or None for unspecified order (will be ignored)
- added *order_attribute* to Level object - specifies attribute to be used for ordering according to *order*. If not specified, then first attribute is going to be used.
- added hierarchy argument to *AggregationResult.table_rows()*
- *str(cube)* returns cube name, useful in functions that can accept both cube name and cube object
- added cross table formatter and its HTML variant
- `GET /dimension` accepts hierarchy parameter
- added *create_workspace_from_config()* to simplify workspace creation directly from slicer.ini file (this method might be slightly changed in the future)
- *to_dict()* method of model objects now has a flag *create_label* which provides label attribute derived from the object's name, if label is missing
- #95: Allow charset to be specified in Content-Type header

SQL:

- added option to SQL workspace/browser `safe_labels` to use safe column labels for databases that do not support characters like `.` in column names even when quoted (advanced feature, does not work with denormalization)

- browser accepts include_cell_count and include_summary arguments to optionally disable/enable inclusion of respective results in the aggregation result object
- added implicit ordering by levels to aggregate and dimension values methods (for list of facts it is not yet decided how this should work)
- #97: partially implemented sort_key, available in *aggregate()* and *values()* methods

Server:

- added comma separator for `order=` parameter
- reflected multiple search backend support in slicer server

Other:

- added vim syntax highlighting goodie

Changes

- AggregationResult.cross_table is depreciated, use cross table formatter instead
- *load_model()* loads and applies translations
- slicer server uses new localization methods (removed localization code from slicer)
- workspace context provides proper list of locales and new key 'translations'
- added base class Workspace which backends should subclass; backends should use workspace.localized_model(locale)
- *create_model()* accepts list of translations

Fixes

- browser.set_locale() now correctly changes browser's locale
- #97: Dimension values call cartesians when cutting by a different dimension
- #99: Dimension "template" does not copy hierarchies

0.10.1

Quick Summary:

- multiple hierarchies:
  - Python: `cut = PointCut("date", [2010,15], hierarchy='ywd')`
  - Server: `GET /aggregate?cut=date@ywd:2010,15`
  - Server drilldown: `GET /aggregate?drilldown=date@ywd:week`
- added experimental result formatters (API might change)
- added experimental pre-aggregations

New Features

- added support for multiple hierarchies
- added `dimension_schema` option to star browser – use this when you have all dimensions grouped in a separate schema than fact table
- added *HierarchyError* - used for example when drilling down deeper than possible within that hierarchy

- added result formatters: simple_html_table, simple_data_table, text_table
- added create_formatter(formatter_type, options ...)
- AggregationResult.levels is a new dictionary containing levels that the result was drilled down to. Keys are dimension names, values are levels.
- AggregationResult.table_rows() output has a new variable `is_base` to denote whether the row is base or not in regard to table_rows dimension.
- added `create_server(config_path)` to simplify wsgi script
- added aggregates: avg, stddev and variance (works only in databases that support those aggregations, such as PostgreSQL)
- added preliminary implemenation of pre-aggregation to sql worskspace:
  - *create_conformed_rollup()*
  - *create_conformed_rollups()*
  - *create_cube_aggregate()*

Server:

- multiple drilldowns can be specified in single argument: `drilldown=date,product`
- there can be multiple `cut` arguments that will be appended into single cell
- added requests: `GET /cubes` and `GET /cube/NAME/dimensions`

Changes

- **Important:** Changed string representation of a set cut: now using semicolon ';' as a separator instead of a plus symbol '+'
- aggregation browser subclasses should now fill result's `levels` variable with `coalesced_drilldown()` output for requested drill-down levels.
- Moved coalesce_drilldown() from star browser to cubes.browser module to be reusable by other browsers. Method might be renamed in the future.
- if there is only one level (default) in a dimension, it will have same label as the owning dimension
- hierarchy definition errors now raise ModelError instead of generic exception

Fixes

- order of joins is preserved
- fixed ordering bug
- fixed bug in generating conditions from range cuts
- `AggregationResult.table_rows` now works when there is no point cut
- get correct reference in `table_rows` – now works when simple denormalized table is used
- raise model exception when a table is missing due to missing join
- search in slicer updated for latest changes
- fixed bug that prevented using cells with attributes in aliased joined tables

0.10

Quick Summary

- Dimension defition can have a "template". For example:

```
{
    "name": "contract_date",
    "template": "date"
}
```

- added table_rows() and cross_table()
- added simple_model(cube_name, dimension_names, measures)

Incompatibilities: **use `create_model()` instead of `Model(**dict)`, if you** were using just `load_model()`, you are fine.

New Features

- To address issue #8 create_model(dict) was added as replacement for Model(dict). Model() from now on will expect correctly constructed model objects. `create_model()` will be able to handle various simplifications and defaults during the construction process.
- added `info` attribute to all model objects. It can be used to store custom, application or front-end specific information
- preliminary implementation of `cross_table()` (interface might be changed)
- `AggregationResult.table_rows()` - new method that iterates through drill-down rows and returns a tuple with key, label, path, and rest of the fields.
- dimension in model description can specify another template dimension – all properties from the template will be inherited in the new dimension. All dimension properties specified in the new dimension completely override the template specification
- added *point_cut_for_dimension*
- added *simple_model(cube_name, dimensions, measures)* – creates a single-cube model with flat dimensions from a list of dimension names and measures from a list of measure names. For example:

```
model = simple_model("contracts", ["year","contractor", "type"], ["amount"])
```

Slicer Server:

- `/cell` – return cell details (replaces `/details`)

Changes

- creation of a model from dictionary through Model(dict) is depreciated, use *create_model(dict)* instead. All initialization code will be moved there. Depreciation warnings were added. Old functionality retained for the time being. (**important**)
- Replaced *Attribute.full_name()* with *Attribute.ref()*
- Removed *Dimension.attribute_reference()* as same can be achieved with dim(attr).ref()
- *AggregationResult.drilldown* renamed to *AggregationResults.cells*

Planned Changes:

- *str(Attribute)* will return *ref()* instead of attribute name as it is more useful

Fixes

- order of dimensions is now preserved in the Model

0.9.1

Summary: Range cuts, denormalize with slicer tool, cells in `/report` query

New Features

- *cut_from_string()*: added parsing of range and set cuts from string; introduced requirement for key format: Keys should now have format "alphanumeric character or underscore" if they are going to be converted to strings (for example when using slicer HTTP server)
- *cut_from_dict()*: create a cut (of appropriate class) from a dictionary description
- *Dimension.attribute(name)*: get attribute instance from name
- added exceptions: *CubesError*, *ModelInconsistencyError*, *NoSuchDimensionError*, *NoSuchAttributeError*, *ArgumentError*, *MappingError*, *WorkspaceError* and *BrowserError*

StarBrowser:

- implemented RangeCut conditions

Slicer Server:

- `/report` JSON now accepts `cell` with full cell description as dictionary, overrides URL parameters

Slicer tool:

- `denormalize` option for (bulk) denormalization of cubes (see the the slicer documentation for more information)

Changes

- all `/report` JSON requests should now have queries wrapped in the key `queries`. This was originally intended way of use, but was not correctly implemented. A descriptive error message is returned from the server if the key `queries` is not present. Despite being rather a bug-fix, it is listed here as it requires your attention for possible change of your code.
- warn when no backend is specified during slicer context creation

Fixes

- Better handling of missing optional packages, also fixes #57 (now works without slqalchemy and without werkzeug as expected)
- see change above about `/report` and `queries`
- push more errors as JSON responses to the requestor, instead of just failing with an exception

Version 0.9

Important Changes

Summary of most important changes that might affect your code:

Slicer: Change all your slicer.ini configuration files to have `[workspace]` section instead of old `[db]` or `[backend]`. Depreciation warning is issued, will work if not changed.

Model: Change `dimensions` in `model` to be an array instead of a dictionary. Same with `cubes`. Old style: `"dimensions" = { "date" = ... }` new style: `"dimensions" = [ { "name": "date", ... } ]`. Will work if not changed, just be prepared.

Python: Use Dimension.hierarchy() instead of Dimension.default_hierarchy.

New Features

- *slicer_context()* - new method that holds all relevant information from configuration. can be reused when creating tools that work in connected database environment
- added *Hierarchy.all_attributes()* and *.key_attributes()*
- *Cell.rollup_dim()* - rolls up single dimension to a specified level. this might later replace the Cell.rollup() method
- *Cell.drilldown()* - drills down the cell
- *create_workspace()* - new top-level method for creating a workspace by name of a backend and a configuration dictionary. Easier to create browsers (from possible browser pool) programmatically. The browser name might be full module name path or relative to the cubes.backends, for example `sql.browser` for default SQL denormalized browser.
- *get_backend()* - get backend by name
- AggregationBrowser.cell_details(): New method returning values of attributes representing the cell. Preliminary implementation, return value might change.
- AggregationBrowser.cut_details(): New method returning values of attributes representing a single cut. Preliminary implementation, return value might change.
- Dimension.validate() now checks whether there are duplicate attributes
- Cube.validate() now checks whether there are duplicate measures or details

SQL backend:

- new StarBrowser implemented:
  - StarBrowser supports snowflakes or denormalization (optional)
  - for snowflake browsing no write permission is required (does not have to be denormalized)
- new *DenormalizedMapper* for mapping logical model to denormalized view
- new *SnowflakeMapper* for mapping logical model to a snowflake schema
- *ddl_for_model()* - get schema DDL as string for model
- join finder and attribute mapper are now just Mapper - class responsible for finding appropriate joins and doing logical-to-physical mappings
- *coalesce_attribute()* - new method for coalescing multiple ways of describing a physical attribute (just attribute or table+schema+attribute)
- dimension argument was removed from all methods working with attributes (the dimension is now required attribute property)
- added *create_denormalized_view()* with options: materialize, create_index, keys_only

Slicer:

- slicer ddl - generate schema DDL from model
- slicer test - test configuration and model against database and report list of issues, if any
- Backend options are now in [workspace], removed configurability of custom backend section. Warning are issued when old section names [db] and [backend] are used
- server responds to /details which is a result of AggregationBrowser.cell_details()

Examples:

- added simple Flask based web example - dimension aggregation browser

Changes

- in Model: dimension and cube dictionary specification during model initialization is depreciated, list should be used (with explicitly mentioned attribute "name") – **important**
- **important**: Now all attribute references in the model (dimension attributes, measures, ...) are required to be instances of Attribute() and the attribute knows it's dimension
- removed *hierarchy* argument from *Dimension.all_attributes()* and *Dimension.key_attributes()*
- renamed builder to denormalizer
- Dimension.default_hierarchy is now depreciated in favor of Dimension.hierarchy() which now accepts no arguments or argument None - returning default hierarchy in those two cases
- metadata are now reused for each browser within one workspace - speed improvement.

Fixes

- Slicer version should be same version as Cubes: Original intention was to have separate server, therefore it had its own versioning. Now there is no reason for separate version, moreover it can introduce confusion.
- Proper use of database schema in the Mapper

Version 0.8

New Features

- Started writing StarBrowser - another SQL aggregation browser with different approach (see code/docs)

Slicer Server:

- added configuration option `modules` under `[server]` to load additional modules
- added ability to specify backend module
- backend configuration is in [backend] by default, for SQL it stays in [db]
- added server config option for default `prettyprint` value (useful for demontration purposes)

Documentation:

- Changed license to MIT + small addition. Please refer to the LICENSE file.
- Updated documentation - added missing parts, made reference more readable, moved class and function reference docs from descriptive part to reference (API) part.
- added backend documentation
- Added "Hello World!" example

Changed Features

- removed default SQL backend from the server
- moved worskpace creation into the backend module

Fixes

- Fixed create_view to handle not materialized properly (thanks to deytao)
- Slicer tool header now contains #!/usr/bin/env python

Version 0.7.1

Added tutorials in tutorials/ with models in tutorials/models/ and data in tutorials/data/:

- **Tutorial 1:**
  - how to build a model programatically
  - how to create a model with flat dimensions
  - how to aggregate whole cube
  - how to drill-down and aggregate through a dimension
- **Tutorial 2:**
  - how to create and use a model file
  - mappings
- **Tutorial 3:**
  - how hierarhies work
  - drill-down through a hierarchy
- **Tutorial 4 (not blogged about it yet):**
  - how to launch slicer server

New Features

- New method: *Dimension.attribute_reference*: returns full reference to an attribute
- str(cut) will now return constructed string representation of a cut as it can be used by Slicer

Slicer server:

- added /locales to slicer
- added locales key in /model request
- added Access-Control-Allow-Origin for JS/jQuery

Changes

- Allow dimensions in cube to be a list, not only a dictionary (internally it is ordered dictionary)
- Allow cubes in model to be a list, not only a dictionary (internally it is ordered dictionary)

Slicer server:

- slicer does not require default cube to be specified: if no cube is in the request then try default from config or get first from model

Fixes

- Slicer not serves right localization regardless of what localization was used first after server was launched (changed model localization copy to be deepcopy (as it should be))
- Fixes some remnants that used old Cell.foo based browsing to Browser.foo(cell, ...) only browsing
- fixed model localization issues; once localized, original locale was not available
- Do not try to add locale if not specified. Fixes #11: https://github.com/Stiivi/cubes/issues/11

Version 0.7

WARNING: Minor backward API incompatibility - Cuboid renamed to Cell.

Changes

- Class 'Cuboid' was renamed to more correct 'Cell'. 'Cuboid' is a part of cube with subset of dimensions.
- all APIs with 'cuboid' in their name/arguments were renamed to use 'cell' instead
- Changed initialization of model classes: Model, Cube, Dimension, Hierarchy, Level to be more "pythony": instead of using initialization dictionary, each attribute is listed as parameter, rest is handled from variable list of key word arguments
- Improved handling of flat and detail-less dimensions (dimensions represented just by one attribute which is also a key)

Model Initialization Defaults:

- If no levels are specified during initialization, then dimension name is considered flat, with single attribute.
- If no hierarchy is specified and levels are specified, then default hierarchy will be created from order of levels
- If no levels are specified, then one level is created, with name `default` and dimension will be considered flat

Note: This initialization defaults might be moved into a separate utility function/class that will populate incomplete model

New features

Slicer server:

- changed to handle multiple cubes within model: you have to specify a cube for /aggregate, /facts,... in form: /cube/<cube_name>/<browser_action>
- reflect change in configuration: removed `view`, added `view_prefix` and `view_suffix`, the cube view name will be constructed by concatenating `view prefix` + `cube name` + `view suffix`
- in aggregate drill-down: explicit dimension can be specified with `drilldown=dimension:level`, such as: `date:month`

This change is considered final and therefore we can mark it is as API version 1.

Version 0.6

New features

Cubes:

- added 'details' to cube - attributes that might contain fact details which are not relevant to aggregation, but might be interesting when displaying facts
- added ordering of facts in aggregation browser
- SQL denormalizer can now add indexes to key columns, if requested
- one detail table can be used more than once in SQL denomralizer (such as an organisation for both - receiver and donor), added key ``alias`` to ``joins`` in model description

Slicer server:

- added `log` a and `log_level` configuration options (under `[server]`)
- added `format=` parameter to `/facts`, accepts `json` and `csv`
- added `fields=` parameter to `/facts` - comma separated list of returned fields in CSV
- share single sqlalchemy engine within server thread
- limit number of facts returned in JSON (configurable by `json_record_limit` in `[server]` section)

Experimental: (might change or be removed, use with caution)

- added cubes searching frontend for separate cubes_search experimenal Sphinx backend (see https://bitbucket.org/Stiivi/cubes-search)

Fixes

- fixed localization bug in fact(s) - now uses proper attribute name without locale suffix
- fixed passing of pagination and ordering parameters from server to aggregation browser when requesting facts
- fixed bug when using multiple conditions in SQL aggregator
- make host/port optional separately

Contact and Getting Help

Join the chat at Gitter.

If you have questions, problems or suggestions, you can send a message to Google group or write to the author (Stefan Urbanek).

Report bugs in github issues tracking

There is an IRC channel `#databrewery` on server `irc.freenode.net`.

CHAPTER 10

License

Cubes is licensed under MIT license with small addition:

```
Copyright (c) 2011-2014 Stefan Urbanek, see AUTHORS for more details

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

Simply said, that if you use it as part of software as a service (SaaS) you have to provide the copyright notice in an about, legal info, credits or some similar kind of page or info box.

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index