university of groningen

University Medical Center Groningen

University of Groningen

Query driven visualization of large scale multi-dimensional astronomical catalogs

Buddelmeijer, Hugo

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2011

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Buddelmeijer, H. (2011). *Query driven visualization of large scale multi-dimensional astronomical catalogs*. s.n.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 04-02-2023

Chapter 3

Catalog Data in Astro-WISE

Abstract: We extend the Astro-WISE concept of Target Processing to source catalogs by implementing new Process Targets for catalog data. These Process Targets are called Source Collections, as described in Chapter 2, and take Target Processing to new heights. There is a base Source Collection class, from which classes are derived that correspond to common operations on catalogs, in analogy with the Frame classes for images.

Source Collection instances can be created without being processed, because the data lineage unambiguously defines what the processing results are. The processing is delayed until the catalog data is required and limited to subsets that are necessary for the processing of Process Targets further in the dependency graph. Furthermore, the resulting catalog data is only stored persistently if this is necessary for performance.

The partial processing is possible because the operations that correspond to every Source Collection class are well behaved and predictable and because Source Collection instances can share processing results by sharing persistent properties. This allows the information system to optimize a dependency graph before processing it, creating temporary transient copies of Source Collections in the dependency graph that represent a subset of the catalogs of the originals. These transient Source Collections are subsequently processed completely, and if necessary their processing result is stored persistently in the same location as the processing results of the original Source Collections.

This allows Source Collections to be created in the most reusable way, because scalability is implicit. This in turn, allows the creation of new Source Collections to be initiated by specifying the required catalogs in a very conceptual and declarative way, enhancing data pulling. Furthermore, the Source Collection classes are designed to facilitate query driven visualization.

3.1 Introduction

Before presenting the Source Collection implementation, we give a description of other functionality in Astro-WISE that is relevant to the Source Collections.

3.1.1 Processing in Astro-WISE

Astro-WISE (Mwebaze et al., 2009; McFarland, 2010) uses the advantages of Object-Oriented Programming (OOP) to model astronomical science products into OOP objects, called *process targets*. The database is used to link all process targets to their *progenitors* (dependencies), creating a *full data lineage* of the entire processing chain. Each process target has associated *processing parameters*, which are configurable parameters that guide the processing or reprocessing of the target. Each of these process target instances knows how to process itself. Processing is initiated by requesting the final desired result. This data pulling is the heart of Astro-WISE and is called *target processing*.

Every object correspond to a Python class that defines these properties and contains all code associated with the object. The properties of an object are store persistently in the database and existing objects can be instantiated by querying the database on these properties. There are a few base classes from which the other classes are derived and some classes that are not persistent or no process targets.

Source Collection Summary

In this chapter we describe the implementation of the Source Collection and associated classes, in which we take target processing to the extreme. The Source Collection classes are Process Targets for catalog data and described conceptually in chapter 2. The focus in this chapter is on the implementation itself, and we refer to chapter 2 for design details.

The Source Collection is a base class that represents a catalog of sources with attributes. Subclasses are derived from this base class, which correspond to different operations that can be performed on catalogs, in analogy with the Frame classes. Processing a Source Collection means creating the catalog data by performing this operation on the progenitors of the Source Collection (section 2.4.2). The operations that the derived classes perform range from selecting a subset of sources of another Source Collection to calculating new attributes of sources from existing attributes.

A Source Collection can be created without being processed: the catalog data itself does not have to be created to make a Source Collection persistent. Source Collections are only processed when data is requested and through optimization of the dependency graph only the required subsets of the catalogs are created. Many Source Collection classes can be processed on the fly, for example when the processing can be performed in SQL on the database. The processing result—the catalog—only has to be stored if this is necessary for performance.

The benefits of the Source Collections are summarized as (section 3.1.3): (1) Persistent sample selections of sources and attributes. (2) Support for derived attributes

with full data lineage and flexible method specification. (3) Pulling of catalog data that can be used for query driven visualization. (4) Indirect support for partial processing of targets. (5) On the fly optimization of dependency graphs. (6) Processing on either the database or Python layer. (7) Seamless integration of catalog handling in the database, awe-prompt and external visualization.

3.1.2 Existing Classes Overview

We give a short description of the Astro-WISE classes that necessary to introduce the Source Collection classes. In section 3.2 they are described in more detail.

Existing Class: DBObject and DataObject

The *DBObject* is the base class of all persistent objects in Astro-WISE. The *DataObject* class is derived from DBObject and is the base class for persistent classes that have an associated file on the dataserver.

Existing Class: ProcessTarget

The *ProcessTarget* class encapsulates the notion of a make-able object. The progenitors of a ProcessTarget have to be made, before the ProcessTarget itself can be made.

Existing Class: SourceList

Catalogs created from images are stored in Astro-WISE as a *SourceList* object. In a sense the Source Collections can be seen as a generalization of the SourceList class, or the SourceList as a special Source Collection (section 3.8.1).

Existing Class: AssociateList

Sources in SourceLists can be associated with one another through *AssociateLists*. An AssociateList only contains information on the identification between sources, it does not represent any attributes.

Existing Class: CombinedList

The *CombinedList* class is used to create a SourceList from an AssociateList. For example to combine SourceLists of different filters into one multicolor SourceList or to 'stitch' several SourceLists that each cover a part of the sky into one larger SourceList.

3.1.3 Source Collection Rationale

The Source Collection classes extend the catalog handling capabilities of Astro-WISE. We intend to prevent duplication of functionality of existing Astro-WISE classes,

though there is some overlap. In section 3.8 we discuss how the integration between the Source Collections and other Astro-WISE classes and functionality can be improved.

Improvement: Persistent Sample Selections

The Source Collections offer a mechanism to persistently store selections of sources and attributes and discover these catalogs through data pulling (section 3.5.4). Without the Source Collections, selections of sources and attributes had to be created by copying the required part of a SourceList. This had several problems:

- The selection criterion was not stored.
- The only link to the original SourceList was given by the `name` property of the new SourceList.
- Parts of the SourceList had to be duplicated, which did not scale to large SourceLists.

Improvement: Derived Attributes

The Source Collections offer functionality to create catalogs with attributes that are derived from attributes in existing catalogs (section 3.6). This can be done through data pulling and with full data lineage. Without Source Collections it was only possible to store such derived attributes in a new, independent, SourceList. This had several problems:

- There was no formal dependency link between such a new SourceList and the original SourceList. To some extent this could be alleviated by creating an AssociateList between the SourceLists.
- There was no data lineage describing how the new attributes were calculated. Therefore it was not possible to inspect or reprocess these SourceLists and such catalogs could not be pulled.
- Combining attributes stored in different SourceLists had to be done through an AssociateList or CombinedList, which was not scalable.

Improvement: Implicit Partial Processing

The Source Collections are Process Targets that do not have to be processed to be stored. That is, a Source Collection object can be made persistent without requiring all the catalog data it represents to be created (sections 3.3.2, 3.5.3). There are two main benefits of this approach.

- Source Collections that only require restructuring of existing catalog data can be processed on demand.

- Source Collections that require creation of new catalog data —Attribute Calculators in particular— are created as general as applicable. The processing of these Source Collections is limited to the necessary subsets that are required to process the final requested target Source Collection. This facilitates sharing and generality of Source Collections without sacrificing performance and scalability.

The partial processing itself is done through optimization of the dependency graph of a target Source Collection (sections 2.4.7, 3.5.2), e.g. by placing selections of sources before calculations of attributes. The dependency graph will contain temporary transient copies of the original Source Collections as a result of this reorganization. These smaller copies are subsequently processed completely. The resulting catalog data will be appended to the already stored catalog data of the original, if necessary. Therefore the original Source Collection is processed partially, albeit the processing itself is always done on an entire Source Collection. This means that the Source Collection instances themselves only require localized knowledge in order to process themselves.

Improvement: Separation of Processing and Storing

The Source Collection classes work similar to the Frame classes in the sense that processing and storing the processing result are separate actions (sections 3.5.3, 3.3.2). The reason for this is threefold:

- The processing result of Source Collections that can be processed on the fly does not have to be stored at all.
- Storing catalog data is a time consuming process, hindering real time interactive visualization, and is therefore delayed.
- The Source Collections are is designed to allow exploration of selection criteria and attribute calculation methods. During such an exploration phase it is not necessary to store any results, because they might be discarded.

Improvement: Catalog Data in the `awe`-prompt

The Source Collections allow catalog data—a collection of sources with associated attribute values—to be handled efficiently on the `awe`-prompt as arrays from the Python numpy package. The ultimate goal is to achieve seamless integration between the `awe`-prompt the database and external visualization software.

Improvement: Pulling of Catalog Data and Interactive Visualization

With the Source Collections it is possible to pull catalog data through query driven visualization (chapter 5).

3.1.4 Source Collection Overview

The Astro-WISE implementation of the Source Collection concept is split up in several classes which are summarized here. A UML diagram is given in figure 3.1. The classes

are named without spaces in the actual implementation, e.g. SourceCollection, SelectAttributes. Spaces are added in this chapter to aid reading, e.g. Source Collection, Select Attributes.

New Classes: Source Collections

Every operation on catalog data—described in section 2.5—has a corresponding Process Target class, derived from the base Source Collection. There is one base Source Collection class contains all the shared code and is described in section 3.3. The derived classes are discussed in section 3.4.

New Classes: Attribute Calculator Definition and Parameters

The Attribute Calculator (sections 2.5.10, 3.6) is a Source Collection class for the derivation of new attributes from existing attributes. The methods applied by Attribute Calculators are described with the persistent Attribute Calculator Definition class (section 3.6). Process parameters of Attribute Calculator Source Collections are stored with instances of Attribute Calculator Parameter classes.

New Classes: Source Collection Tree

The Source Collection Tree (section 3.5) is a non-persistent class that handles dependency graphs of Source Collections. It contains most of the logic that is attributed to "the information system" in chapter 2 and is responsible for modifications of the dependency graphs such as optimizations and data pulling.

New Classes: Set Relations and Set Relations Set

The Set Relations and Set Relations Set (section 3.7) are two non-persistent classes to determine the logical relations between sets of sources represented by the Source Collections. They are the implementation of the algorithm in section 2.6.

New Classes: Table Converter

The Table Converter (section 3.3.4) is a non-persistent class that is used to hold and modify tabular data in Python. In the context of Source Collections it is used to process Source Collections in Python and for interaction through the Simple Application Messaging Protocol (SAMP).

3.2 Existing Classes

We describe existing Astro-WISE classes that are related to the Source Collection classes. Only the most relevant functions and properties of the classes are given.

3.2.1 Astro-WISE Class: DBObject

The *DBObject* is the base class of all persistent objects in Astro-WISE.

Persistent Properties

There is one persistent property that all classes share:

- `object_id`: A machine readable identifier of the object. It is not necessary to refer to this property manually. With respect to the Source Collections it is only used explicitly for the Attribute Calculator (section 3.6).

Functions

The DBObject class contains the functionality to store objects persistently:

- `commit()`: Makes a transient object persistent. All objects it depends upon are made persistent as well.

3.2.2 Astro-WISE Class: DataObject

The *DataObject* class is derived from DBObject and is the base class for persistent classes of which the instances have an associated file on the dataserver.

Persistent Properties

- `filename`: The name of the file as it is stored on the dataserver.

3.2.3 Astro-WISE Class: ProcessTarget

The *ProcessTarget* class encapsulates the notion of a make-able object. The Process Targets have full data lineage: their persistent properties contain all information required to process the object. A ProcessTarget object has persistent properties that refer to other persistent objects that are required to for processing the Process Target. These objects are the *progenitors* of the ProcessTarget and the process of linking to the progenitors is called *backward chaining*. The progenitors of a ProcessTarget have to be made, before the ProcessTarget itself can be made.

Persistent Properties

- `creation_date`: The date the object is created.
- `is_valid`: A flag to indicate whether the object is still considered valid.

Functions

- `make()`: Process the object to produces all data and computes all values associated with the object from its progenitors.
- `is_made()`: Returns `True` if the object has been made.

3.2.4 Astro-WISE Class: SourceList

Catalogs created from images are stored in Astro-WISE as a *SourceList* object. Every SourceList is identified by its `SLID`, and every source in the SourceList by its `SID`. Usually a SourceLists has a Frame object as dependency, but they can also be ingested 'stand-alone'. SourceLists can also be created with the CombinedList class (section 3.2.6).

In this thesis we often regard the sources conceptually as objects themselves, even though there is no persistent *Source* class. Sources have attributes, which are parametrized properties of the sources. The values of the attributes in a SourceList are created by running SExtractor on the corresponding Frame. The sources are stored in specific database tables such as `SOURCELIST*SOURCES`, where every row corresponds to a sources and every column to an attribute.

The Source Collection classes can be seen as a generalization of the SourceList class. It would be possible to integrate the two classes, which would make the SourceList a specific Source Collection (section 3.4.1). The catalog data of a Source Collection can be stored using special SourceLists (section 3.3.3). These SourceLists should only be seen as a mechanism to store the result of processing a Source Collection, not as the result itself.

Persistent Properties

- `SLID`: Unique integer identifying the SourceList.
- `name`: A descriptive human readable name of the SourceList.
- `number_of_sources`: The number of sources.
- `sources`: A link to the sources.
- `frame`: The Frame object that this SourceList corresponds to, if any.
- `sexconf`: SExtractor configuration parameters.
- `sexparam`: Additional extraction parameters.

3.2.5 Astro-WISE Class: AssociateList

Sources in SourceLists can be associated with one another through *AssociateLists*. An AssociateList only contains information on the identification between sources, it does not represent any attributes. In general, AssociateLists are created from sky associations on two or more input SourceLists, but they can also be created manually. Every AssociateList is identified by its `ALID`, and every association in the AssociateList by its `AID`.

Persistent Properties

- `ALID`: Unique integer identifying the AssociateList.
- `number_of_associates`: The number of associations
- `associates`: A link to the associations.
- `sourcelists`: The SourceLists that are used as input to the AssociateList.
- `inputassociatelist`: Optionally, another AssociateList used to create this AssociateList.
- `associatelisttype`: The type of the AssociateList.
- `process_params`: Parameters used for the creation of the AssociateList.

3.2.6 Astro-WISE Class: CombinedList

The *CombinedList* class is used to create a SourceList from an AssociateList. For example to combine SourceLists of different filters into one multicolor SourceList or to 'stitch' several SourceLists that each cover a part of the sky into one larger SourceList. The `AID` of the associates of the input AssociateList will become the `SID` of the sources in the new SourceList.

The attributes of the sources in the input SourceLists are either copied directly to the sources in the new SourceList or are averaged to a single attribute first. The default for attributes representing a magnitude is to average magnitudes from the same filter and to keep magnitudes from different filters separate.

There is some overlap in functionality between the CombinedList and the Source Collection classes (e.g. see section 3.4.4). Conceptually it might be possible to convert the CombinedList class into a Source Collection class entirely, ensuring it becomes a proper persistent class.

Persistent Properties

The CombinedList class creates a new SourceList, it is not a persistent class itself. Some persistent properties are added to the SourceList for the benefit of the CombinedList:

- `associatelist`: `ALID` of the input AssociateList.

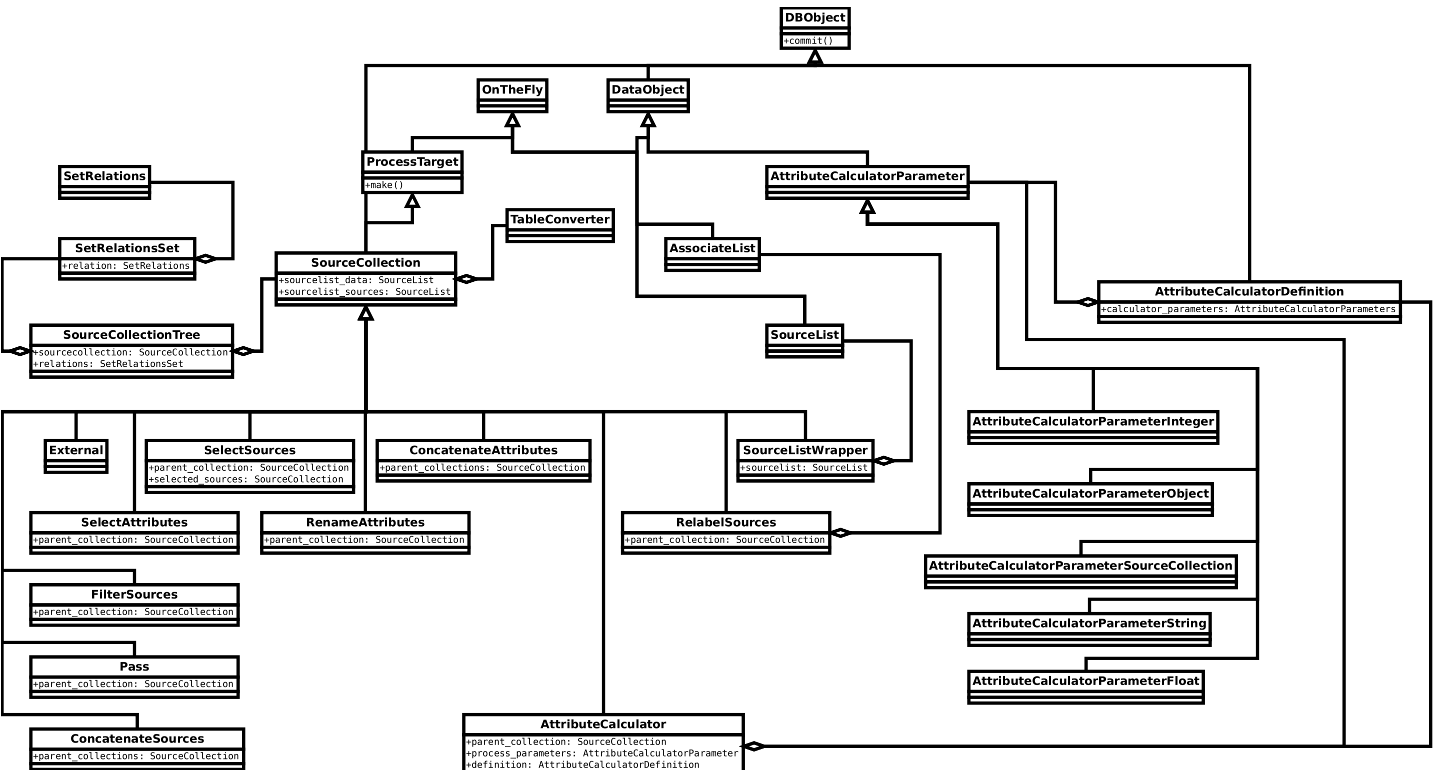

Figure 3.1: Astro-WISE classes discussed in this chapter. For example, a Select Sources is a persistent class that is derived from the Source Collection class and has a (persistent) property referring to another Source Collection.

3.3 Source Collection Class

The Source Collection is derived from DBObject and ProcessTarget, and can be seen as the catalog counterpart of what the BaseFrame class is for frames: Elementary operations on catalogs (section 2.5) are implemented as separate Process Targets derived from this base Source Collection class (section 3.4). The Source Collection class has a virtual methods for `make()`, `is_made()`, etc., which have to be provided by the derived classes.

The Source Collection class extends the principles of the ProcessTarget, and therefore can be used differently than other process targets. In particular, it is not required to process a Source Collection instance to make it persistent (section 3.3.2). Furthermore the result of the processing is only stored in the database if the Source Collection cannot be reprocessed on the fly.

First we explain how a Source Collection is defined and what catalog data it represents, then we describe the main functions to create and interact with Source Collections. This is followed by a more in dept discussion about how data is handled on the SQL and Python layers and we end with a short example.

3.3.1 Source Collection Definition

We describe how a Source Collection object is defined and how the catalog data is represented (section 2.4).

Definition: Persistent Properties

The base Source Collection class contains several persistent properties. Only the `SCID` is part of the definition of the object:

- `SCID`: The unique identifier of a Source Collection, generated from the `SourceCollection` database sequence.

  A unique identifier is required for transient Source Collections as well, to facilitate interaction through data pulling. However, the round trip to the database to access the sequence is too time consuming during the dependency graph optimization. Therefore transient Source Collections can also be identified through a negative integer that is only unique within the session.

  The `get_scid()` function will return the definitive positive identifier if one has been generated and the negative temporary identifier otherwise. Accessing the `SCID` directly ensures that a globally unique identifier is generated by the sequence. This ensures that a positive `SCID` is set when the Source Collection is made persistent.

The Source Collection objects also has descriptive properties, such as:

- `name`: A name of the Source Collection.
- `creation_date`: The creation date of the Source Collection.

Furthermore a Source Collection class has persistent properties that are used for caching the processing results, these are described in section 3.3.3. Other persistent properties, such as the required progenitors and process parameters are defined in the derived classes (section 3.4).

Definition: Sources

The sources of a Source Collection are identified by their `SLID-SID` combination (section 2.4.3). The set of sources is defined implicitly by the operator and the progenitors of the Source Collection. The identifiers of the sources can be cached in the database for performance reasons (section 3.3.2). There are several functions to retrieve information about the sources in a Source Collection:

- `load_sources()`: Loads the source identifiers into Python (see section 3.3.2).
- `store_sources()`: Stores the source identifiers in the database (section 3.3.2).
- `get_source_relations()`: Creates a Set Relations Set object that is used to determine the logical relations between the set of sources represented by this Source Collection and those earlier in the dependency graph (see section 3.7).

Definition: Attributes

The attributes of the sources represented by a Source Collection, are handled as is customary in Astro-WISE: They are identified by a character string (section 2.4.4) and it is assumed that the units and unified content descriptors are understood. Attributes with the same name represent the same physical property, although they might been derived in different ways. The set of attributes a Source Collection represents only has to be stored if the catalog data itself is stored.

The `SLID` and `SID` can be seen as 'special' attributes: they represent the sources themselves, not quantified properties of the sources. A Source Collection always contains the `SLID` and `SID`, even if they are not mentioned explicitly (e.g. by a Select Attributes). There are several functions to retrieve information about the attributes in a Source Collection:

- `get_attributes()`: Creates a list of the attributes in the Source Collection. The elements of the list are dictionaries describing the attributes, which can be used for the `attributes` property of a Table Converter object (see section 3.3.4).
- `get_attribute_names()`: Returns a list of attribute names only.

Definition: Class or Operator

A Source Collection is processed by performing an operation on its progenitors. Every Source Collection subclass represents a specific operator to create a catalog of sources, usually from other catalogs. Besides the catalog data itself, a Source Collection can also be seen as the representation of this operation. This turns the Object-Oriented perspective on Target Processing into a Functional one.

The implemented classes are summarized in table 3.1, and elaborated on in section 3.4. Some Source Collection can be processed on the database in SQL (section 3.3.3) and some on the `awe`-prompt in Python (section 3.3.4) and some on both. Processing Source Collections on the database is done by formulating the operation it represents as an SQL query. Multiple of these operations can be combined into one query.

Operator	SQL	Python	Description
SourceList Wrapper	no	no	Wrapper around the SourceList
External	no	no	Catalogs with no data lineage
Select Attributes	yes	yes	Selects a subset of attributes
Rename Attributes	yes	yes	Renames attributes
Concatenate Attributes	yes	yes	Groups attributes
Attribute Calculator	some	some	Calculates new attributes
Filter Sources	yes	some	Applies selection criterion
Select Sources	yes	yes	Selects specific sources
Concatenate Sources	yes	yes	Groups sources
Relabel Sources	yes	no	Relabels sources
Pass	yes	yes	No operation

Table 3.1: Different Source Collection classes implemented within Astro-WISE. Each class corresponds to a specific operation to create catalog data. Some Source Collection can be processed on the database, some in Python. In section 3.8 we describe possible improvements to increase the number of 'yes' cells in this table.

3.3.2 Usage of Source Collections

We give a description on how Source Collection objects can be created and how the catalog data is handled. This section explains how the Source Collections work; for a more user oriented hands-on approach see the online documentation.

Usage: Creating and Committing Source Collections

There are three ways to create Source Collections:

- Manually by instantiating the class of the required operation and setting the progenitors and process parameters.
- Automatically using the data pulling functions from the Source Collection Tree.
- Implicitly as temporary transient copies when using a Source Collection Tree to process Source Collections.

Functions for committing Source Collections:

- `commit()`: Stores the Source Collection persistently. It is not required that the Source Collection has been processed. That is, its composition of sources and

values of the attributes is unambiguously defined, but might yet be undetermined. All dependencies are committed as well. Source Collections will get a globally unique `SCID` from the database if none has been set yet.

Usage: Processing Source Collections

The main functions to process a Source Collection and store the result are: `is_made()`, `make()`, `load_data()` and `store_data()`. The latter can be seen as the equivalent of `retrieve()` and `store()` of the Frame classes.

A design concept of the Source Collections is that it is not required to process a Source Collection in order to use it (section 2.3.1). In practical terms this means that it is not required to call `make()` before performing `commit()`. Some Source Collections do not require to be made at all (sections 2.4.8, 3.3.3) because they can be processed implicitly by calling `load_data()`.

The Source Collection classes work similar to their Frame counterpart, in the sense that processing and storing are separate actions. Calling `make()` does not store the result. The `store_data()` stores the catalog data persistently, and `load_data()` retrieves the catalog data to be used on the Python prompt.

The processing related functions can either be performed on an individual Source Collection or on its entire dependency graph. The latter is the default: a call of these functions on an individual Source Collection object will be deferred to a Source Collection Tree created from the object. This Source Collection Tree will perform the function in the most optimal way (section 3.5.3), which might require optimization of the depenency graph. This default is required, as explained for each function below.

Setting the `optimize` parameter of these functions to `False` will perform the required operation on the Source Collection itself; in principle only the Source Collection Tree should have to do this. The functions are described here as they function on individual Source Collection objects.

- `is_made()`: Returns `True` if it is not required to call `make()` to derive the catalog data of this Source Collection. That is, `True` is returned if the catalog data is available in Python, stored in the database, or can be derived on the fly by calling `load_data()` directly.

  Calling this function on an individual Source Collection should only be done by a Source Collection Tree because it might have some unintuitive results. This is due to the localized knowledge a Source Collection has.

  This function will always return `True` for Source Collections that can be processed on the fly. In practice, only some Attribute Calculators will have to be processed, so this function will return `True` for all other Source Collections. Furthermore, `is_made()` is not called recursively on the progenitors. Therefore it is possible that `True` is returned, even though it is not possible to retrieve the catalog data, because the progenitors have to be processed first.

  Vice versa, it is not always possible for a Source Collection itself to determine whether it has been made, because an individual instance does not necessarily

have knowledge about the exact composition of sources it represents. Therefore, `False` might be returned even though the Source Collection has been processed entirely.

Calling `is_made()` through a Source Collection Tree (the default) mitigates these two problems because it has overview of the entire dependency graph.

- `make()`: Process the Source Collection. In practice only Attribute Calculators have to be processed. However, future operators might require processing as well. The result of the processing —the catalog data— is not automatically stored in the database. The `load_data()` function below can be used directly to retrieve the catalog data of Source Collections that can be processed on the fly.

  It is required to have the (catalog) data of the progenitors loaded into Python in order to process a Source Collection, otherwise `False` is returned. Calling `make()` through a Source Collection Tree ensures that this is the case.

- `load_data()`: Loads the catalog data into a TableConverter object, either from the database or from the progenitors directly in Python. In essence, `load_data()` can be seen as the equivalent of `make()` for Source Collections that can be processed on the fly, or as the catalog counterpart of the `retrieve()` function of the Frame classes.

  Retrieving catalog data from the database utilizes the query generation function (section 3.3.3). Deriving the catalog data on Python uses the TableConverter class (section 3.3.4).

- `store_data()`: Stores the processing result —the catalog data— from Python into the database in the `sourcelist_data`[1].

Usage: Storing and Loading Sources

There are several functions defined to specifically handle the identifiers of the sources represented by a Source Collection. These functions can also be called on a Source Collection Tree or on an individual Source Collection, similarly to the above. The former is the default, since Source Collection instances do not necessarily have knowledge of the exact set of sources they represent. A Source Collection Tree can infer this composition from the backward chaining. Furthermore, the Source Collection Tree can optimize the dependency graph specific for the retrieval of the source identifiers.

- `load_sources()`: Load the source identifiers into Python.

- `store_sources()`: Store the source identifiers in the database in the `sourcelist_sources`[2].

[1] In the current implementation it is not yet possible to store catalog data directly from an SQL query into the database.

[2] In the current implementation the sources are always ingested from a FITS file. It is not yet possible to ingest the sources directly through SQL.

3.3.3 SQL Layer

Besides storing the objects themselves, the Astro-WISE database is also used to store the result of processing a Source Collection persistently, that is, the composition of sources and the values of the attributes. Furthermore most operators can be evaluated in SQL: all the restructuring operators and some of the Attribute Calculators.

SQL Layer: Data Caching

A design principle of the Source Collections is that they can be reprocessed from their data lineage (section 2.3.1). Therefore, storing the values of the attributes or the exact composition of sources should only be done for performance reasons and is thus seen as a form of caching (section 2.4.8). In our Astro-WISE implementation this results in the following guidelines:

- The catalog data of a Source Collection can be stored in the database through a SourceList[3], referenced through the `sourcelist_data` property (see below).
- Only SourceList Wrapper and External Source Collections must have their entire catalog stored in the database.
- Attribute Calculator Source Collections store the values of the attributes in the database if and only if the attributes cannot be calculated in SQL. Source Collections can be processed partially and only the processed results are stored.
- Instead of storing the values of the attributes, it is also possible to store only the source identifiers (`SLID-SID` combinations) in a SourceList, referenced through the `sourcelist_sources` property (see below). In practice this only used for Filter Sources Source Collections.
- A scientist can overrule these principles manually by explicitly turning of optimization when calling `store_data()` (see below).

In the current implementation nothing is stored without user interaction, primarily for performance reasons. Storing sources in accordance with the previous points is done by calling `store_data()` on a Source Collection Tree (section 3.5). In section 3.8.3 we discuss improving the storage mechanisms.

SQL Layer: Persistent Properties

The Source Collection class contains a set of persistent properties that are only used for caching. It should be possible to change these properties after a Source Collection has been committed because the decision to cache sources can be made after the Source Collection is made persistent[4]. This does not violate the persistent object

[3] These SourceLists use `SOURCECOLLECTIONDATA` as name to avoid confusion with SourceLists created from an image.

[4] In the current implementation it is not yet possible to change these properties for persistent Source Collections. Therefore the decision to store data or not should be made before the Source Collection is committed. For Attribute Calculators this is done on initialization.

philosophy: these properties only refer to the *result* of the processing, not to the processing itself. In fact, conceptually it is not even necessary to expose these properties to users of the object.

- `sourcelist_data`: An optional SourceList to store the catalog data of this Source Collection.

  This SourceList might contain only a subset of the sources, but has to have all the attributes for the sources that are stored. It is assumed that all processed sources should be stored if a `sourcelist_data` is set.

  A Source Collection that represents a subset of another Source Collection can use the `sourcelist_data` of the larger Source Collection to prevent duplication of catalog data. Therefore the `sourcelist_data` might contain more sources than are represented by the Source Collection.

  Most Source Collections represent sources that originate from another Source Collection. The `SLID` and `SID` of these sources will be stored in the `SLIDorg` and `SIDorg` columns of the `sourcelist_data`. The new, actual, `SID` of the sources is meaningless; it only serves to create a unique `SLID-SID` combination for the database. Using the original `SID` (now `SIDorg`) for this is not possible for two reasons: A Source Collection can contain a subset of sources of another Source Collection which would result in nonconsecutive `SID`s which is not allowed. Furthermore, a Source Collection can describe sources from multiple SourceLists, which might have identical `SID`s which would make it impossible to use it as a unique identifier within the Source Collection.

  The `SOURCELIST*SOURCES**05` database table is used for these SourceLists because this table has an index on the `SLIDorg-SIDorg` columns. This makes it possible to join the rows of the sources of different Source Collections in SQL, e.g. through a Concatenate Attributes operator.

- `sourcelist_sources`: An optional SourceList that holds only the source identifiers of this Source Collection. All sources, and no other sources, should be included in the `sourcelist_sources`. Any attributes in the `sourcelist_sources` are ignored.

  Different Source Collections can (or even should) share the same `sourcelist_sources`, if they represent the same set of sources. If the `sourcelist_sources` contain `SLIDorg-SIDorg` attributes, then these are the identifiers of the sources, otherwise the `SLID-SID` attributes are the identifiers. A way to ensure that the `sourcelist_data` data contains data for all the sources, is using the same SourceList for the `sourcelist_sources` and the `sourcelist_data`.

- `attribute_names`: An ordered list with names of the attributes of the Source Collection. This property is required when the `sourcelist_data` is set, but can also be used to cache the names of the attributes when the catalog data itself is not stored. The `SLID` and `SID` should not be included.

- `attribute_columns`: A list of column names of the `sourcelist_data` in which the attributes are stored. This property should only be set when the `sourcelist_data` is set as well. The order should be the same as in `attribute_names`.

  There are several reasons for having this property. Every SourceList in Astro-WISE has to have a specific set of columns (`RA`, `DEC`, `HTM`, `A`, `B`, `POSANG` and `FLAG`), which might not be required in the Source Collection. Furthermore, by allowing Source Collections to share their `sourcelist_data`, the `sourcelist_data` might contain other extra attributes as well.

  Also, the `attributes_store` abstracts away the database column from the actual physical parameter description. This prevents the requirement of a separate column per attribute by allowing generic columns to be used which are renamed on the fly.

  The position columns (`RA`, `DEC`, `HTM`) of the `sourcelist_data` and `sourcelist_sources` will always contain values that actually belong to the source—even if the Source Collection itself does not represent these attributes—because the database is partitioned and indexed on these columns.

- `all_data_stored`: A flag to indicate whether all the sources are stored. A 1 means that all attributes are stored for all sources. A 0, however, could also mean that all the data is stored, but this has not yet been determined. In both cases it might be that the `sourcelist_data` contains sources that are not part of this Source Collection.

SQL Layer: Functions

On the SQL layer, the action to process a Source Collection—also called its operator—can be expressed as an SQL query. Processing a Source Collection is equivalent to executing this query. Sequential operators can be combined into one query by using the SQL query of progenitor Source Collections as `WITH` statements in the query of the final Source Collection. We summarize the benefits of the operator-as-query approach:

- Many Source Collections operations translate naturally to SQL, especially those that select or combine sources and attributes.
- The calculation of some of the simpler Attribute Calculator Source Collections can be implemented in the SQL layer as well.
- The database can cache query results, such as the `WITH` sub-queries.
- The query can be parallelized with `PARALLEL` hints[5].

There are several Source Collection functions to create SQL queries. These functions require that no more catalog data has to be stored in the database in order to execute the query. `False` is returned if the Source Collection determines that the

[5] In the current implementation the `PARALLEL` hints are not yet used.

required data is not stored completely or when the query could not be generated for other reasons.

All required attributes are mentioned explicitly in the queries. If the source identifiers are stored in the `SLIDorg-SIDorg` columns then they are renamed to the original `SLID-SID` to be joined upon. In principle these functions should not be called directly, but through other functions such as `load_data`.

- `load_data_sql()`: Loads the catalog data from the database into Python as a Table Converter object. This function calls `get_query_full()` and is itself called by `load_data()`.
- `get_query_full()`: Creates the full query that can be sent to the database to retrieve the catalog data. This function calls `get_query_self()` and `get_query_with_clauses()`.
- `get_query_self()`: Creates the query for this specific operator. The parent Source Collections are referred to by the alias that corresponds to their `WITH` clause.
- `get_query_with_clauses()`: Creates a list of queries of the parents of this Source Collection, which are used as `WITH` clauses in the final query. This function calls `get_query_self()`, `get_query_with_clauses()` and `get_query_alias()` on the progenitors.
- `get_query_alias()`: Returns the alias that should be given to the `WITH` clause.
- `create_empty_sourcelist_data()`: Creates a new `sourcelist_data` to store the catalog data in.

3.3.4 Python Layer

On the Python layer, the sources of a Source Collection and their attributes are held in Table Converter objects. The result of processing a Source Collection with `make()` or retrieving catalog data with `load_data()` is kept in a Table Converter.

Catalog data should only be kept on the Python layer temporarily and should be stored in the database to become persistent. It is assumed that only relatively small datasets are kept on the Python layer and that it is not problematic to keep multiple copies of the same data.

Python Layer: Holding Catalog Data

A Source Collection object has three optional non-persistent properties to hold catalog data on the Python layer:

- `data`: A Table Converter that contains all attribute values for the exact set of sources of this Source Collection.
- `datasuper`: A Table Converter that contains attribute values for a superset or subset of the sources of the Source Collection.

- `sources`[6]: A Table Converter that contains the exact set of source identifiers of the Source Collection. That is, it contains only the `SLID` and `SID` attributes.

Source Collections can share the same `datasuper` if they represent subsets of the same larger set of sources, but only the `data` or `sources` if they represent the exact same set of sources. The `datasuper` is automatically shared between related Source Collections when a Source Collection Tree is optimized before processing (section 3.5.2). This ensures that Source Collections can be processed in parts without duplication of catalog data.

Python Layer: Table Converter Class

The Table Converter is a non-persistent class that is used to hold and manipulate catalog data in an efficient way within Python. Catalogs can also be sent through SAMP or stored as a FITS file or VOTable with a Table Converter object. The most important properties are:

- `attribute_order`: A list which contains the names of the attributes in a preferred order.
- `attributes`: A dictionary with information about the attributes.
- `data`: A dictionary with as keys the names of the attributes and as values Python numpy arrays containing the values of the attributes.

The `attributes` property is a dictionary where the keys are the attribute names and the values another dictionary describing the attribute. These latter dictionaries contain the following keys:

- `name`: The name of the attribute.
- `format`: A string representing the format of the attribute.
- `ucd`: The Unified Content Descriptor of the attribute[7]
- `null`: Which value is used to represent NULL values. This is necessary for formats that have no natural NULL values, such as integers.
- `length`: The length of a table cell. Mainly applicable to strings, because Astro-WISE does not support multi-element cells.

The Table Converter class can be used to export and import catalog data with the following functions:

- `save_fits()`: Stores the catalog in a FITS file, often used by Attribute Calculators to process the catalog with external software (section 3.6).

[6] The name of the `sources` property conflicts with the SourceList class. Therefore it might need to be changed for consistency in the future.

[7] The `ucd` and `null` values of an attribute are not always used in the current implementation of the Source Collection.

- `save_votable()`: Stores the catalog in a VOTable file, used when sending catalog data over SAMP (chapter 5).
- `load_fits()`: Loads a FITS file.
- `load_votable()`: Loads a VOTable.

Python Layer: Catalog Data Functions

- `load_data_python()`: Process a Source Collection with restructuring operators in Python. That is, all operators that do not create new catalog data, such as Select Attributes, Concatenate Sources. This is usefull for small datasets that have to be manipulated on the fly, e.g. for real time visualization, because for such datasets the round trip to the database with `load_data_sql()` takes too much time. This function should be called through `load_data()`.
- `make()`: This function is already described in section 3.3.2. We list it here because `make()` only works in Python. In practise, only certain Attribute Calculators (section 3.6) require to be made, in section 3.8 we describe the design of other operators that require to be made. This function might need to call external programs to create the attributes.

The difference between `load_data_python()` and `make()` is that the former assembles existing catalog data, while `make()` derives new attribute values that probably will be stored in the database.

3.3.5 Example Dependency Graph and SQL

Figure 3.2 shows an example of a dependency graph that can be processed on the database entirely. The generated SQL query is given below, slightly simplified and formatted. The Attribute Calculator Source Collection does not store its attributes in the database because it can be processed on demand.

```
WITH COLLECTION100511 AS (
  SELECT
    COLLECTION100511."SLID" AS "SLID",
    COLLECTION100511."SID" AS "SID",
    COLLECTION100511."RA" AS "RA",
    COLLECTION100511."DEC" AS "DEC",
    COLLECTION100511."SDSS_petroR90_g" AS "SDSS_petroR90_g",
    COLLECTION100511."SDSS_petroR50_g" AS "SDSS_petroR50_g",
    COLLECTION100511."SDSS_petroR90_i" AS "SDSS_petroR90_i",
    COLLECTION100511."SDSS_petroR50_i" AS "SDSS_petroR50_i",
    [...]
  FROM
    AWOPER."SOURCELIST*SOURCES**04" COLLECTION100511
  WHERE
```

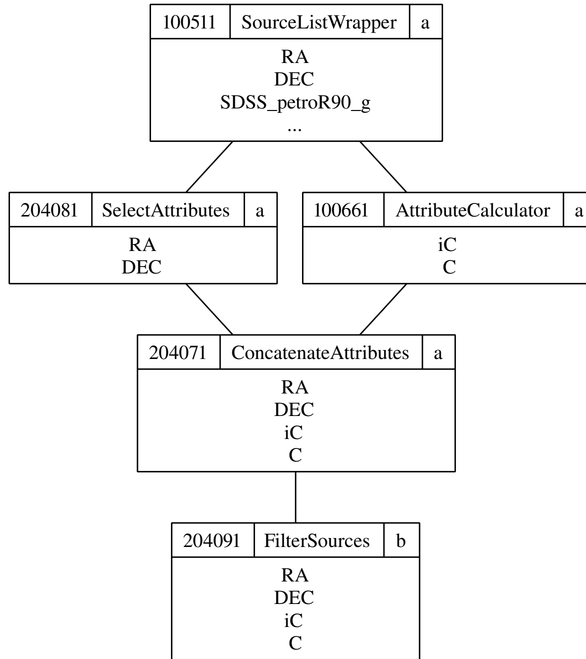

Figure 3.2: Example dependency graph of Source Collections, see also figure 2.2. The top left number of every node is the `SCID`. The top center string is the operator. The top right symbol represents the set of sources, equal letters mean equal sources. The area below the node is used to show the attributes. The process parameters, such as the selection criterion used by the Filter Sources, are not shown.

```
    "SLID" = 1575051
),
COLLECTION204081 AS (
  SELECT
    COLLECTION100511."SLID" AS "SLID",
    COLLECTION100511."SID" AS "SID",
    COLLECTION100511."RA" AS "RA",
    COLLECTION100511."DEC" AS "DEC"
  FROM
    COLLECTION100511
),
COLLECTION100661 AS (
  SELECT
    COLLECTION100511."SLID" AS "SLID",
    COLLECTION100511."SID" AS "SID",
    ( ( (COLLECTION100511."SDSS_petroR50_g" /
         COLLECTION100511."SDSS_petroR90_g")  +
```

```
      (COLLECTION100511."SDSS_petroR50_i" /
       COLLECTION100511."SDSS_petroR90_i") ) / 2 ) AS "iC",
    ( 2 /( (COLLECTION100511."SDSS_petroR50_g" /
             COLLECTION100511."SDSS_petroR90_g")  +
      (COLLECTION100511."SDSS_petroR50_i" /
       COLLECTION100511."SDSS_petroR90_i") ) ) AS "C"
  FROM
    COLLECTION100511
),
COLLECTION204071 AS (
  SELECT
    COLLECTION204081."SLID" AS "SLID",
    COLLECTION204081."SID" AS "SID",
    COLLECTION204081."RA" AS "RA",
    COLLECTION204081."DEC" AS "DEC",
    COLLECTION100661."iC" AS "iC",
    COLLECTION100661."C" AS "C"
  FROM
    COLLECTION204081,
    COLLECTION100661
  WHERE
    (COLLECTION204081.SLID = COLLECTION100661.SLID) AND
    (COLLECTION204081.SID = COLLECTION100661.SID)
 )
SELECT
  COLLECTION204071."SLID" AS "SLID",
  COLLECTION204071."SID" AS "SID",
  COLLECTION204071."RA" AS "RA",
  COLLECTION204071."DEC" AS "DEC",
  COLLECTION204071."iC" AS "iC",
  COLLECTION204071."C" AS "C"
FROM
  COLLECTION204071
WHERE
  "iC" < 0.5
```

3.4 Source Collection Classes

The Source Collection class should be seen as a base class, similar to the Frame class. The operators of section 2.5 are implemented in separate classes derived from this base class. These derived classes prescribe how to process an instance of the class. These define the required process parameters and progenitors and several functions, in particular `make()`, `get_query_self()` and `load_data_python()`.

The Astro-WISE implementation has slightly different operators in order to inter-

face with existing functionality. For each class we describe its persistent properties and most important functions[8]. We refer to section 2.5 for a conceptual description of the operators, dependency graph modifications and examples.

3.4.1 Operator: SourceList Wrapper

A *SourceList Wrapper* Source Collection represents sources and attributes that are resp. detected and measured on an image (sections 2.5.1, 2.5.2). It is a wrapper around the SourceList class and in principle there should be a one to one correspondence between a SourceList and a SourceList Wrapper Source Collection.

The SourceList Wrapper Source Collection cannot actually run the SourceList code. In section 3.8 we discuss how the integration between the SourceList Wrapper and the SourceList can be improved.

Persistent Properties

- `sourcelist`: The SourceList that corresponds to this Source Collection.

Functions

- `set_sourcelist()`: Sets the `input_sourcelist`, the `sourcelist_data`, `attribute_names` and `attribute_columns`.

3.4.2 Operator: Select Attributes

A *Select Attributes* Source Collection is used to select a subset of attributes from another Source Collection (section 2.5.7).

Persistent Properties

- `parent_collection`: One parent Source Collection.
- `selected_attributes`: A list of selected attribute names.

3.4.3 Operator: Rename Attributes

A *Rename Attributes* Source Collection is used to rename attributes from another Source Collection (section 2.5.8).

Persistent Properties

- `parent_collection`: One parent Source Collection.
- `attributes_old`: A list of old attribute names.

[8]Process parameters of Source Collections are not stored in a separate *Parameters* object, but are persistent properties of the respective classes directly.

- `attributes_new`: A list of new attribute names.

Attributes that are not renamed, keep their original name. The SLID and SID attributes should not be renamed.

3.4.4 Operator: Concatenate Attributes

A *Concatenate Attributes* Source Collection is used to combine different attributes from the parent Source Collections into one (section 2.5.9). On the SQL level this is done with a JOIN clause on the SLIDs and SIDs of the sources.

Persistent Properties

- `parent_collections`: A list of one or more parent Source Collections.

3.4.5 Operator: Filter Sources

A *Filter Sources* Source Collection is used to select sources from one parent Source Collection through the evaluation of a logical expression (section 2.5.3).

Persistent Properties

- `parent_collection`: One parent Source Collection.
- `query`: A selection criterion to specify the required sources. Attributes used in the `query` should be enclosed in double quotation marks, e.g. `"redshift" < 0.1`. The query should be set through the `set_query()` function, which ensures that the query can be evaluated.

Functions

- `set_query()`: A function to set the selection query used in the Filter Sources.

The `query` is used as the WHERE clause in `get_query_self()` to evaluate the operator on the database. The Table Converter class is able to parse a limited set of queries to evaluate the operator in Python. The following query components can be used in Python: =, <, <=, >, >=, NOT, NULL, IS, AND, OR, BETWEEN, *, /, +, -, (and).

The attributes referenced in the query should be represented by the parent of the Filter Sources. The data pulling functions of the Source Collection Tree (sections 2.4.6, 3.5.4) ensure this automatically when creating Filter Sources Source Collections.

3.4.6 Operator: Select Sources

A *Select Sources* Source Collection is used to select set of sources from the parent Source Collection by listing their identifiers explicitly (section 2.5.4). In SQL this is done with a JOIN clause on the SLIDs and SIDs.

Persistent Properties

- `parent_collection`: One parent Source Collection.
- `selected_sources`: Another Source Collection that represents the selected sources. Any attributes in this Source Collection are ignored.

3.4.7 Operator: Concatenate Sources

A Concatenate Sources Source Collection concatenates the sources of several parents Source Collections (section 2.5.5). On the SQL level this is done through a `UNION ALL` clause.

Persistent Properties

- `parent_collections`: A list of one or more parent Source Collections.

3.4.8 Operator: Relabel Sources

A *Relabel Sources* Source Collection is used to relabel the sources of one parent Source Collection (section 2.5.6). The identification of sources is given by an input AssociateList.

The input AssociateList has to be a master AssociateLists with two input SourceLists[9]. Every source should appear a maximum of one time in the associates, because the operator requires an unambiguous remapping of the sources. The same AssociateList can be used in several Relabel Sources Source Collections.

For every source in the parent Source Collection, the Relabel Sources operator looks up the association that corresponds to the source and subsequently substitutes the original `SLID-SID` combination with those of the other source in the association.

Note that the parent Source Collection and the `selected_sources` do not have to be SourceList Wrapper Source Collections. They could be any Source Collection, as long as all the sources have the same `SLID`. In section 3.8 we discuss how the integration between the Relabel Sources and the AssociateList can be improved.

Persistent Properties

- `parent_collection`: One parent Source Collection.
- `associatelist`: The AssociateList that contains the identification.

3.4.9 Operator: Attribute Calculator

A *Attribute Calculator* Source Collection is used to calculate new attributes from existing attributes (section 2.5.10). The actual method to calculate the attributes is decoupled from the Source Collection and stored as a Attribute Calculator Definition

[9] In the current implementation there are some mechanisms to allow for other AssociateLists.

object. A Attribute Calculator contains a reference to such a definition and includes the process parameters used in this particular calculation. The Attribute Calculator and related classes are described in detail in section 3.6.

An Attribute Calculator object represents the new attributes for the same set of sources as its parent. The parent Source Collection should represent exactly the attributes required by the Attribute Calculator Definition.

Persistent Properties

- `parent_collection`: One parent Source Collection.
- `definition`: The Attribute Calculator Definition object that is responsible for the actual calculation (section 3.6.1).
- `process_parameters`: A list of Attribute Calculator Parameter objects that are used in the calculation (section 3.6.2).

3.4.10 Operator: External

An *External* Source Collection represents attributes of sources that have no data lineage (section 2.5.11). SourceLists that have no corresponding frame should be wrapped with an External instead of with a SourceList Wrapper. An External is also used as a transient object during optimization of a Source Collection Tree (section 3.5.2).

Persistent Properties

- `origin`: A string to describe the origin of the catalog data.

Functions

- `set_sourcelist()`: Sets the `sourcelist_data`, `attribute_names` and `attribute_columns`.

3.4.11 Operator: Pass

A *Pass* Source Collection represents exactly the same sources and attributes as its parent (section 2.5.12). A Pass Source Collection should only be used internally by a Source Collection Tree.

Persistent Properties

- `parent_collection`: One parent Source Collection.

3.5 Source Collection Tree

A *Source Collection Tree* is a non-persistent object that manages a dependency graph of Source Collections[10] It is the core of the data pulling mechanisms and fulfills some of the roles the Target Processors has for other Process Target classes. Most of the logic that is attributed to the "information system" in chapter 2, is implemented in the Source Collection Tree. A Source Collection Tree object is responsible for:

- Governing the processing of Source Collections and the caching of catalog data.
- Modifying dependency graphs of Source Collections, e.g. for scalability during processing and interaction.
- Finding existing Source Collections and creating new ones when pulling data.

This chapter describes how the Source Collection Tree performs these tasks, see the online manual for a more hands on approach.

3.5.1 Initializing a Source Collection Tree

A Source Collection Tree is initialized from a Source Collection and there are three ways to do so:

- Manually from the `awe`-prompt by using the Source Collection as argument in the constructor of the Source Collection Tree.
- By calling certain functions of a Source Collection with optimization enabled, e.g. `load_data()`.
- Through data pulling, e.g. through query driven visualization over the Simple Application Messaging Protocol (chapter 5).

The Source Collection Tree creates a new Pass Source Collection with the given Source Collection as the parent. This is because the end node of the Source Collection Tree should be fixed, but some actions of the Source Collection Tree require nodes to be replaced.

3.5.2 Dependency Graph Modifications

One of the main functions of the Source Collection Tree is modifying dependency graphs of Source Collections. There are two ways the dependency graph is modified:

- Replacing a progenitor of a Source Collection with another Source Collection that represents the exact same catalog. This is the only mechanism that is applied when the dependency graph is being optimized for processing (section 2.4.7), and the main mechanism in the construction of the dependency graph during data pulling (section 2.4.6).

[10] The Source Collection Tree class is named the way it is due to legacy reasons, even though the dependency graphs it manages are not actual *trees*.

- Replacing a progenitor of a Source Collection with a Source Collection that represents a different catalog than the original. This mechanism is only used during the construction of dependency graphs and only on Pass Source Collections that are themselves not used as a progenitor (section 2.4.6).

In most cases the Source Collection Tree delegates the actual creation of modified Source Collection copies to the Source Collection instances themselves. The modifications that are possible for a each Source Collection class are discussed in section 2.5. Algorithms for the dependency graph modifications are given in section 2.4.

Dependency Graph Modification Functions: Source Collections

The base Source Collection class has a set of virtual methods for dependency graph modifications that are overloaded by the derived classes. These functions do not modify the Source Collection itself, but return a new Source Collection that meets the required modification.

- `copy()`: Creates a transient copy of this Source Collection.

- `modify_remove_dependencies()`: This function is used to simplify the dependency graph before processing by removing dependencies that are not required to process this Source Collection. For example, Source Collections that are already processed are replaced by an External Source Collection that represents the exact same catalog.

- `modify_integrate_parents()`: This function tries to create a new Source Collection that combines the operator of this Source Collection with one of its parents. For example two subsequent Concatenate Attributes can be replaced with one.

- `modify_remove_self()`: This function returns the parent of this Source Collection if this Source Collection is essentially a Pass Source Collection, e.g. a Select Attributes that selects all attributes.

- `modify_move_up()`: This function tries to move the Source Collection up in the dependency graph. In the most simplistic way this is done by creating a copy of the Source Collection and its parent and switching their parents.

Dependency Graph Modification Functions: Source Collection Tree

The Source Collection Tree class contains several functions to manipulate the dependency graph, we list the important ones:

- `copy_tree()`: Creates a copy of the entire dependency graph. This is useful when the dependency graph has to be optimized for processing. The copied Source Collections can usually be discarded after it fulfilled its purpose.

- `optimize_for_load()`: Optimizes the dependency graph to limit processing to subsets required to create the catalog data of the Source Collection at the end of the graph. This calls several of the other modification functions.

- `simplify_tree()`: Simplifies the dependency graph without reordering the Source Collections. This function might modify the dependency graph in a conceptual way and should not be used on Source Collection Trees with Source Collections that will be made persistent.

- `simplify_attributes()`: Simplifies the dependency graph in a way that can be used on Source Collection Trees that need to be made persistent. For example, when creating a dependency graph through data pulling (section 3.5.4).

- `move_selectattributes_up()`: Moves a Select Attributes Source Collection up the dependency graph.

- `move_selectsources_up()`: Moves a Select Sources Source Collection up in the dependency graph.

- `move_selectsources_down()`: Moves a Select Sources down the dependency graph by moving a Source Collection with a Select Sources as parent up.

- `selectsources_from_filtersources()`: Converts a Filter Sources Source Collection into an Select Sources.

3.5.3 Processing Source Collections

The Source Collection Tree has the same `is_made()`, `make()`, `load_data()` and `store_data()` functions as individual Source Collections, which work on the entire dependency graph.

A call to these functions on an individual Source Collection instance will be deferred to a Source Collection Tree unless the `optimize` parameter is explicitly set to `False`[11]. This will ensure that the function is performed in the most optimal way, without requiring the user to interface directly with a Source Collection Tree.

- `is_made()`: Optimizes the dependency graph and recursively calls `is_made()` on each node in the graph. Furthermore, the Source Collection Tree checks whether partially processed Source Collections are actually processed completely. Source Collection instances cannot always do this themselves, since they have no direct knowledge of the composition of sources that they represent. Returns `True` if all the Source Collections in the dependency graph are made or can be processed on the fly, `False` otherwise.

[11] In the current implementation, the call to the dependency graph optimization routine is performed by the Source Collection instance, not by the Source Collection Tree itself. This should be changed to match the description here.

- `make()`: Optimizes the dependency graph and traverses the graph to recursively call `make()` on the Source Collections for which `is_made()` returns `False`. The Source Collection Tree will assure that the data of the parents of the Source Collection is available in the database or on Python, depending on what is required to run `make()`[12].
- `load_data()`: Optimizes the dependency graph and loads the catalog data of the end node —the target— into Python.
- `store_data()`: Traverses the dependency graph and stores the catalog data of Source Collections that have a `sourcelist_data`.

3.5.4 Creating New Source Collections by Pulling Data

One of the Astro-WISE paradigms is that data is pulled by specification of the required science product (section 2.4.6). A Source Collection representing a required catalog is requested by specifying a selection of sources and a set of attributes (section 2.4.6). Astro-WISE uses the Source Collection Tree to create a dependency graph of Source Collections with an end node that represents the required catalog. In this section we describe the process of building this dependency graph.

Formulating a Request

Pulling catalog data is done by specifying three things (section 2.4.6):

- A base Source Collection to select the sources from.
- A list of attributes to request.
- A selection criterion.

This can be done with the following convenience function of a Source Collection:

- `derive()`: Creates a Source Collection Tree from the given Source Collection. The end node of this Source Collection Tree is a transient Pass Source Collection (section 3.5.1). The dependency graph of this end node is modified until it represents the requested attributes for the sources that match the selection criterion. This function will call the other functions described in this section.

Pulling Source Collections: Finding Source Collections

The Source Collection Tree will search for existing Source Collections that can be used in the dependency graph to create the target Source Collection. To do this, it keeps

[12] In the current implementation, the Source Collection Tree will not automatically ingest catalog data into the database, even if this is required to process a specific Source Collection further down the dependency graph.

track of Source Collections fetched from the database and newly created transient Source Collections[13].

The `find_*` functions below return a list of Source Collections that can be used to fulfill a specific part of the request. The Source Collection Tree uses a key function to assign a value to each Source Collection and orders the list according to this value. Source Collections with a positive key value are suitable objects to use as a progenitor. The Source Collection with the highest value will be used in the dependency graph. The exact way this is done, varies per function and depends on the goal. The Set Relation classes are used to determine whether the a Source Collection represents the required sources (section 3.7).

The `key_functions` property of the Source Collection Tree contains the key functions that correspond to the `find_` functions. Scientists can influence the ranking mechanism by specifying their own key functions. Furthermore, a scientist can always overrule a decision by the Source Collection Tree by manually swapping the chosen Source Collection with another one with a positive key value.

Pulling Source Collections: Applying a Selection Criterion

If a selection criterion is given, the Source Collection Tree will parse the query and tries to find a Source Collection that represents the requested sources, using the Set Relations classes[14]. If no such Source Collection is found, a new Filter Sources is created with the specified selection criterion as `query`. Any attributes that are referenced in the query are added automatically through the process below. There are several functions used to apply a selection criteria to a Source Collection in a Source Collection Tree:

- `apply_filter()`: A general function that will use `find_selection()` to find an existing Source Collection to use with `insert_selectsources()`. If no such Source Collection is found, `insert_filtersources()` is called.
- `find_selection()`: Searches for a Filter Sources Source Collection that represent the requested sources.
- `insert_selectsources()`: Creates a Select Sources with an existing Source Collection as `selected_sources`. This Select Sources is inserted inbetween the end node of the dependency graph and its parent.
- `insert_filtersources()`: Creates a new Filter Sources. This Filter Sources is inserted inbetween the end node of the dependency graph and its parent.

Pulling Source Collections: Finding Attributes

The Source Collection Tree will search for a Source Collection that represents the requested attributes for the requested set of sources. The Set Relations class (section

[13] Tracking of Source Collections is done through `track_*` functions of the Source Collection Tree, which are not complex enough to warrant a discussion here.

[14] In the current implementation this algorithm is not very strong: The Source Collection Tree will search for a Filter Sources with the exact same query.

3.7) is used check whether a Source Collection that has the required attribute also refers to the same (or super) set of sources. There are several functions to find and add attributes:

- `apply_attribute_selection()`: Applies a selection of multiple attributes to the end node of the Source Collection Tree by calling `add_attribute()` for every required attribute.
- `add_attribute()`: Adds an attribute from an existing Source Collection to the end node of the Source Collection Tree. Calls `find_attribute()` if no Source Collection specifying the attribute is given through the `origin` parameter.
- `find_attribute()`: Searches for the required attribute in existing Source Collections. `find_attribute_new_calculators()` is called if no suitable Source Collection is found.
- `find_attribute_new_calculators()`: Instantiates new Attribute Calculators that are able to create the required attribute (section 3.6.4).

3.5.5 Interaction and Visualization

There are several ways to interact with a Source Collection Tree and the Source Collections therein.

Interaction and Visualization: Dependency Graph Visualization

The dependencies in a Source Collection Tree can be visualized with the following functions:

- `make_dot_graph()`: Creates a diagram like figure 3.2.
- `set_graph_after_replace()`: Automatically creates a diagram after every dependency graph modification.

Interaction and Visualization: Send Source Collections over SAMP

The catalog data of Source Collections can be sent to other applications using the Simple Application Messaging Protocol (SAMP) using the `broadcast()` function of an instance of the Samp class (chapter 5).

Interaction and Visualization: Query Driven Visualization Through SAMP

New SAMP messages are created to perform query driven visualization of catalog data using the Source Collections (chapter 5).

3.6 Attribute Calculators

Attribute Calculator Source Collections are used to to calculate new attributes from existing attributes (section 2.5.10). The calculations themselves are decoupled from their application and specified through an Attribute Calculator Definition object.

The Attribute Calculator operator is designed such that all methods for attribute calculation share the same persistent Source Collection class. Defining a new method is done by creating a new Attribute Calculator Definition object, which does not require alteration of the database (DDL commands), unlike the creation of new Source Collection operators.

3.6.1 Attribute Calculator Definition Class

An Attribute Calculator Definition is a persistent DataObject that defines a calculation that can be applied by an Attribute Calculator Source Collection. Every calculation method should be described by a different Attribute Calculator Definition. Different Attribute Calculators can use the same Attribute Calculator Definition. The persistent properties of an Attribute Calculator Definition describe the calculation, e.g. what attributes are calculated, the attached file contains the calculation itself.

Persistent Properties

- `ACDID`: An integer to identify the calculation method of this Attribute Calculator Definition[15].
- `version`: An integer to describe the version of this Attribute Calculator Definition. The `ACDID-version` combination uniquely identifies a Attribute Calculator Definition object.
- `name`: A human readable name describing the method, e.g. 'kCorrect'.
- `attribute_names`: The attributes that are calculated with an Attribute Calculator using this method.
- `input_attribute_names`: The attributes that are required to calculate the new attributes. The parent Source Collection of an Attribute Calculator using this definition should represent these attributes.
- `calculator_parameters`: A list of Attribute Calculator Parameter objects that represents the free parameters that can be set when using this definition (section 3.6.2). The Attribute Calculator Source Collection stores these parameters as its `process_parameters`. The Attribute Calculator Definition specifies default values for the parameters[16].

[15] Unlike similar identifiers like a `SLID`, the `ACDID` is not used to group objects together.

[16] In the current implementation there is no explicit functionality for scientist to set their own defaults.

- `filename` (inherited from DataObject): The file containing the code to performs the actual calculation (section 3.6.3).

3.6.2 Attribute Calculator Parameter Class

An Attribute Calculator Parameter is a persistent object to store parameters used in the derivation of the new attributes. The `calculator_parameters` property of a Attribute Calculator Definition and the `process_parameters` property of a Attribute Calculator Source Collection are lists of Attribute Calculator Parameter objects. The Attribute Calculator Parameter class has the following persistent properties:

- `name`: Name of the parameter (e.g. `omega_m`).
- `ptype`: Format of the parameter as numpy dtype (e.g. `float`).
- `value`: The value of the parameter (e.g 0.3). Contains the default in Attribute Calculator Definitions and the actual value for the Attribute Calculator Source Collections.
- `description`: A human readable description of the parameter (e.g. 'matter content of the universe').

There are different Attribute Calculator Parameter classes for different types of parameters (different `ptype`). These classes all inherit from the Attribute Calculator Parameter class. The different parameter classes are:

- Attribute Calculator Parameter Integer with `int` as `ptype`.
- Attribute Calculator Parameter Float with `float` as `ptype`.
- Attribute Calculator Parameter String with `str` as `ptype`.
- Attribute Calculator Parameter Source Collection with `SourceCollection` as `ptype`.
- Attribute Calculator Parameter Object with as `ptype` the class name of the required object.

The Attribute Calculator Parameter Object is used for all parameters that are Astro-WISE objects themselves other than Source Collections. Specific Attribute Calculator Parameter classes for other objects could be created in case some classes are used as a parameter often enough to warrant this.

3.6.3 Code

The code for the actual calculation is stored in the file attached to the Attribute Calculator Definition. This file contains a new Attribute Calculator class that inherits from the base Attribute Calculator class. The original class has several virtual

functions that are overloaded by the class in the Attribute Calculator Definition file. In particular these are `make()`, `get_attributes()` and `get_query_self()`.

A directory `ACD[acdid]-v[version]` is created when a Attribute Calculator Definition object is initialized and the code file is retrieved. If the attached file is a `.py` file, it is copied to `AttributeCalculator.py` in this directory, if it is a `.tgz` file, it is unzipped into this directory and should contain a file named `AttributeCalculator.py`. When a Attribute Calculator object is initialized, it is cast to the new class defined by the Attribute Calculator Definition.

The actual calculations can be performed on the Python or SQL level, depending on which methods are provided by the code file. The calculation can be performed on the database if the `get_query_self()` function is overloaded[17]. Most Attribute Calculator Definitions only work on the Python level by overloading the `make()` method of the Attribute Calculator. The `make()` function assumes that the sources of the parent are available through its `data` property. The actual calculation can be performed in Python, but also with an external program. The calculated attributes are kept in the `data` property and appended to `datasuper`.

3.6.4 Instantiating New Attribute Calculators

The Source Collection Tree can automatically instantiate new Attribute Calculators when creating a dependency graph for a pulled catalog (section 2.4.6). This is only done when no suitable existing Source Collections can be found to provide a specific attribute. The database is searched for Attribute Calculator Definitions that list the requested attribute in their `attribute_names`. The Source Collection Tree proceeds to create Source Collections that can be used as a parents for Attribute Calculators using the found definitions. New Attribute Calculators are instantiated for all Attribute Calculator Definitions for which suitable parents could be created.

Creating such a parent is done by pulling the attributes listed in the respective `input_attribute_names` of the Attribute Calculator Definitions. This might again require the instantiation of new Attribute Calculator Source Collections and therefore this mechanism is applied recursively.

The `sourcelist_data` and `datasuper` are created as soon as a new Attribute Calculator is instantiated for which the attributes cannot be derived on the fly. This is done on initialization because it is not yet possible to set the `sourcelist_data` once the Attribute Calculator has been made persistent (section 3.3.3).

3.7 Set Relations

The logical relationships between the sets of sources represented by different Source Collections is inferred from the data lineage with the algorithm described in section 2.6. Knowing these relations is required to pull data (section 2.4.6) and to perform certain dependency graph modifications (section 2.5).

[17] A better way to perform the calculation in SQL might be to let the Attribute Calculator only specify the attribute selection part, instead of the entire query.

The logical relations algorithm is implemented in two non-persistent classes. The *Set Relations* class is used to create and maintain a specific relation between sets, that is, one specific hypercube. The *Set Relations Set* class is based on a set of Set Relations objects that contains all relations that are consistent with the available knowledge about the sets.

The classes are in principle not specific to Source Collections and labels (strings) are used to indicate the different sets. We first describe the general properties of the classes and subsequently the details related to Source Collections.

3.7.1 Set Relations Class

A Set Relations object represents a specific relation between sets as a Python numpy array in combination with a list of labels that correspond to the dimensions of the hypercube (section 2.6.2). The Set Relations can be initialized from a given relation number or symbol (section 2.6.2) and is a hashable object.

Properties

- `matrix`: A Boolean numpy array of size 2^d representing the hypercube of the algorithm.
- `labels`: An ordered list with for each hypercube dimension a list of corresponding set labels.

Multiple set labels can be attached to the dimension of the hypercube, therefore d is usually smaller than the total number of sets.

Functions

We list the member functions that are important for understanding how the Set Relations class works (see section 2.6.4 for details about the algorithm):

- `__init__()`: The Set Relations initializes with relation 3 —there are elements, but no sets— by default. This can be overruled by specifying the `relation` parameter.
- `add()`: Requires a new set label as argument. Returns a list of all Set Relations objects that contain this extra set and are consistent with the original relation.
- `remove()`: Requires an existing set label as argument. Removes a set from the relation, by removing the label from the dimension it is associated with. The dimension itself is removed, by summing over the respective dimension of the array, if there are no more labels associated with it.
- `relation()`: Requires a list of labels as parameter. Returns a Set Relations object that represents the sub-relation between these sets only.

- `number()`: Returns the number of the relation, only useful for relations with dimension three or lower.

- `symbol()`: Returns the symbol of the relation, only applicable for relations with dimension two or lower.

3.7.2 Set Relations Set Class

The Set Relations Set class is used to hold incomplete information about the relationship between a number of sets (section 2.6.1) and is the class used by the Source Collection Tree.

Properties

- `relations`: A set of Set Relations objects that hold the actual relations.

Functions

- `__init__()`: The Set Relations Set initializes with the two relations with no sets.

- `add()`: Requires a new set label as argument. Creates all the relations with this extra set.

- `remove()`: Requires an existing set label as argument. Removes that set.

- `filter()`: Requires a list of labels and a list of allowed relations as argument. Removes all relations that are not consistent with the allowed relations.

- `relation()`: Requires a list of labels as argument. Returns a set of Set Relations objects that corresponds to these labels only.

- `relation_holds()`: Requires a list of labels and a list of relations as arguments. Returns `True` when all relations are consistent with the given sub-relations.

- `find_equals()`: Requires a set label as argument. Returns the labels of all the sets of which it is certain that they have the same elements.

- `find_supers()`: Requires a set label as argument. Returns the labels of all the sets of which it is certain that have the same or a super set of elements.

3.7.3 Source Collection Specifics

The Set Relations classes are tailored for use in the Source Collection Tree.

Source Collection Specifics: Usage by the Source Collection Tree

The `relations` property of the Source Collection Tree refers to a Set Relations Set object. This object is used to track the available knowledge of the relationships between the Source Collections in the dependency graph.

The Source Collection Tree calls `get_source_relations()` of individual Source Collections to track the relation between that Source Collection and its dependencies. The Source Collection Tree will not add a Source Collection to the Set Relations Set when this would make its list of consistent Set Relations too large.

The Set Relations Set is passed to the modification functions of Source Collections (section 3.5.2), because some modifications can only be performed if a specific relation between the Source Collections and its progenitors hold.

Source Collection Specifics: Convenience Methods

The generic way to add a set to a Set Relations Set is to combine the `add()` and `filter()` functions above. The relationships between Source Collections are very specific and therefore convenience functions exist to add such a relationship. These functions are fast because they do not have to perform the filter operation.

Furthermore, the Set Relations class has class members that contain often used relations, for example `AsmallerthanB` contains the four relations that represent that a set is smaller than another set.

3.8 Extensions

We describe several ways to improve the current Source Collection implementation. See also the discussion in section 2.7.

3.8.1 Improved Operators

Some of the Source Collection classes (section 3.4) can be improved and new ones can be designed.

Improved Operators: Source Extraction

The SourceList Wrapper operator is a wrapper around the SourceList class and cannot actually run the SourceList code. Connecting the image pipeline to the Source Collections would make the pulling data paradigm complete from visualization to raw data. There are several ways to approach this:

- Design Source Collection operators that can be performed on images (e.g. Detect Sources and Measure Photometry, section 2.7.2). This could replace some of the functionality of the SourceList and could function as an interface to the subimage pipeline, since selection of sources could be performed before the photometry measurement.

- Allow the SourceList Wrapper to run the SourceList code.
- Let the SourceList be a Source Collection itself.

Improved Operators: Source Association

The identification of sources, can be seen as a two step process: First an association between sources has to be made, which is done with the AssociateList. This involves running actual association code and requires intelligence from the scientist. Subsequently the identification has to be applied, which is done with the Relabel Sources operator. There are benefits to designing and implementing an Associate Sources that would create AssociateLists (see also section 2.7.2):

- All Source Collections can be associated, not only SourceLists.
- Therefore pre- and post-filtering of associates is handled implicitly by using Filter Sources Source Collections.
- Associations can be defined without requiring the association to be performed and the association could be done partially.
- The association can be parallelized in the same way as the other Source Collection operators.

There are technical reasons why it is not trivial to implement such an operator:

- A Source Collection can represent sources with different `SLID`s, the AssociateList data structure cannot handle this.
- It might be difficult to fit all uses of the AssociateList in this scheme, although this might not be necessary (section 2.7.2).
- The result of processing a Source Collection can be seen as tabular data. It is not a priori clear what this tabular structure would be for such an operator.

Improved Operators: Attribute Calculator

There are several possible improvements to be made for the Attribute Calculator and Attribute Calculator Definition classes (see also section 2.7.3):

- Improved mechanism to check whether the calculation can be performed, for example whether any required software is available.
- Some general Attribute Calculator Definitions might be usefull, for example for arithmetic.
- The code that contains the actual calculation is stored as a file on the data server. There might be better ways to approach this, such as with the experimental CodeObject class.

3.8.2 Integration with other Astro-WISE Functionality

The integration between the Source Collection classes and other Astro-WISE functionality could be improved.

Improved Integration: ProcessTarget

More functionality provided by ProcessTarget should be available in the Source Collection classes:

- The `process_status` could possibly be used instead of the `all_data_stored` flag.
- The Source Collections have no explicit functionality to deal with invalid data, the `is_valid` should be used for this.
- Some ProcessTarget functionality is not yet integrated, such as provided by `set_process_parameters_from_dict()`, `set_user_config()`.
- Most functionality that ProcessTarget inherits from OnTheFly is not yet integrated, such as `after_do_make()`, `exist()`, `uptodate()`.

Improved Integration: Target Processor

Some functionality of the Source Collection Tree could be integrated with the Target Processor. This would complete the pulling data paradigm from raw pixels to data used in published graphs.

Improved Integration: DPU utilization

There is currently no explicit functionality to process Source Collections on the distributed processing unit (DPU). The Source Collections parallelize naturally (section 2.4.7), albeit this has not yet been implemented.

3.8.3 Improved Storage

Seamless integration between the database, the `awe`-prompt and external visualization software with respect to handling of catalogs, is one of the goals that lead to the design of the Source Collection classes. The Source Collection implementation is a step in that direction and we describe how this can be improved further.

Improved Storage: Database Caching Algorithm

The principles we use to decide what processing results to store in the database (section 3.3.3) are crude: only attributes derived with an Attribute Calculator are cached by default. The information system can use more advanced heuristics to determine what data should be cached and what not, for example by analyzing which operations take a long time to perform (see also section 2.7.6).

Also, there is no mechanism in place to decide whether the source identifiers of a Filter Sources should be stored, this is left to the scientist. In some circumstances it is faster to perform the selection every time the data is requested, usually if the selection is done on attributes that are indexed.

Improved Storage: Automatic Persistent Storing

Processing results are only stored through user interaction in the current implementation. The main reason for this is that storing catalog data in the database is a relatively time consuming process, which hinders interactive exploration of the data. A way to improve this would be to store catalog data in a parallel process.

Improved Storage: Ingestion from the Database

Catalog data that has to be stored, will always go through the Python layer. It could be a significant improvement if catalog data could be ingested directly through SQL for those cases where the operation can be performed on the database entirely. This is especially relevant for caching the identifiers of the sources of a Filter Sources Source Collection.

Improved Storage: Local Data Caching

A local copy of the catalog data could be kept in addition the persistent storage in the database. This would greatly benefit interactive visualization of often used data. Successfull experiments have be done with caching catalogs in FITS files.

3.8.4 Scalability

A main driver of the Source Collection class is to achieve scalability. We describe several ways in which this can be improved further.

Scalability: `Parallel` hint

Most generated queries could speed up by adding the Oracle `PARALLEL` hint in the right sub query. Usually there is one specific `WITH` clause that is quick to execute and that contains all the required sources. E.g. for a Source Collection with a `sourcelist_sources` set. This has been experimented with successfully, but is difficult to implement in a way that always works well.

Scalability: Fetching Data Piecewise

Currently, catalog data is retrieved from the database with the `fetchall()` function from the database cursor. It might be better to retrieve the rows in multiple small sets using `fetchmany()` so the interface stays responsive if loading takes long.

Scalability: Tree Parallelization

Parallelization as described in section 2.4.7 is not yet implemented.

Scalability: Performing Operators in Python

The Source Collection Tree prefers to perform restructuring operators on the database, while in some case they might better be executed in Python, which should be recognized by the Source Collection Tree. For example, a Concatenate Sources operation is faster in Python than in SQL under some circumstances.

3.9 15 square degree SDSS pilot

In this section we describe how Astro-WISE was used to create the data for our research on density estimation (chapter 4). The research goal is to study the effects of different density estimation methods on the inferred relations of galaxies with their environment.

Catalog data from SDSS is ingested as External Source Collections, from which a data set of approximately 15 square degrees is selected. We quantify the environment of a galaxy as the number density of galaxies at position of the galaxy. It is unknown whether there is a perfect way to calculate densities. In the literature several methods are used, each with its own pros and cons. Four different density estimation algorithms are incorporated as Attribute Calculator Definitions. The differences between the density estimators are studied by investigating their effects on relations between environment, color and structure of galaxies.

Figure 3.3 is used as a reference to describe how the catalog data was handled. We would like to note that parts of the dependency graph were created by 'pushing', because not all data pulling mechanisms were in place at the time these datasets were derived.

3.9.1 Data Ingestion

The `PhotoObjAll` and `SpecObjAll` tables from SDSS DR7 have been ingested as External Source Collections:

- External 100181 contains the 585 634 220 sources (set b) from `PhotoObjAll` with 446 mainly photometric attributes.
- External 100191 contains the 1 635 683 sources (set a) from `SpecObjAll` with 65 mainly spectroscopic attributes.
- The SDSS tables are ingested as independent Source Collections, with different `SLID`s. Since we are interested in sources with spectroscopy only we use External 100191 as a basis for our analysis. Relabel Sources 106791 relabels the sources of the photometric Source Collection to the corresponding sources in the spectroscopic Source Collection. The relabeling is done on the basis of the `specObjID` attribute, a unique key in the original SDSS tables. Only the sources that are both in the spectroscopy Source Collection and in the photometry Source Collection are included, therefore the composition of sources (set c) is not equal to either of them.
- The photometric Source Collection contains attributes that are also in the spectroscopic Source Collection. Select Attributes 106881 selects the non-duplicate attributes.
- The photometric and spectroscopic attributes are combined in Concatenate Attributes 106891.

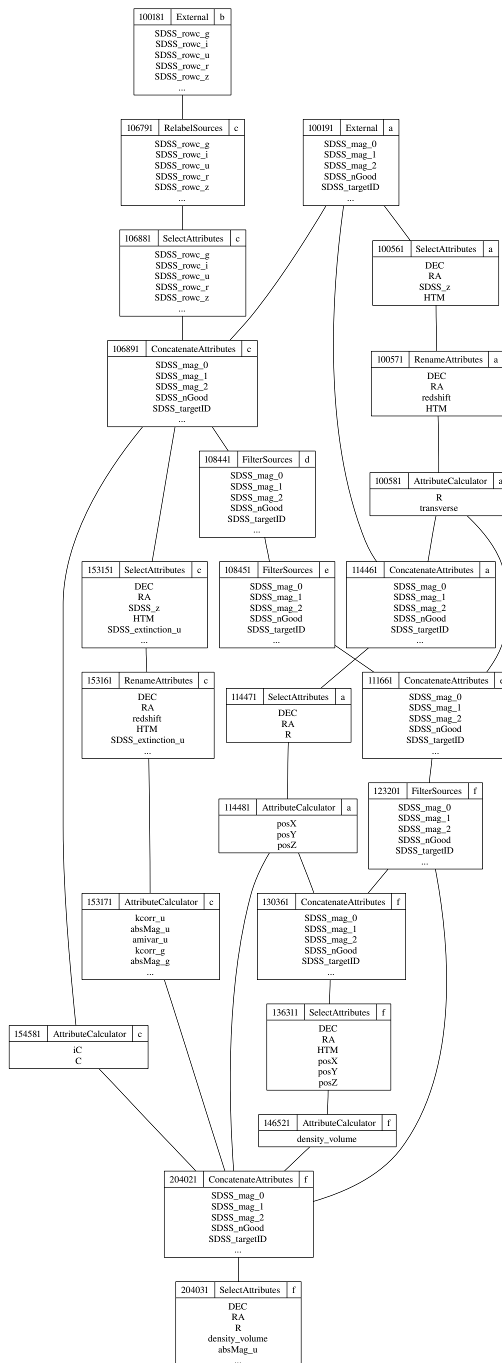

Figure 3.3: The dependency graph of our sample.

Data Ingestion: SDSS attributes

SDSS attributes are all prefixed with the SDSS_ prefix, except for RA and DEC. The shorthand columns of the SDSS model magnitudes (e.g. r for modelMag_r) are removed, because in the SDSS archive there is already another attribute called z, namely the redshift in SpecObjAll. In Astro-WISE, attributes with the same name should always represent the same physical quantity, therefore the duplicates are removed.

3.9.2 Derived Attributes

New attributes are calculated from the SDSS Source Collections using Attribute Calculators: comoving positions, absolute magnitudes (and k-corrections) and the inverse concentration index.

Derived Attributes: Comoving Distance

An Attribute Calculator Definition has been created to calculate the comoving distance (R) from a redshift (redshift). The SDSS Source Collection has the redshift in the SDSS_z attribute, so it has to be renamed first:

- Select Attributes 100561 selects the attributes required for the calculation of comoving distances[18]
- Rename Attributes 100571 renames the redshift attribute so the Attribute Calculator understands it.
- Attribute Calculator 100581 represents the comoving distance for all sources with spectroscopy (set a). A flat universe of with $\Omega_m = 0.3$, $\Omega_\Lambda = 0.7$ and $h_0 = 0.7$ is used. The catalog data of this Attribute Calculator is stored in the database, but only for sources for which the comoving distance is actually requested. The transverse comoving distance (transverse) is also calculated.

Derived Attributes: Cartesian Coordinates

Our density estimators require Cartesian coordinates, so the comoving distance is combined with the sky positions to calculate Cartesian coordinates.

- With Concatenate Attributes 114461 the comoving distance is combined with the photometric attributes.
- Select Attributes 114471 selects the attributes required for the calculation of Cartesian coordinates (RA, DEC, R).
- Attribute Calculator 114481 calculates Cartesian position for all the original sources. The catalog data that is the result of processing this Source Collection is not explicitly stored, because it can be calculated directly in SQL.

[18]The sky position attributes are selected as well, because they are indexed by the database. However, it is not necessary to do this explicitly.

Derived Attributes: Absolute Magnitudes

The absolute magnitudes are calculated using the Petrosian magnitudes with KCORRECT V4.1.4 (Blanton and Roweis, 2007):

- The Petrosian magnitudes, extinction and redshift is selected with Select Attributes 153151.
- The redshift attribute is renamed from `SDSS_z` to `redshift` with Rename Attributes 153161.
- The absolute magnitudes and k-corrections are calculated with Attribute Calculator 153171. This uses an IDL program and therefore this Attribute Calculator can only be processed if IDL is installed. Scientists who do not have IDL can still use the data lineage to see how the attributes are calculated, even if they cannot redo the calculation themselves.

Derived Attributes: Inverse Concentration

The inverse concentration index is a meassure for the compactness of a galaxy. The index is defined as

$$iC = \frac{r_{50}}{r_{90}}, \tag{3.1}$$

where r_{50} and r_{90} are the radii containing 50% and 90% of the Petrosian flux (Baldry et al., 2006).

- Attribute Calculator 154581 calculates the concentration index and its inverse (`C` and `iC`), averaged over the r and i band. In a data pulling environment the information system would have first selected the required attributes with a Select Attributes. However this Source Collection was created manually and this was not done properly.

3.9.3 Sample Selection

Source Collection 106891 contains all SDSS sources with both spectroscopy and photometry. `PhotoObjAll` includes primary and secondary objects, as well as family objects. Family objects are not considered to be separate objects, but belong to one of the primary objects, therefore we should not include them in our analysis. Secondary objects are objects that are measured for the second time, so we disregard these as well. Furthermore, we limited our sample to a small portion of the sky and a limited range in distance:

- With Filter Sources 108441 only primary sources are selected with selection criterion `"SDSS_mode" = 1`. Since this dataset is never requested in its entirety, the exact composition of sources has not been determined.

- With Filter Sources 108451 selects the 2233 sources within our sample region with selection criterion `("RA" BETWEEN 185.0 AND 190.0) AND ("DEC" BETWEEN 9.0 AND 12.0)`. The identifiers of the selected sources are explicitly stored in the `sourcelist_sources` SourceList of the Source Collection.
- The comoving distance is added to this sample with Concatenate Attributes 111661.
- With Filter Sources 123201 a distance and magnitude limit is imposed on the sample with `("R" BETWEEN 50 AND 515) AND ("SDSS_petroMag_r" < 17.7)`. The result is 1030 sources, whose identifiers are cached as well. The SDSS spectroscopic sample is complete to an Petrosian r magnitude of 17.77, we applied a slightly stricter limit.

3.9.4 Galaxy Densities

The environment of our 1030 galaxies is quantified using four different density estimators. For simplicity we show the processing using only one method. The density estimation methods require a Source Collection as input with the sample that defines the density. This is in addition to the parent Source Collection which represents the sources for which the density is estimated. In this study we use the same sample for both the parent and the density defining population (DDP):

- The comoving positions are added to our sample with Concatenate Attributes 130361.
- The comoving positions are selected with Select Attributes 136311.
- The volume density is calculated using Attribute Calculator 146521.

3.9.5 Combining All Attributes

All attributes are subsequently combined:

- Concatenate Attributes 204021 combines the spectroscopic attributes, the photometric attributes, the concentration indices, the absolute magnitudes, the Cartesian positions and the densities.
- Select Attributes 204031 selects the density, the inverse concentration, absolute magnitudes and sky position and distance.

3.9.6 Processing

The node at the end of the dependency graph is the final target Source Collection which represents the catalog data that will be used in plots etc. The information system reorganizes the dependency graph to build the target Source Collection as efficiently as possible (figure 3.4):

- From the entire dependency graph a temporary transient copy has been made which can be seen from the negative numbers used as identifiers.
- The density calculator had all its data stored and has been converted into External -17380791.
- The Select Attributes at the end of the dependency graph has been moved upwards. The part of the graph were comoving positions were calculated was not required anymore and is thus removed from the dependency graph.
- The Filter Sources that performs the final sample selection had the identifiers of its sources stored. Therefore it has been converted into a Select Sources with External -95651700 as `selected_sources`. This Select Sources has subsequently been moved through the dependency graph, resulting in several copies to limit the required processing in all parts of the graph.
- The other Filter Sources Source Collections are removed entirely, since the final sample is determined by the last Filter Sources.
- The Source Collections are subsequently processed. The transient copies of the Attribute Calculators represent a subset of the originals, requiring less processing.
- The target Source Collection is assembled last and the catalog data is returned to the scientist.

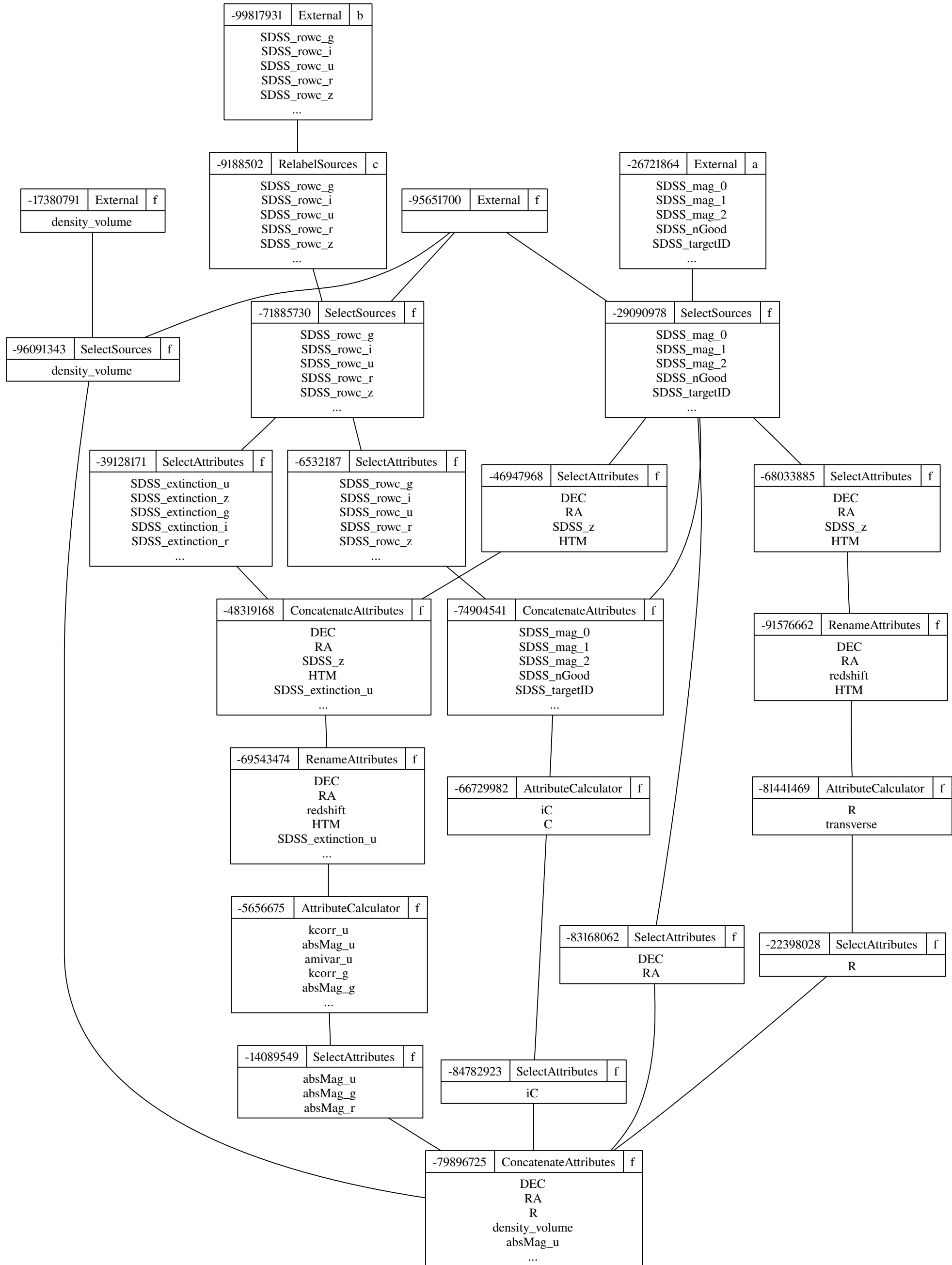

Figure 3.4: A diagram describing the pipeline of our cone sample, optimized for processing.