ASTRO-WISE Architectural Design

including some detailed design of critical components

VERSION 1.0

Document history
DRAFT 0.1: May 2002 - Conceptual design review
DRAFT 0.2: June 2002 - resolving comments from CDR
DRAFT 0.5: Following Quality Control meetings
DRAFT 0.9: May 2003- Following experience with working prototypes
specification of object attributes, fileserver, data ingestion, visualization tool,
hardware updates
QC- Quality Control
5LS- 5 lines script
Version 1.0: 11 July 2003- Approved by Consortium
more details Quality control: Terapix Derfotron tool + new Appendix

The **Astro-Wise** Consortium

July 11, 2003
www-astro-wise.org

Authors

This document is made using the CVS system in a federated database as described in Chapter 3. All ASTRO-WISE partners have contributed to or commented on (Jim Emerson UK, Luiz Da Costa, ESO) the text.

The CVS log recorded written contributions by:

Dr Kor Begeman, OmegaCEN, Groningen
Dr Emanuel Bertin, IAP, Paris
Drs Danny Boxhoorn, OmegaCEN, Groningen
Dr Erik Deul, Sterrewacht, Leiden
Dr Aniello Grado, OAC, Napoli
Dr Mark Neeser, USM, Munich
Dr Roeland Rengelink, OmegaCEN, Leiden, Groningen
Dr Roberto Silvotti, OAC, Napoli
Dr Edwin Valentijn, OmegaCEN, Groningen

Contents

0 Introduction **5**
0.1 Scope of ADD . . . 5
0.2 Scope, federating calibrations . . . 6
0.3 Glossary . . . 8

1 WP1: Provide and operate processing facility of raw image data **10**
1.1 Introduction . . . 10
1.2 Requirements . . . 11
1.2.1 Processing capabilities . . . 14
1.2.2 Database requirements, WP1–WP3 interface . . . 14
1.2.3 Operational capabilities . . . 16
1.2.4 User Interface . . . 17
1.2.5 Pipelines . . . 19
1.2.6 Quality Control . . . 20
1.2.7 Projects, Surveys, What's in a name . . . 21
1.3 The ASTRO-WISE pipeline(s) . . . 23
1.3.1 Scope . . . 23
1.3.2 Tasks . . . 23
1.4 Development framework . . . 24
1.4.1 Methodology . . . 24
1.4.2 Python . . . 25
1.4.3 Unit testing and qualification . . . 25
1.4.4 Inline documentation . . . 25
1.4.5 CVS . . . 26
1.5 Prototypes . . . 26
1.5.1 The EIS pipeline . . . 26
1.5.2 The OmegaCAM pipeline . . . 27

2 WP2: Provide tools for querying, searching and visualisation **30**
2.1 Visualization tools . . . 30
2.1.1 Raster images . . . 30
2.1.2 Multidimensional vector data . . . 36
2.1.3 Graphical database query tools . . . 37
2.2 Source extraction/ morphological classification tools . . . 40
2.2.1 SExtractor . . . 40
2.3 Quality checks/ human interface to data/ data mining . . . 40
2.3.1 Data browser/ plotting / trend-analysis . . . 41
2.4 Near/inside database access tools . . . 42
2.4.1 Basic set of queries . . . 42

2.4.2 Variability tool . . . 42
2.4.3 Associate source lists . . . 43
2.4.4 Tell me everything within ASTRO-WISE tool (basic ingredients provided by WP1 and WP3) . . . 43
2.5 Spectral Energy Distribution fitting tool . . . 46
2.5.1 PSF Homogenization . . . 47

3 WP3: Provide Federated database 48
3.1 Introduction . . . 48
3.2 Provide database engine which supports the following operations . . . 50
3.2.1 Administration of pipeline operations as done under WP1 . . . 50
3.2.2 Storage for all source list data . . . 50
3.2.3 User programs, contexts, permissions . . . 50
3.2.4 Provide basic set of queries, including complex queries . . . 51
3.3 Provide WAN for ASTRO-WISE . . . 51
3.3.1 Introduction . . . 51
3.3.2 The context of the database operations . . . 51
3.3.3 Network speeds . . . 52
3.3.4 What are the options? . . . 52
3.4 The database interface . . . 53
3.5 CVS database for federating . . . 53
3.6 Interoperability with WP2 . . . 54
3.6.1 Introduction . . . 54
3.6.2 Data types . . . 54
3.6.3 Querying . . . 54
3.6.4 Representation . . . 54
3.6.5 Tell me everything within ASTRO-WISE . . . 55

4 WP4: Provide parallel processing power to ASTRO-WISE 57
4.1 Introduction . . . 57
4.2 Work description . . . 57
4.3 Investigations . . . 57
4.4 Relation with WP5 . . . 58
4.5 AstroBench . . . 58
4.6 Hardware at Groningen / Leiden . . . 59
4.7 Hardware at Paris . . . 60
4.8 Hardware at USM . . . 60
4.9 Hardware at Capodimonte . . . 61

5 WP5: Provide data storage to ASTRO-WISE 63
5.1 Introduction . . . 63
5.2 Work description . . . 63
5.3 Investigations . . . 63
5.3.1 Hardware . . . 63
5.3.2 Software . . . 64
5.3.3 Prototype . . . 66
5.4 Hardware at Groningen / Leiden . . . 66
5.4.1 - dataserver for CVS, dbI, observational data . . . 66
5.5 Hardware at Paris . . . 66
5.6 Hardware at USM . . . 67
5.7 Hardware at Capodimonte . . . 67

6 WP6: Coordination **68**
6.1 Project organization . . . 68

A Persistency Interfaces **69**
A.1 Introduction . . . 69
A.2 Background . . . 69
A.2.1 Object Oriented Programming . . . 69
A.2.2 Persistency . . . 70
A.2.3 Relational Databases . . . 70
A.3 Problem specification . . . 71
A.4 Interface Specification . . . 71
A.4.1 Persistent classes . . . 72
A.4.2 Persistent Objects . . . 73
A.4.3 Queries . . . 74
A.4.4 Functionality not addressed by the interface . . . 74

B Quality Control: possible problems and solutions **75**

Chapter 0

Introduction

0.1 Scope of ADD

This Architectural Design Document (ADD) gives an overview of the ASTRO-WISE survey system which the consortium plans to develop and operate. It follows the organizational structure of the project with 6 Work Packages (WP). The document gives an overview of the joint developments of the different partners.

For each of the WP's the document describes the components that will be supported. The text lists the high-level ASTRO-WISE specific concepts, without much details. However, the concepts relating to the *interfaces* between different components and WP's are detailed (see also Appendix).

The document forms the backbone of the design of the ASTRO-WISE survey system. The goals for the system are ambitious in the sense that the system should support with a minimum of administrative manpower the massive dataflow from wide field imagers, which in the case of OmegaCAM is planned for a 10 year period for, say, 300 nights per year. Next to this, all the raw and processed data should be disseminated over the partner sites. Therefore, the project has a lot of emphasis on the administrative problem which, in turn, is handled by early definitions of standards and a common database engine which both handles the pipeline I/O, all the source lists and the distribution of data items over the various partner sites. A key of the project is that the administrative system at the same time facilitates overall operations, like calibrations and project surveys, and the *personal* use of the system.

A key concept of the ASTRO-WISE system is that data items will be continuously added to the database, either due to new data coming in from the telescope, or due to new versions of methods or calibration data. ASTRO-WISE will create *dynamical catalogues* often computed on request of the end user. The same system can be used to create more classical, static, catalogues when for a particular project one wants to freeze versions of methods and calibration data. But, in fact, the ASTRO-WISE system can contain *ALL* OmegaCAM data in a form which is of direct interest for astronomical use. The system allows a user to do their own experiments with large or small subsets of the data. The system is tuned for OmegaCAM data but other Wide field imager data will be supported as well.

The current prototypes can deal with WFI@2.2m and INT wide field image data. The system is designed to support other wide field imagers, such as MegaCAM and VISTA. The design of the system is such that it is a relatively simple task to add another instrument to the system (by specifying their own classes).

WP1 describes the ASTRO-WISE pipelines which contain a lot of OmegaCAM components. These pipelines should either run in "standard" mode or be user tuneable, facilitating interactive analysis, and supporting the zoo of scientific objectives (ranging from finding a single variable or moving

object, to comparing galaxy counts in different hemispheres).

WP2 describes the various back-end tools for visualization, source extraction and querying large (Tbyte) databases. The present documentation of WP2 in Chapter 2 provides much more details than the other sections, and should be assessed on the Detailed Design level, while the other sections are on the ADD level.

WP3 describes the federated databases which will support the ASTRO-WISE administration and glue together the operations and developments at the various sites.

WP4 and WP5 list the hardware provided by the various partners to the ASTRO-WISE survey system. This hardware will form a wide area network (WAN), both for development work and for the astronomer using the system. It is envisaged that most WAN services will be provided by the implementation of the database in WP3.

Network speeds are taken as a free parameter in the present concept, but in the ideal system no data is replicated and the database always knows where to point and collect the relevant data. In practice, replication will be needed, in the Oracle environment this will be provided by the Advanced Replication components. The system is designed to allow to move to European direct access, with no replication of image data, to be implemented as soon as partners are linked to the required network throughput.

WP6 is coordinating this project, where the view is that the key to the success of the ambitious system is to have a lot of emphasis on design and setting standards. Also it will require a lot of art, flexibility and good-will from all partners to find compromises between local interests (the bazaar) and the overall system (the cathedral). The ambition is indeed to find the proper balances between the two.

ASTRO-WISE can deliver data products to the Astrophysical Virtual Observatory (AVO) and close contact with the VO's and the Grid projects, like AstroGRID and DataGRID, is very desirable, particularly for the inter-operability issues.

In itself the ASTRO-WISE system, which only supports wide field imaging instruments, represents a kind of Virtual Survey System, combining archive access, pipeline processing and user tuneable querying and processing.

0.2 Scope, federating calibrations

This section is essentially a summary of the proposal and contract texts and serves as a reference for the present document.

The ASTRO-WISE consortium is an initiative of the OmegaCAM and VST consortia, together with the ESO-EIS group and Terapix. The ASTRO-WISE system initially focuses on the support of the OmegaCAM /VST optical camera, though the design of the system should allow support of other wide-field instruments.

Current wide field imagers, such as WFI@2.2m and INT can be supported.

It is planned to start including MegaCAM in the federation starting in 2004 and at a later phase the system can be tuned to also host VISTA data.

The ASTRO-WISE system will facilitate distributed data reduction and calibration of the OmegaCAM /VST data. The procedures for data acquisition at the telescope and calibrations are outlined in the OmegaCAM URD and the CP documents. The NL-Nova team builds such components for the ESO operations pipeline for handling the calibrations and "removing instrument fingerprints" from the raw data. Experience with and also libraries of the EIS programme supported this activity. The components of these pipelines and calibration reductions will be delivered by the NOVA team to ESO-DMD. The production of these components is part of the OmegaCAM - ESO contract and is not part of the official ASTRO-WISE programme. ESO-DMD will operate these components in a DFS pipeline, to maintain all calibrations thoroughly, and to do a first cut data reduction for the VST observing programmes, in service mode.

A general OmegaCAM /VST observing and calibrations strategy is to maintain the instrument continuously, with standardized procedures. In this concept the "instrument is calibrated", any "standard" observation done with the instrument should be calibratable with these general calibration procedures (there are very few exceptions, like programmes requiring extreme photometric accuracy - say better than 0.02mag).

The same procedures used by ESO/dmd to maintain the instrument will be used by ASTRO-WISE . In fact, the federated system operated by ASTRO-WISE will disseminate the calibrations. By merging these data reductions procedures with ESO-EIS components ASTRO-WISE builds an exportable system for maintaining the calibrations. It is because of the availability of the OmegaCAM components and the extensive EIS libraries that we feel we are in a position to go one step further and to disseminate an integrated system through the community. Basically, the ASTRO-WISE system can be viewed as a technique to distribute software and calibrations, together with the raw and processed data through the community. At the same time the community can contribute to the calibrations and feed back their results into the system. Data can be re-derived with such new calibrations. Projects, such as large survey or Public Surveys can be run independently. The work horse for the federation is a common database engine which will control both pipeline operations and will supervise the dissemination. In the Oracle 9i era the dissemination is handled by the Advanced Replication components. Available network speeds will determine how much data will be replicated or only stored locally. Interoperability to external VO systems is open through the VO table concept, while direct interoperability to external archives can be achieved through the Oracle STREAMS components.

As all the data taking and data reduction is strongly procedurized, all OmegaCAM observations can potentially be used in an archive. The ASTRO-WISE federation means to provide such an archive, next to large surveys and public surveys. Contrary to the Public Surveys, such an overall archive will have a variable quality and will change continuously. Users can define their own archive project and re-run the calibrations, this way effectively creating dynamic catalogues. Because the intimate coupling between the sourcelist/catalogue level and the maintaining of the calibrations, both being disseminated and federated it is desirable to have all these handled by the same database engine. Common interfaces for pipeline and calibration derivation procedures will be developed (the delivery to ESO-dmd will contain an I/F to a file system, while the general ASTRO-WISE system will interface via Python and SQL to the Oracle-db). Oracle STREAMS will be used to connect the database to other databases, such as SyBase (as operated by ESO-DMD) and MySQL (as operated by Terapix). These connections will allow the exchange of data files, code and calibrations. The ASTRO-WISE federations can be used to disseminate Public Surveys over the community, but also recursively to involve the community in the production of these surveys. Thus, the ASTRO-WISE system can expand on the current EIS system and provide a wider dissemination and involvement of the community.

WP3 provides such a database engine, while WP1 provides the user interface and operations of an image reduction and calibration pipeline, which heavily leans on OmegaCAM and EIS components. WP2 provides additional analysis and visualization tools, while WP 4 and WP5 provides the necessary hardware to the ASTRO-WISE system.

The ASTRO-WISE project is essentially a Research and Development project, the present document means to outline the grand context, which will be build on top on existing successful libraries like EIS, Ldac, Eclipse, OmegaCAM. During the project the emphasis might change, as industrial products (processors, storage, db) will evolve, AVO projects will deliver useful middleware and available network speed will increase. The ASTRO-WISE archives or certain project contexts areas out of the the ASTRO-WISE archive will be made available with the AVO infrastructure. ASTRO-WISE also supports a limited amount of operations of the network.

0.3 Glossary

5LS Five lines script: handfull of user specified Python code allowing to execute complex operations and/or queries

ADD Architectural Design Document

API Application Programmers Interface

AVO Astronomical Virtual Observatory

CP The OmegaCAM Data Flow System Calibration Plan

CGI A method to serve dynamic web pages

DB Database

DBMS Database Management System

DDL Data Definition Language, a language to define the contents of a database (see also SQL).

DFS (ESO) Data Flow System

DML Data Manipulation Language, a language to manipulate the contents of of a database (see also SQL)

DRS The OmegaCAM Data Reduction Specifications

GID A Global Indentification within the federated database such that an ingested data item can be uniquely referenced

GTK A software library (Graphical ToolKit) for GUIs

GUI Graphical User Interface

OOP Object Oriented Programming, a paradigm based on the notion that data and functions are intimately related

Object The central concept in OOP, a "thing" that comprises data (attribute-values, state) and functions (methods, behavior)

Persistence The mechanism used in OOP to save and retrieve data (i.e. the state of objects) from a database

PHP A language for dynamic web pages.

QC Quality Control

Qt A software library (toolkit) for GUIs

RDBMS Relational Database Management Systems

SQL Structured Query Language A language for manipulating databases and their contents (SQL is both a DDL and a DML)

Tkiniter A software library (toolkit) for GUIs

UI User Interface

URD The OmegaCAM Data Flow System User Requirements

VO Virtual Observatory

WAN Wide Area Network, a network connecting multiple sites (using the internet)

WP Workpackage

XML eXtendable Markup Language

Chapter 1

WP1: Provide and operate processing facility of raw image data

1.1 Introduction

The aim of work package 1 (WP1) is to *develop, implement and operate a full wide-field imaging pipeline from raw data to astronomically calibrated images*, and *to populate the database with the necessary calibration information*.

Here, we give an overview of the requirements of the pipeline and the key concepts of the pipeline architecture.

Details of the required processing capabilities of both the calibration and data-reduction pipelines are given in the OmegaCAM User Requirement Document, the Calibration Plan and the Data Reduction Specification Document. These documents provide a baseline for the ASTRO-WISE requirements. Hence, we will focus here on additional requirements and designs. These designs define the ASTRO-WISE pipeline environment and include:

- user-driven operations (both automatic and interactive operations),
- transparent database interfaces for pipeline administration, and
- distributed processing.

A prototype pipeline, following the present specifications, has been implemented and was used to iterate on the present design. The aim of this prototype is to investigate solutions to the design problems posed by the pipeline requirements. This prototype is based on design decisions that will be clarified in this document.

The WP1 text has been organized as follows. Section 1.2 gives an overview of the requirements, both from a user and a developer point of view. Section 1.3 translates these requirements to specifications of the ASTRO-WISE pipeline architecture, including an overview of the implementation tasks. Section 1.4 specifies a development framework that will deliver this pipeline. Finally, section 1.5 describes the current prototype.

1.2 Requirements

The following OmegaCAM documents provide a detailed description of the requirements of the OmegaCAM operations, including calibration procedures, scientific observing modes, and required data reduction operations.

1. **OmegaCAM Data Flow System User Requirements (URD)** (VST-SPE-OCM-23100-3050) gives an overview of the scientific requirements, including calibration and processing requirements.
2. **OmegaCAM Date Flow System Calibration Plan (CP)** (VST-PLA-OCM-23100-3090) gives a detailed description of all calibration procedures.
3. **OmegaCAM Date Reduction Specifications (DRS)** (VST-SPE-OCM-23100-3051) specifies the data reduction tasks for the calibration and science data

The ASTRO-WISE pipeline requirements obey the requirements as laid out in these documents, but in addition involves important new features such as interoperability, user-driven processing and archiving of all pipeline I/O. These will be discussed here.

An overview of the data processing modules as it is anticipated to operate in the ESO environment is given in Fig. 1.1. The diagram is exhaustive and includes procedurized operations at Paranal and at ESO headquarters both regarding quality control (QC0), calibration file derivations (QC1), the operation of an image pipeline, and transferring raw images into astrometrically and photometrically calibrated images. The colored boxes in the flow diagram represent data processing modules that produce results represented by white boxes.

Main ASTRO-WISE concept are:

- **to operate and distribute the same datamodel over the various ASTRO-WISE sites**
- **the end-results, all the I/O of the pipeline obeying this datamodel, and all information necessary to reproduce the results, are stored in a database.**

The yellow boxes in Fig. 1.1 represent the modules that will form a persistent part of the system. The green labelled boxes will not form part of the ASTRO-WISE pipeline, they function in the local quality control of operations (at Paranal and ESO-DMD). These modules, in general, do not produce calibration data required for pipeline operations. The discrimination between green and yellow boxes is one of the compromises made to keep the project manageable. For the same reason, critical operations done at Paranal are repeated at ESO headquarters and at the ASTRO-WISE sites, where the relevant databases are maintained.

In addition, Fig 1.1 contains blue colored modules, representing post-processing and analysis tools, that are described in more details in WP2. Next to the image stacking, image differencing, image mosaicing, and image monitoring which the ASTRO-WISE pipeline will support (all beyond what is operated for the general ESO user by ESO-DMD), a number of source extraction and visualization tools and overall source list database query tools will be supported.

- **An important ASTRO-WISE concept is to integrate these tools, and the source lists in the same environment as the pipeline operations**.

A pipeline may be defined as a series of connected modules, where the output of one module is the input of another module. It is convenient to distinguish between different pipelines, as illustrated in Fig.1.2. Here we distinguish between the bias pipeline, the flatfield pipeline, the photometry pipeline, and the image pipeline. Each pipeline produces a distinct result (bias, flatfield, zeropoint and calibrated science data respectively), that provides a convenient/obvious entry-point for human quality control.

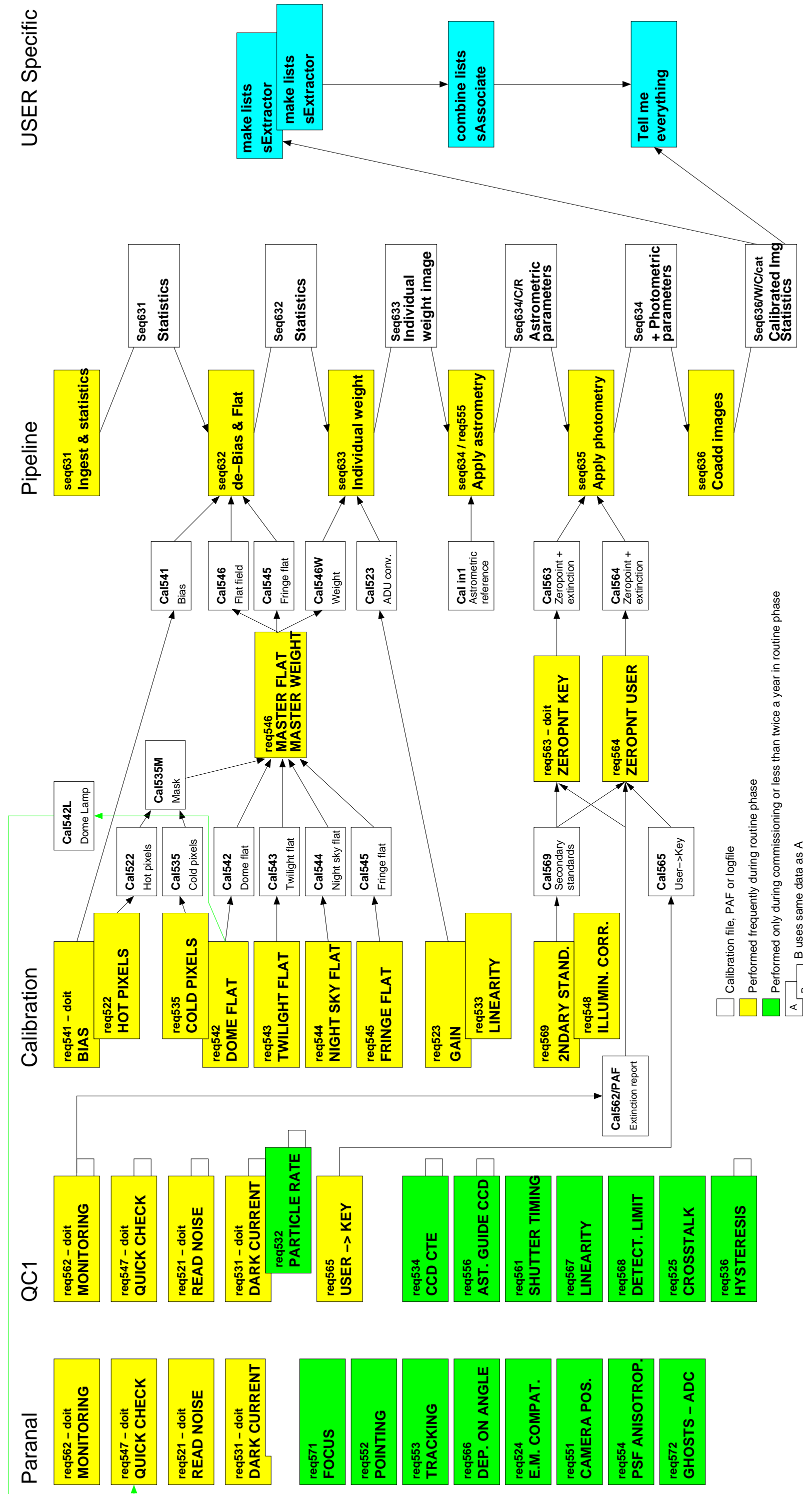

Figure 1.1: The OmegaCAM data model. The ASTRO-WISE pipeline encompasses the data-reduction modules colored in yellow (all visible to the end-user).

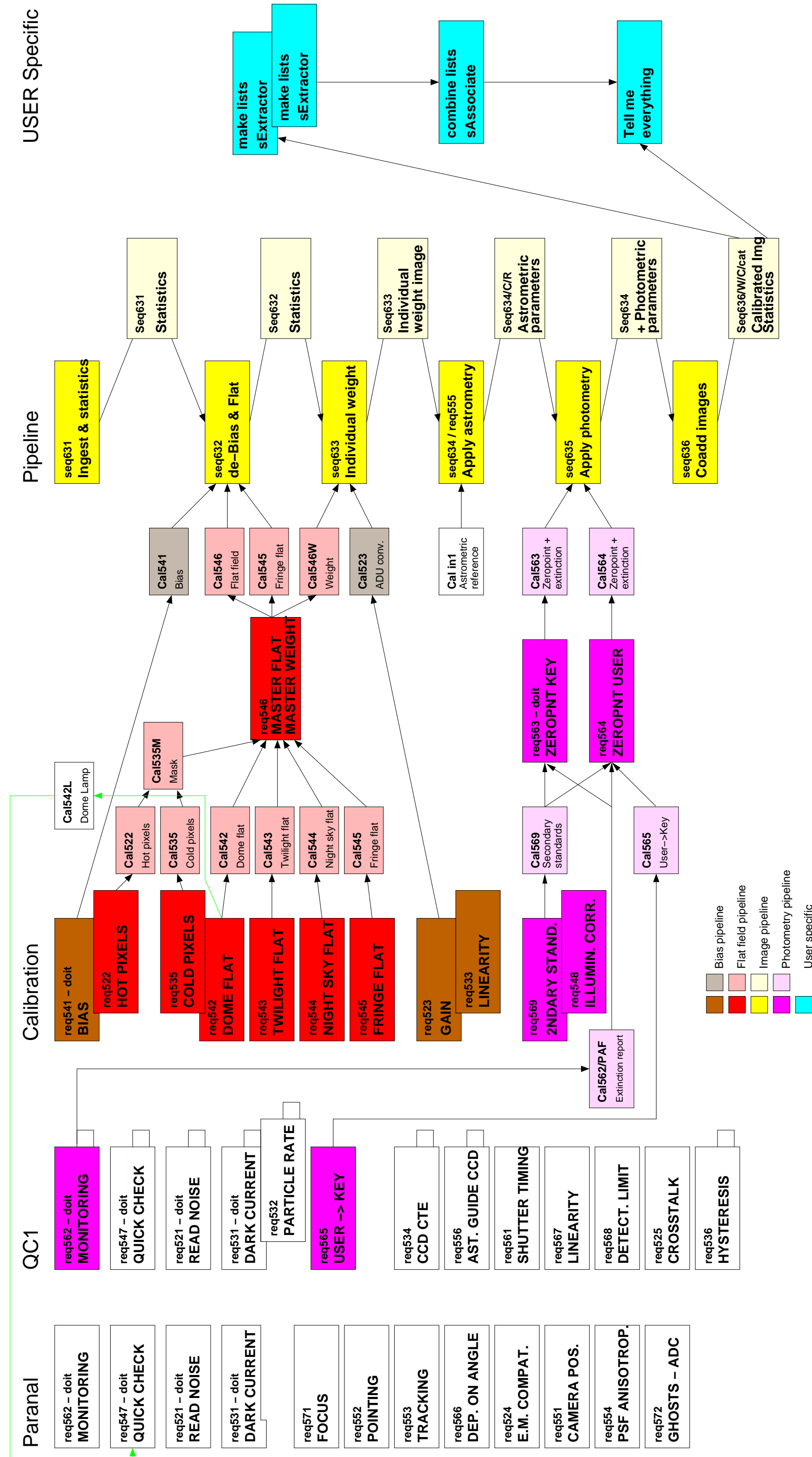

Figure 1.2: The OmegaCAM data model. The model distinguishes between different pipelines. The most important are the flatfield pipeline, the zeropoint pipeline, and the image pipeline

In addition to the requirements set forth by the OmegaCAM documents, the following high-level requirements are specified.

1. **Multi instrument support:** The aforementioned documents specify the requirements of a single instrument. The ASTRO-WISE pipeline should be easily extensible to support multiple optical (first of all MegaCAM) and IR imaging instruments (first of all VISTA).
2. **Multi site operations:** The aforementioned documents specify data processing within an ESO Data Flow System (DFS) context. The ASTRO-WISE concept assumes that processing and administration may be distributed over multiple sites.
3. **Database driven operations:** DFS assumes that processing is driven by the contents of FITS files/headers. ASTRO-WISE assumes the meta-data that drives processing is primarily stored in a database and processing is triggered by queries of the user. Moreover, post-pipeline analysis will ideally be accomplished within the same framework as used by standard pipeline processing.
4. **Reprocessing:** The ASTRO-WISE pipeline should enable on-the- fly reprocessing of the data. In the ideal implementation of the system, it does not matter whether a result is returned from the database or (re)processed on the fly. This implies that (i) all information necessary to reproduce a result should be retained and (ii) the interface should allow this.
5. **Extendibility:** It should be possible to add/replace processing methods and define new data products, extending (but not modifying) the data model,

In the following sections we will detail some of the requirements:

1.2.1 Processing capabilities

Section 7 of the DRS gives a detailed description of low-level processing functionalities required by the OmegaCAM pipeline, including:

- **Image manipulation** including, arithmetic, statistics, convolution, masking, and coaddition
- **Catalog manipulation** including source extraction, association, astrometric calibration, and PSF measurement
- **Header manipulation** including reading, writing, and validation

1.2.2 Database requirements, WP1–WP3 interface

Here, we address some of the database issues related to pipeline administration and operations. Related issues are addressed in WP3 and an implementation of persistent objects (see below) is explained in more detail in Appendix A.

While many image pipeline algorithms are well developed nowadays an important challenge is designing and developing the administrative aspects of a pipeline. ASTRO-WISE focus is on handling pipelines as an administrative problem.

Our solution relies on:

- **Persistence:** The pipeline operates by manipulating persistent objects.

Persistence is a concept from Object Oriented Programming (OOP). An object is said to be persistent if it is able to 'remember' its state across program boundaries. Persistence is usually implemented on top of a database. The aim is to make this process transparent. This means that modifying the state of an object (for example, assigning a value to an attribute) will automatically lead to an update of the database. Modifying an object and storing/updating a database with the objects contents should not be separate tasks.

- **Query:** The user should have maximum freedom in querying the database.

Persistence is usually implemented as a mapping from object identity to object state. That is, given the identity of an object, retrieve its state from a database. 'Searching a database', however, is an activity that relies on solving the opposite problem. Namely, finding a mapping from a (partially specified) state to object identities. This is the problem that relational databases solve (i.e: SELECT object WHERE object.state = value). Therefore, the database requirements suggest an implementation of an interface from persistent objects to an (object-)relational data base.

- **Extendable schema:** The database schema should be extendable.

Extending and modifying object attribute and method definitions through inheritance and polymorphism is the central concept in OO technology. The database should support these concepts, allowing the end-user to define new persistent data products.

Because schema modification complicates history tracking, it is mandatory to use inheritance and polymorphism to introduce new classes or processing routines to the pipeline. That way, any new processing routines or structures that are defined will be additions to the database schema instead of modifications to the database schema.

The ASTRO-WISE database provides an operational environment for both pipeline data processing, data processing for specific (science) projects, and end-user data processing and analysis. Therefore the database should provide facilities for the following related concepts:

- **Multi-user:** The database should allow for multiple users with different privileges (e.g.: developer, pipeline operator, project scientist, browser).
- **Multiple contexts:** The objects in the database may have meaning only in specific contexts (e.g.: development and testing, general pipeline operations, alternative pipeline operations (reprocessing), a particular survey, a particular science project, a data mining project)

These concepts together provide a course-grained and fine-grained notion of ownership/responsibility of data respectively. Specifically, it is understood that objects may have a limited lifetime within the database for some users or in some contexts. The valid context(s) is an attribute of each persistent object. This is implemented with a link to one or more context objects. The definition of the corresponding (derived) types (see 1.2.7) will enable the end-users to select based on the similarities and differences between various kinds of contexts. A number of attributes shared by many persistent objects, that may be useful in determining a context is given in Table 1.1.

The validity of persistent object may also be limited to a particular time. Hence:

- **Time-stamping** The database should be able to timestamp objects with a single time or a time range.

Calibration data have an assigned validity period. The assignment of the proper validity period is provided by the calibration pipelines and quality control mechanisms. This is the key mechanism for the image pipeline to identify the proper calibration files, but also other mechanisms can be supported, like using other object attributes such as `owner` or `project`.

Table 1.1: Some key object attributes to determine object context. These attributes are used in selecting relevant data for particular operations.

Atribute	Values
`project`	Calibration, Science, Survey, Virtual
`owner`	pipeline, developer, user
`observing_block`	Bias, DomeFlat, Dither
`strategy`	Standard, Deep, Freq. (see CP)
`mode`	Quick, Jitter, Dither (see CP)
`timestamp`	valid time range

Table 1.2: Some key characteristics of data that can be used to select data for processing and analysis. Attributes marked with an asterisk also describe the observing block, and, hence, can be seen as part of the context

Atribute	Values
∗`RA, DEC`	Target position
∗`filter`	The filter
`seeing`	The FWHM of stellar objects
`DATEOBS`	Date of observations
`EXPTIME`	Exposure time

The relevance of data (to a user) is not only determined by context, but may also be determined by physical characteristics of the data. These characteristics can be used to select data for processeing and/or analysis. An overview the most important of these characteristics is given in Table 1.2

The database will store both the meta data (administrative information) and the bulk data (FITS files and catalogs). Hence, the following requirement:

- **Bulk data storage** The database interface should hide the details of the storage of the bulk data (FITS files and catalogs)

Finally, the operational environment of the pipeline (see next section) dictates the following requirement:

- **Multi-site** The database should be distributable over multiple sites

1.2.3 Operational capabilities

The data rate of the instrument sets a firm constraint on pipeline performance:

- **Pipeline performance** The pipeline shall reduce data at a rate of at least 1 Mpix/sec (100 exposures/night, reduced in the same time as a night lasting 8 h, i.e. 3 times faster than overall data acquisition).

It is expected that this performance target requires parallel processing. In addition, data must be distributed over multiple sites. Hence:

- **Parallel processing** The pipeline shall work in a parallel processing environment (Beowulf)
- **Remote processing** The pipeline shall work in a remote processing environment

Both requirements are met with a single interface.

1.2.4 User Interface

The ASTRO-WISE pipeline framework comprises a library of re-usable software components written in Python and external packages with Python interfaces. The core of this library consists of the definition of persistent objects which provide a binding to the Oracle database. These persistent objects have attributes which describe the data. The persistent objects are stored in the database and their attributes can be used to select objects from the database. The objects also provide methods to process and analyse the data. The control of data processing is discussed in section 1.2.5. A key part of the analysis is quality control, which is discussed in section 1.2.6.

There are three different levels for the user to interact with this infrastructure:

- **Command lines:** Standard applications can be started from the ASTRO-WISE monitor in Unix-type command line mode (in some cases to be covered by GUIs).
- **User provided applications:** Users can modify functionalities in modules and insert them into the system when these modules obey the standard data model.
- **User scripts:** users can write (short) scripts allowing to glue existing pipeline applications and data base access in a compact way (five lines script- 5LS).

User interfaces should provide pipeline operators and end users the ability to:

1. operate the pipelines (image, photometry, flatfield) in interactive (single-stepping through modules) and batch mode;
2. retrieve and modify the (default) configuration parameters of each data reduction operation, and, if necessary, (re-)execute the operation with the new parameters;
3. create or import alternative (calibration) data, and (re-)execute data reduction operations using these data;
4. visually inspect the (intermediate) results of data reduction operations, using image visualization tools and interactive plotting tools;
5. execute additional analysis operations on selected data in order to verify results.
6. perform trend analysis on selected results;
7. produce derived data products from default data products produced by standard pipeline operations (e.g. color catalogs from single-pass band catalogs; deep observations or mosaics from combined pointings).

Command lines

A user interface for applications may be a simple unix-like command line with arguments, or a more fancy GUI. The key feature of the user interface is that it hides from the user all information and data-processing capabilities that are not directly relevant to the task at hand. Examples of standard applications are given in the Recipes directory of the ASTRO-WISE pipeline prototype. The ESO Imaging Survey provides examples of how an ASTRO-WISE GUI could look like. This is discussed in more detail in Section 1.5

User provided applications

Users can modify functionalities in modules and insert them into the system when these modules obey the standard data model. By using the full OO inheritance machinery, the user can modify and extend any part of the ASTRO-WISE pipeline. User-defined scripts provide a flexible way to control data processing and analysis.

User scripts - 5LS

User provided handful of lines of Python script - 5LS (five lines script) provide an additional powerful tool. All data transfer is, at least on Meta-data level, passed through the data base. The ASTRO-WISE Python binding to the Oracle database allows the end-user to write his top-level script addressing the database exclusively in Python code. This way, the user can, from the same script, trigger the processing of image or catalogue data (as in command lines and user provided applications) and query the database on results, apply computations to these results and visualise them. This facilitates, in a powerful way, complex and USER-defined add hoc operations to the data, which will have more flexibility than pre-defined Gui's, that are more useful for the more routine type of operations.

In the following Python examples data is selected from the database, manipulated and plotted. When used with a database that is populated with raw, calibration and processed data the examples result in figure 1.3.

```
################################################################################
from astro.main.RawFrame import RawBiasFrame, RawDomeFlatFrame
import Kplot
################################################################################
# Example A
# Find all raw bias frames for a ccd named A5506-4.
q = RawBiasFrame.chip.name == 'A5506-4'

# From the query result, get the observing date in terms of the modified Julian date
x = [k.MJD_OBS for k in q]
# Get the mean pixel value of the raw bias image.
y = [k.imstat.mean for k in q]

# Make a plot of observing date vs. the mean bias value.
g1 = Kplot.simple.MarkerGraph(x=x,y=y)

################################################################################
# Example B
# Find all raw bias frames for a ccd named ccd53.
q = RawBiasFrame.chip.name == 'ccd53'

# From the query result, get the observing date in terms of the modified Julian date
x = [k.MJD_OBS for k in q]
# Get the difference between the mean of the overscan in the x-direction and
# the mean of the prescan in the x-direction.
y = [k.overscan_x_stat.mean-k.prescan_x_stat.mean for k in q]

# Make a plot of observing date vs. difference of the pre- and overscan mean
# values in the x-direction
g2 = Kplot.simple.MarkerGraph(x=x,y=y)

################################################################################
# Example C
# Find all raw domeflat frames for a ccd named ccd56 observed with filter
# called #841 with exposure times > 2 seconds
```

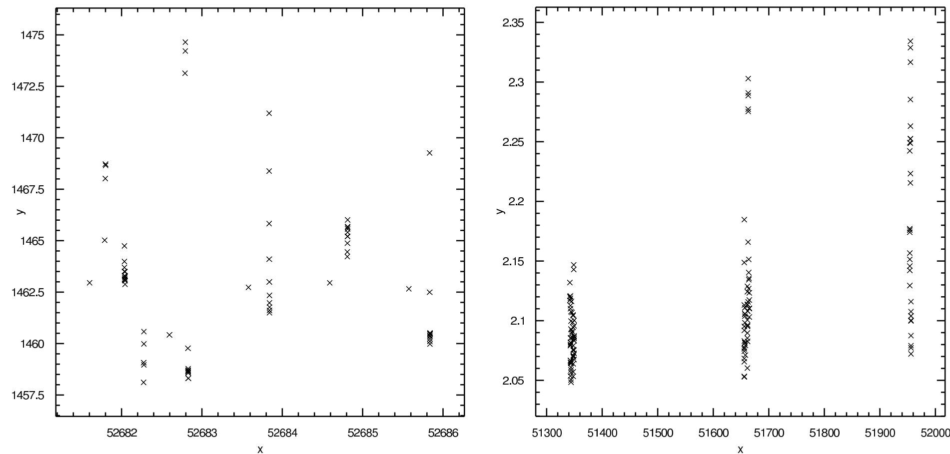

Figure 1.3: Demonstrating power of a 5LS: Selection and output of mean bias as function of time and difference in pre- and overscan as function of time, both for a single ccd.

```
q = ((RawDomeFlatFrame.chip.name == 'ccd56') &
     (RawDomeFlatFrame.filter.name == '#841') &
     (RawDomeFlatFrame.EXPTIME > 2))

# From the query result, get the observing date in terms of the modified Julian date
x = [k.MJD_OBS for k in q]
# Get the mean pixel value of the domeflat divided by the exposure time.
y = [k.imstat.mean/k.EXPTIME for k in q]

# Make a plot of observing date vs. mean domeflat value divided by the
# exposure time.
g3 = Kplot.simple.MarkerGraph(x=x,y=y)
################################################################################
# Postscript output of the plots
g1.device.hardcopy('g1')
g2.device.hardcopy('g2')
################################################################################
```

1.2.5 Pipelines

The ASTRO-WISE data model describes several "pipelines". Pipelines may be simple or complex but basically perform the following functions:

1. determine what result should be processed
2. select the data that is needed to produce that result
3. do the processing

To facilitate re-processing we follow the make-metaphor.

'make' is the unix utility used to keep track of dependencies in building large software packages. The aim of this utility is to only rebuild (recompile) those parts of a package that depend on

something that has been changed. The unix *make* utility works by specifying **targets**. Targets have **dependencies** that may themselves be targets. Targets are rebuilt, in a recursive cascade, when their dependencies change.

The ASTRO-WISE pipelines specify processing in terms of **targets** and **dependencies**, similar to the make utility. A target is built by calling its `make()` method. This should be contrasted to a more conventional implementation, where the target are the return values of a method of one of the depending objects.

The dependencies may be filled by queries to the database, based on properties of the target. For example, one of the dependencies in processing science data is the flatfield. A flatfield can be selected from the database based on the date of observation of the science data and the validity timestamp of flatfields in the database. In code:

```
target = ScienceFrame()
...
# select a flat based on filter and timestamp
target.flat = list((MasterFlatFrame.filter == target.filter) &
                   (MasterFlatFrame.valid_time_start < target.DATE_OBS) &
                   (MasterFlatFrame.valid_time_end > target.DATE_OBS))[0]
```

Data processing can depend on user-tunable parameters. In order to keep track of these parameters they are implemented as attributes of persistent parameter objects. In practice:

```
# a target
bias = BiasFrame()

# the processing parameters
pars = BiasFrameParameters()

# do overscan correction of prescan_x region (=1)
pars.OVERSCAN_CORRECTION = 1

# After selecting dependencies start processing
bias.pars = pars
bias.make()
```

1.2.6 Quality Control

In order to verify the results of the data processing, objects can have `verify()`, `compare()`, and `inspect()` methods. These methods implement basic quality control mechanisms.

verify The `verify()` method inspects the values of various attributes of the object to see if these are within the expected range for that object. The purpose of this method is mostly to perform sanity-checks on measured results. It is assumed that the required measurements (for example image statistics) are done during data reduction (i.e. while executing `make()`), and stored in persistent attributes.

compare The `compare()` method is used for default trend analysis. This is done by either comparing with a previous version of the same object (last weeks bias, for example) or by comparing with a series of previous versions. This may be as simple as comparing attribute values, but may also involve more complex computations (e.g., subtracting the two images, and analysing the residuals)

inspect Visual inspection of the data remains a powerful tool in quality control. The `inspect()` method provides the mechanism to record the results of visual inspection for posterity.

The `verify` and `compare` methods provide simple algorithmic quality measurements. They result in binary quality assessments that are stored in flags. For example, the verify method of raw twilight flat frames may specify that exposures with a mean level larger than 35.000 counts are overexposed, which sets the `frame.OVEREXPOSED` flag to True. Defining and implementing QC measurements (beyond those already defined in the CP) is a vital task of WP1, necessary to achieve the level of quality we envision for ASTRO-WISE .

The `inspect` method allows the user to record Quality Control assessments, based on his or her inspection of the data. In addition to the standard inspection methods built into the recipes the user can run interactive tools to do:

Visual Inspection Quality control will often entail no more than simple (visual) inspection of results produced by the pipeline (e.g. the display of a reduced image, a set of numbers from the database, or a log file.)

(Trend) Analysis Quality control can also be based on simple analysis of aggregate results (i.e. trend analysis). This will usually involve retrieval of selected data, some simple processing and subsequent display of that data using simple plots.

Processing Finally, quality control may be based on additional data processing steps. An example would be the production of color catalogs, and subsequent comparison of color-magnitude and color-color diagrams with theoretical models of stellar populations. This kind of quality control implies running additional pipelines

1.2.7 Projects, Surveys, What's in a name

The ASTRO-WISE framework provides an operational environment for data processing for specific science projects and end-user data processing and analysis. The framework will, at the same time, host individual users running their own experiment, finished projects, projects combining observations over larger areas of the sky (surveys), combining time series, or combinations of these. Technically, these enterprises are not very different from each other, but global parameters are required to identify individual projects, to partition them in the data base (for instance for access privileges), but also for identifying applicable methods. Therefore ASTRO-WISE uses the abstract term `context` which gives a certain meaning to data. In practice, astronomers, or groups of astronomers, will have to agree what they have in mind with a certain project and how they want to constrain their `context`. A set of context attributes facilitates this.

One parameter that assigns a context to data, is the `project`. The following `types` of projects are supported:

Calibration Project All standard calibration observations belong to the same "project"

Science Project A standard science project, comprising a number of observations that may be more or less inter-dependent, depending on the chosen strategy (normal, deep, or freq, see CP)

Survey Project (see next section)

Virtual Project A project that uses archive data, rather than observed data, as its source.

Maintenance Project all data related to testing and maintaining the database belong to this type of project.

In addition to its type, a project is characterized by the following attributes:

ID A project will have a unique identifier

People A project will have people responsible

Status e.g.: Defined, Observed, Processed, Verified

(Public) Surveys

(Public) Surveys aim for coherent data products – a set of data products that are more than just aggregates of data from individual pointings. It is an example of a project for which astronomers have agreed about certains constraints on parameter values and methods.

That coherence is provided by:

1. the limited scope of the survey (survey area, pointings, filters);
2. uniformity of the data (sensitivity/exposure times, seeing);
3. uniform calibration procedures;
4. uniform data reduction procedures;
5. a limited, predefined, number of data products;
6. rigorous data validation, at the level of the data products;
7. preliminary scientific analysis as an integral part of the validation.

The scope of the survey (1) and uniformity of the data (2) will be determined by the observations and not by the ASTRO-WISE specifications. The uniformity of the calibration procedures (3) and data reduction procedures (4) are provided by the ASTRO-WISE pipeline. Furthermore, the pipeline will provide an environment for defining data products (5), and implementing validation procedures on these products (6, 7). However, the definition of data products and validation procedures constitute a separate task. Some of the issues related to these aspects of surveys will be detailed here.

Some key services provided by the ASTRO-WISE environment, that are of particular importance for surveys are:

versioning Since surveys strive for rigorous data validation, producing survey products is very much a process of iterative refinement of the processing operations (fine-tuning parameters). Hence, many different versions of data products will exist.

projects The data model for describing science projects will include the notion of surveys. Surveys differ from ordinary science projects in that they may encompass multiple observing sessions on different telescopes. The project notion is particularly important for representing the state of the survey.

processing interface A "survey-aware" user interface for data processing will be implemented. This interface may complement the standard pipeline processing user interface.

data products Survey projects aim to provide self-contained data products. Data products will be defined, and the ASTRO-WISE environment will support production, validation and distribution of these data products. These data products will be "self-describing", and compatible with AVO standards.

1.3 The ASTRO-WISE pipeline(s)

In the previous section we have outlined some of the requirements and concepts for the ASTRO-WISE pipelines. Here, we will give an overview of the tasks that have to be accomplished to produce these pipelines

1.3.1 Scope

The ASTRO-WISE data reduction pipelines should meet the ASTRO-WISE design goals like multi-site processing/archiving, remote user access and integrated post-processing facilities.
The ASTRO-WISE pipelines will:

1. implement all procedures defined in the OmegaCAM CP and DRS needed to process the calibration data for a multi-CCD imaging instrument.

2. implement a data-reduction process to produce calibrated co-added images of a single pointing.

In addition, WP1 will provide the tools to:

1. build maps combining data from different pointings, or different observations of the same pointing, including large area mosaics.

2. mask defects other than CCD defects, such as cosmic rays and satellite tracks.

3. homogenize the PSF.

1.3.2 Tasks

Below, we give a tentative breakdown of tasks that need to be accomplished in order to build the ASTRO-WISE pipeline.

Build interfaces to processing libraries

The goal of this task it to provide uniform interfaces to low-level processing libraries (eclipse, LDAC, Sextractor, SWARP). Particular attention should be given to specifying configuration parameters.

Implement object persistence

The goal of this task is to implement the base class (DBObject) that gives derived classes the property of being persistent. In particular, the necessary database interfaces have to be implemented that support schema evolution and querying.

Define a persistent class hierarchy

The goal of this task it to build a class hierarchy, derived from the persistent base class (DBObject), that supports the data model specified in the CP. This includes an implementation for all data reduction procedures specified in the CP and DRS as methods on persistent objects. In addition a number of persistent objects specific to the ASTRO-WISE pipeline need to be defined. In particular a Stack type (subtypes: DeepStack and MosaicStack) needs to be defined.

Pipeline interfaces

The goal of this task is to provide high level interfaces to the various pipelines (bias, flatfield, photometric, image). These interfaces should simplify routine data processing.

Quality Control

The QC measurements specified in the CP give only basic insight into the quality of the data and the data processing. Hence, an important task is to define additional QC measurements. This task comprises a number of sub-tasks:

1. list which problems and errors can occur in data acquisition and processing.
2. define which measurement could detect these problems.
3. implement the tools to do these measurements.
4. incorporate the measurements in standard processing.

An initial list of possible problems (and solutions) is given in appendix B.
In particular, for what concerns the Derfotron tool cited in table B.2 and B.3 (appendix B), this QC tool, which is under development at Terapix-IAP, is mainly focused on tracking background level, background noise, source density and PSF variations across the field of view. Moreover it will be usefull also for statistics on residual fringing and for automatic masking of trails, spikes and large optical ghosts.

1.4 Development framework

The ASTRO-WISE pipeline will probably comprise more than 100k lines of Python code. It will be developed and maintained by a number of programmers distributed over multiple sites. This cannot be achieved without agreement on a common development framework.

A development framework comprises a set of methods, practices and tools that steer the development process. Below we give an outline of the elements of the development framework currently in place.

1.4.1 Methodology

We will not adopt a formal development methodology. However, it is necessary to establish a number of practices aimed at what would normally be the goal of any formal development methodology[1]

Deliver usable code

The key word is usable. This implies, in order of importance:

- **Working**, meaning running, more specifically: passing the unit tests.
- **Understandable**, both to developers (understandable code) and to end users (understandable interfaces)
- **Satisfying requirements**, which can only be determined by using the code.

[1] Actually, most of the practices outlined here are part of a development methodology known as 'agile' or 'extreme' programming. See `http://www.martinfowler.com/articles/newMethodology.html` and `http://www.extremeprogramming.org/` for details.

1.4.2 Python

ASTRO-WISE uses Python as scripting languages. It is an important glue between sites, and levels, astronomers, and programmers. Astronomers will modify Python scripts as a means to run their own recipes and methods. The db recognizes such actions by means of the flagging system.

The prototype has been written in Python .

Python in particular facilitates:

- **Modular programming**
- **Object Oriented Programming**
- **Rapid Application Development**
- **In line documentation**

1.4.3 Unit testing and qualification

The only way to know if the code works is to run it and see if it produces the expected results. Conversely, code that produces the expected results, by definition, works. Unit tests aim at specifying the expected results. Hence, code can only be said to work if it passes the unit tests.

Establishing a proper unit testing framework relies on the following practices.

1. When adding a new feature, first write a test for the feature.
2. After modifying something, run the complete test suite, so that you know your changes didn't break anything.
3. When you discover a bug, first write a test that would have caught the bug, then correct the bug.

Apart from the obvious 'knowing that your code works', we find that unit testing provides at least two additional benefits. Firstly, unit testing allows real maintenance of the code base, because the developers do not have to be afraid of accidentally breaking code when they modify it. Secondly, writing tests first forces developers to specify exactly what they wish to accomplish with a new feature, leading automatically to better design.

Code should be tested and **qualified**. Unit-testing is a tool for the programmer that helps him to prove that the code does what he thinks it should be doing. Code should be tested and also **qualified**. Qualification and verification are meant to prove that the code does what the specifications say it should be doing. This implies that:

1. Qualification is carried out by the writer of the specifications.
2. Qualification is carried out at the user interface level.
3. Qualification is carried out, as much as possible, under real-life circumstances, using real data.

Of course, qualification should be part of the development process too.

1.4.4 Inline documentation

Python allows one to attach documentation (doc-strings) to modules, classes, functions and properties.

There are several tools to give the user convenient access to the documentation provided in doc strings, including a html-server that can be accessed with any browser. The ASTRO-WISE website contains a link to the documentation of the ASTRO-WISE pipeline.

1.4.5 CVS

A versioning system like CVS is essential for a collaborative project like the ASTRO-WISE pipeline. However, the benefits of CVS go beyond maintaining a central repository of the code base. The development methodology is based on continuous delivery of working code. CVS provides a straightforward delivery mechanism for the evolving ASTRO-WISE pipeline, thus facilitating continuous feedback from the end-users (who, in the end, determine if the pipeline satisfies the requirements).

Using CVS implies that developers follow the following practices:

1. Update your local repository before committing your work. Ensure that your code is consistent with these latest updates.
2. Commit early, commit often, but only commit working code. Hence, aim for small, localized changes.

A convenient web interface to the ASTRO-WISE CVS repository can be found at cvs.astro-wise.org.

1.5 Prototypes

The development effort for WP1 does not start from scratch. The EIS pipeline provides good examples of the kind of capabilities specified in this document for ASTRO-WISE . In addition it provides a large code base that may be integrated into ASTRO-WISE . We have also implemented a pipeline prototype. The aim of this prototype is to investigate solutions to the design problems posed by the pipeline requirements. This section discusses both the EIS pipeline and the ASTRO-WISE prototype.

1.5.1 The EIS pipeline

The pipeline data processing framework for public surveys developed for the ESO Imaging Survey (EIS) provides a good example of the kind of capabilities of a graphical user interface (GUI) that one might want to develop as an interface to the ASTRO-WISE pipeline.

EIS has built extensive expertise and a large infra-structure to support (public) surveys on a wide variety of instruments. ASTRO-WISE aims to leverage both the expertise and results from EIS. However, it is important to note that two distinct concepts underly EIS and ASTRO-WISE respectively:

data products The fundamental goal of a survey, and hence the focus of the applications developed by EIS, is to provide data products.

services The ultimate ambition of ASTRO-WISE is to provide a set of services that enable the end-user to build their own data-products.

Although these concepts may appear somewhat orthogonal, it should be realized that:

1. A substantial amount of observing time on wide field imaging telescopes is, and will be, committed to (public) survey work. Hence, the ASTRO-WISE environment will support the concepts of surveys and data products.
2. Data products are built by utilizing the very same services that will also be used by end-users.

Survey support in ASTRO-WISE is discussed in detail in Section 1.2.7

The EIS GUI comprises a number of top level interfaces to control the operation of various pipelines (e.g. the image pipeline, the photometric pipeline, catalog production pipeline). Each module provides interfaces to (i) select the data, (ii) build a reduction plan for the selected data, (iii) execute the operations in the reduction plan and (iv) inspect and analyse the results. Analysis tools include: a skycat interface to overlay generated catalogs; plotting tools to present calibration results; facilities to inspect logs; and tools to measure pipeline performance. In addition the GUI provides top-level modules to define and compile data-products, perform basic scientific analysis on catalogs, and do pipeline and database administration. This description only scratches the surface.

The complete EIS pipeline processing code base, including GUIs, but excluding external libraries, comprises more than 250k lines of Python code and is the result of over 10 man-years of full-time development. This is, arguably, a complex application! And, although the application is probably more complex that it would be if were written **again** from scratch, most of this complexity reflects the inherent complexity of the problems that EIS and ASTRO-WISE wish to solve.

As much as 50 percent of the development effort for EIS (150k lines of code, 5+ man years) has been spent on the user interface. It is impossible to give an exhaustive list of UI components included in EIS, however, a large number of these components will also be present in ASTRO-WISE UIs. Below we outline a strategy for using the EIS efforts in developing ASTRO-WISE UI components.

To use EIS UI components within the ASTRO-WISE framework the following issues are being addressed:

Toolkits EIS will upgrade the existing graphics toolkit (Tkinter) to either Qt or GTK, this will provide an excellent opportunity to define interfaces that will make the UI components (widgets) reusable in the ASTRO-WISE pipeline.

Libraries EIS uses a number of external packages to analyse and validate their data products. In an effort to further modularize the EIS code base, uniform interfaces to these packages will be developed.

1.5.2 The OmegaCAM pipeline

Scope

The goals of the pipeline prototype are the following:

1. provide a complete set of working modules capable of reducing Wide Field Image data within the context of the ASTRO-WISE data model
2. provide a complete set of interfaces to low-level data-reduction tools
3. provide a technology demo of an object-oriented interface to an (object-)relational database
4. provide a technology demo of a parallel processing environment
5. establish the development framework

Overview

Figure 1.4 shows the various components of the ASTRO-WISE pipeline. There are three layers. The Library layer provides (i) a number of low level interfaces for data processing, (ii) a persistence mechanism with database interface(s), and (iii) a set of auxiliary libraries, many of which are part of the standard Python library. The API (Application Programmers Interface) layer consists of

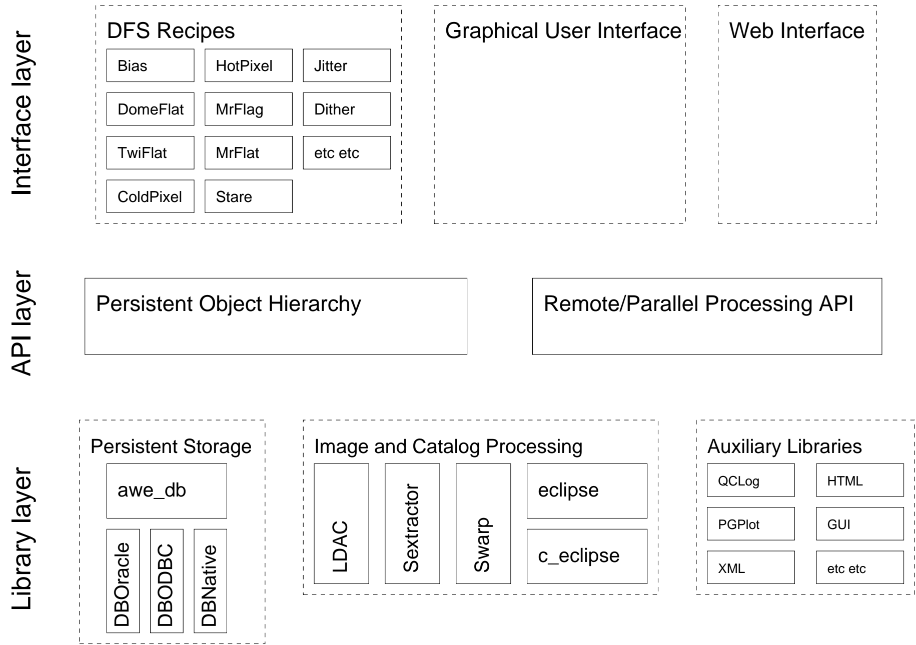

Figure 1.4: Components of the ASTRO-WISE pipeline.

two components. The Persistent Object Hierarchy is the core of the pipeline. Processing is done through invocation of methods on these objects. Since these objects are persistent, all operations are automatically saved into a database. The API layer also provides an interface for parallel/remote processing. Finally, the Interface layer, provides user interfaces, or applications, to invoke and manipulate the pipeline processing.

Note that normally the user interacts with the library through the persistent object layer. That is, the user manipulates the persistent Catalog and Stack objects, rather than Sextractor and SWARP.

Interfaces to external packages

The library layer provides a number of interfaces to external data processing packages. These interfaces can be used independently from the pipeline.

eclipse Low level image and header processing facilities are provided by the eclipse processing package. Using SWIG, the Simplified Wrapper and Interface Generator, the package has been transformed to a Python C-extension library (c_eclipse). A Python wrapper around the extension module exposes the library to the user through a number of objects. These include:

- **cube** an object encapsulating image data from one or more FITS files. Most of the image processing requirements of the pipeline are provided by methods on this cube object.
- **header** an object that can be used to manipulate a header pretty much as one would use a Python dictionary
- **pixelmap** an image for which each pixel has a value of either 1 or 0, allowing for logical operators on pixel data.

The eclipse Python extension is currently part of the eclipse distribution but maintained by a member (Rengelink) of the ASTRO-WISE development team.

The ASTRO-WISE version of eclipse includes some new routines that were developed by the consortium in order to solve a few specific problems. The new routines are:

– sigmaclip (author A. Volpicelli) used to combine BIAS and Flat-Field frames;

– Bicubic spline method (author A. Volpicelli) for 2d cubic spline fitting;

– Sign test (author A. Volpicelli) for BIAS QC.

LDAC Catalog manipulation is done through the LDAC programs. The interface provides a number of functions that are thin wrappers around system calls to the LDAC programs. Configuration parameters are passed to the LDAC function as keyword arguments.

The interface also provides a number of utility functions that extract data from catalogs into native Python types.

Sextractor Source extraction is done through Sextractor. The interface provides a number of functions that are thin wrappers around system calls to sex. Configuration parameters are passed to functions as keyword arguments.

SWARP Coaddition is done through SWARP. The interface provides a function that is a thin wrapper around a system call to SWARP. Configuration parameters are passed to the function as keyword arguments.

Object hierarchy

The persistent object hierarchy provides the core functionality of the ASTRO-WISE pipeline prototype. Processing steps are performed by calling methods on these objects. Values assigned to attributes of these objects are stored in the database. The implementation is discussed in more detail in Appendix A.

User Interfaces

The top layer of the ASTRO-WISE pipeline components consists of applications built on top of the pipeline APIs. These applications allow users to process actual data. Currently, only a minimal set of 'applications' is provided, namely the recipes.

The recipes included in the pipeline prototype provide minimal examples of the use of the ASTRO-WISE functionality. These recipes are command-line programs to perform data reduction operations on the files provided by the user as arguments to the recipe. These recipes operate on single-chip data, and can be used to build calibration data, and to reduce and calibrate science data.

Note that the layer between the persistent objects and the user interface, i.e.: the pipeline objects, are not currently part of the ASTRO-WISE prototype.

Parameters

Many processes require parameters. Default values for these parameters are set in the code. Parameters are themselves persistent objects, so that we can always retrieve from the DB the parameters that were used in a particular reduction process. Parameters can be changed by:

1. Making new instances of parameter objects (with new values) in user-defined scripts

2. Providing new parameter values through a User Interface

Note that the user does *not* change parameters by changing the default value that is supplied in the source code, but by providing alternative values in their own scripts.

Chapter 2

WP2: Provide tools for querying, searching and visualisation

2.1 Visualization tools

A large part of Work-Package 2 addresses software issues related to the interface between humans and the data.

First of all, ASTRO-WISE has to deal with pixels, lots of them: ASTRO-WISE needs a sophisticated and efficient image visualization tool for both quality control and science work. This is described in §2.1.1.

In addition to reduced image data, the ASTRO-WISE pipeline generates multi-dimensional vector data in the form of source lists and calibration statistics. Such high-dimensional data require a somewhat more complex visualization tool than raster images, because of the stronger interaction needed with the database. Incidentally, it is difficult to separate the aspects of "vector data visualization"(§2.1.2) and "graphical query" (§2.1.3); at some level they probably should be integrated as a single tool: a "data explorer" in the virtual observatory vocabulary. One of the important technical issues related to the data exploration tool is that of the interface. A web-based interface (PHP, CGI, Java,...) would be best in terms of portability and accessibility. But this approach has severe limitations in the display capabilities it provides with currently available browsers. Possibly, a hybrid solution based on a "standard" web-based interface, plus optional "plug-in's" (downloadable executables) would offer the best functionalities. As a first attempt, the plug-in's should be made available on the Linux-x86 (Mozilla) and Windows (Explorer, Netscape) platforms, at least.

2.1.1 Raster images

The ASTRO-WISE project develops a generic visualisation tool optimized for astronomical survey images. Indeed, existing visualisation tools are far from optimum for survey applications:

- The most popular tools are built over TCL/TK and are fairly inefficient in terms of display speed (scrolling, zooming) and X11 bandwidth usage (see Panorapix documentation).
- Although they now handle true-colour displays (24 and 32 bit), most are still using 8 bit look-up tables. This does not allow real-time adjustment of intensity mappings with sufficient quality, causing banding if the dynamic range is not manually cropped initially.
- Initial display parameters (intensity range) are generally inappropriate for quick look, and require systematic fiddling with contrast and offset adjustments to get the image right.

- Many of these tools assume that the whole frame can fit into the memory of a typical workstation, which is certainly not the case for forthcoming wide-field instrument images, especially the co-added ones.
- The capabilities of current display hardware, now dominated by accelerated graphic cards with true-colour display, real-time resampling capabilities, and alpha channels, are not used at all.

We may take advantage of the new possibilities of display hardware (hardware texture mapping/anti-aliasing through the OpenGL standard) to provide a tool with unprecedented real-time functionalities like smooth panning, zoom-in, zoom-out features, transparency, image composition, and real-time image morphing.

A simplified, web-browser "plug-in"version may also be made available for image visualization through a database query.

Based on our past experience at TERAPIX developing the Panorapix software, we can estimate the time necessary to fulfil the basic requirements above to about 2 man-years for a qualified engineer.

The AstroWISE visualisation tool will be built in 3 phases.

Phase 1

The purpose of the Phase 1 software is to provide basic functionalities and a working core for the visualisation engine. We will benefit a lot from the work done on the visualisation engine of the Panorapix prototype `http://terapix.iap.fr/soft/panorapix` developed at TERAPIX by N. Decoussemaker in 1999. The phase 1 engine shall feature

- Modular, object-oriented approach (See Fig. 2.1 for the Panorapix model)
- optimized support for Multi-Extension FITS (MEF); it should be possible to have access to a list of their content, and to select one or several extensions; at least in the common case where all MEF files have identical sizes, a crude, low-resolution version of the whole field should be displayed
- support for FITS data-cubes, where selectable 2D slices, from BITPIX=8 to BITPIX=-64 format must be readable (at least to float accuracy, with double precision as a compilation option for instance)
- handling of very large images thanks to memory mapping; images bigger than 3GB must be displayable on 32bit systems with filesystems that support larger files (LFS)
- ability to handle, at least partially, incomplete images if the header is correct
- optimized handling of true colour (16, 24 and 24 bit) displays with support for palettized displays; conversion from image data to screen pixel values takes into account the gamma of the monitor
- optimized X11 management, with emphasis on remote displays
- smooth scrolling ("hand" pointer)
- large (3×2^{16} elements), non-linear floating-point RGB lookup-table to allow for an extremely large dynamic range in intensity (160dB or more); the Panorapix intensity-mapping pipeline has proven to work extremely well in this respect (Fig. 2.2)

- adjustable 4-dimensional (R,G,B,alpha) LUTs using B-splines or linear sections; predefined LUTs include the traditional linear, sqrt, square and log, modulated by the gray, heat, rainbow, and negative colour tables
- compensation for monitor gamma, adjustable in the preferences with built-in calibration tool
- intelligent default flux scaling when loading images
- simultaneous handling of at least 10 frames with a list tool "à la Panorapix" to copy/exchange/apply display parameters (contrast, luminosity, LUT, position in image,...) between frames
- flexible real-time magnifier
- ultra-fast blinking between frames (using the X11 backstore function), 'curtain blink' (à la Gipsy), and frame tiling (à la DS9)
- zoom-in/zoom-out (with binning and decimation options) available at any time, with no limit in both directions; the use of OpenGL texture MIP-mapping to generate in real-time anti-aliased images has to be investigated
- fast image flipping and rotation
- portable, POSIX compliant and autoconfigurable C/C++ code; does not rely on the endianity of the processor; note that portability to Microsoft-WindowsXP, although not required in the professional world, would facilitate the diffusion of this tool in the amateur-astronomer community.
- possibility to save and load a configuration files (preferences) stored in editable ASCII format; a ".***rc" file in the user root directory may be used as the default configuration file
- rely as much as possible on open-source, portable libraries; suggestions:
  - GUI interface:
    - Qt (`http://www.trolltech.com/products/qt/index.html`) or
    - GTK (`http://www.gtk.org`)
  - Math/numerical analysis routines:
    - GSL (`http://sources.redhat.com/gsl`)
  - High-level image formatting/rendering
    - OpenGL (`http://www.opengl.org`)
    - Mesa3D (ersatz of OpenGL: `http://www.mesa3d.org`)
  - FITS input/output
    - CFITSIO (`http://heasarc.gsfc.nasa.gov/docs/software/fitsio/fitsio.html`)
  - World Coordinate System
    - WCSlib (`http://www.atnf.csiro.au/people/mcalabre/WCS.htm`)
- support for SIMD features (MMX/SSE/3Dnow!/SSE2) and 64bit addressing
- If possible, the compiled code should fit in a single executable that can be run from any directory path.
- Root privileges must not be necessary to install/use the program.

Additionally, the phase 1 software shall include the following basic features

- pixel and world coordinate display; WCS keywords are recognized according to the latest Greisen & Calabretta proposal (http://www.atnf.csiro.au/people/mcalabre/WCS.htm)
- a built-in tool to calibrate the monitor gamma
- basic Postscript and GIF[1]/PNG/JPEG/TIFF export function

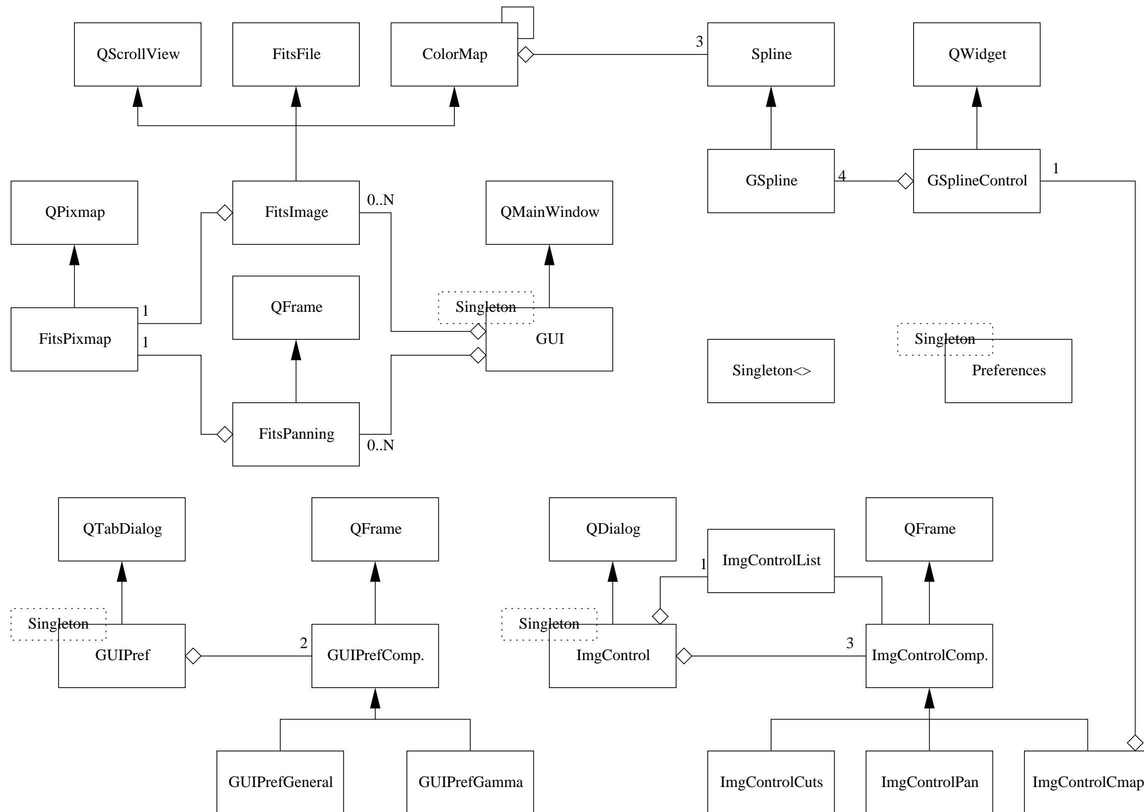

Figure 2.1: Panorapix object model.

The following suggestions have been brought concerning the GUI (Graphical User Interface):

- The GUI should be event-based, preferably using one of the Graphical high-level libraries mentioned above
- important functions (zooming, display parameters) must be accessible through widgets (not only menu items)
- the GUI should give access to redefinable keyboard shortcuts, especially for frequently used functions like blinking
- it should be possible to have most of the widgets placed to the left or the right of the main display area (most astronomical CCD frames are taller than wide when displayed with north-south along the vertical axis)

[1] Copyright issues are in the process of being settled.

- the pointer must be changed to a crosshair when placed over the display area. It must be controllable with the keypad arrows at the resolution of a pixel
- at any time it must be possible to change the contrast and luminosity of each image interactively by moving the mouse in both directions while holding a certain mouse button
- one mouse button allows an image to be "dragged" within its frame
- a special mouse action (e.g. double click) should result in recentering of the image at current pixel position
- a "selective zoom" feature, by drawing a rectangle defining the closest zoomed field-of-view, must be available
- the same rectangle feature could be used as a more general way to select part of the displayed area for various operations
- if possible, all interface windows must be usable on a screen with VGA resolution (800x480), down to 4 allocated colours
- a "full-screen mode", in which the pixel raster occupies the quasi-totality of the screen, should be available. One may also think of reconfigurable widgets à la Netscape.
- A quick display at start of available hardware features (screen depth, resolution, hardware acceleration, OpenGL available, etc.) would be valuable

The target minimum requirements to run the program shall be (under Linux 2.4.x and above):

- 64MB of memory
- a PentiumII processor running at 500MHz
- an X11 server

The optimum configuration would have

- enough memory to fit a whole image in floats
- a fast 64bit processor (1GHz or more)
- a local, accelerated X11 server in 32bits (true-colour and alpha-channel)
- hardware OpenGL support (to be defined).

Phase 2

The phase 2 software shall complete the GUI and the visualisation engine from phase 1 with more advanced features:

- Colour Overlays:
  - region and shape editor, with lines, arrows, circles, rectangles and polygons as well as text, at an arbitrary resolution over the image
  - load/save overlay/region functionality
  - coordinate grid overlay
  - mini 2D plotting-tool for histogram and image profile displays

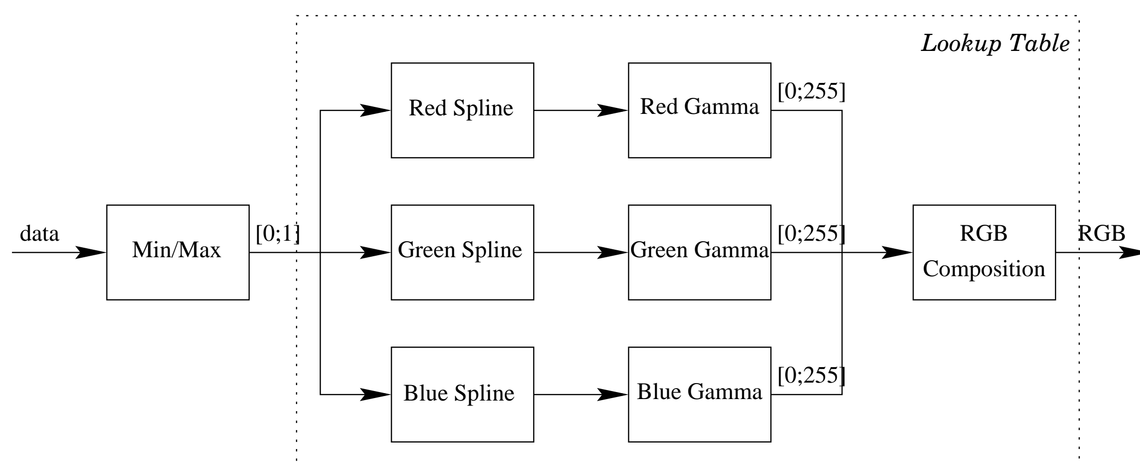

Figure 2.2: The Panorapix intensity-mapping pipeline.

- (simplified) contour plot option (à la DS9)

- Catalog overlay tool: functionalities similar to those of Skycat, including
  - automatic name-matching (CDS) queries
  - region-based image queries (DSS, CFHT, HST,...)
  - source catalog queries (GSC, USNO, SIMBAD,...)
  - support for local catalog tables in ASCII or FITS format
  - alternate highlighting/selection either on image or on the source list
  - configurable symbols, symbol colours and sizes

- it should be possible to use the overlay and catalog features without actually having an image loaded

- Import from image servers: the possibility of optimized interfacing with image servers (download low resolution image, send back coordinate infos) must be investigated

- Image analysis:
  - 2-dimensional Gaussian and Moffat semi-automatic profile-fitting tool, providing best-fit position and shape parameters for any hand-picked source in the image
  - aperture photometry tool, in a user-defined circular aperture for any hand-picked source in the image; this implies robust and automatic estimation of the local background level
  - geometric measurements (distance, angle to the north-south axis)

- Advanced export features:
  - possibility to export, either the displayed area or the full image at an arbitrary resolution, to popular formats including FITS (at least BITPIX=-32), TIFF (B&W and colour), compressed if possible, with 16 bit option, JPEG (with adjustable quality), GIF, PNG, Postscript level2 (B&W and colour) with "encapsulated" option, and MPEG/Animated GIF animation (cycling through the frames) with adjustable frame rate
  - the gamma parameter can be changed for all export formats

  - "negative" option for all export formats
  - anti-aliasing option (especially when the output resolution is less than the image resolution) for all export formats
  - possibility to superimpose the overlay for all export formats

- Interfacing with other software: an interface port to control at least the most basic operations of the software:
  - load/save/update/control image, catalog or overlays
  - send/receive a part of an image
  - draw shapes/write text on the overlay
  - change display parameters
- FITS header keyword search and display tool
- interface to manually add/edit the FITS WCS parameters of each image/extension (to be defined)
- possibility to superimpose frames using transparency and other linear combinations, allowing for RGB compositing, or image subtraction (to be defined)
- Geometric matching with image remapping to a common WCS projection (à la DS9)

Phase 3

In addition to the necessary debugging/maintenance of the existing code, and implementation of not-yet foreseen functionalities, phase 3 shall be spent in investigating integration as a web-browser plugin for remotely examining arbitrary parts of large compressed images in a grid environment with distributed data.

2.1.2 Multidimensional vector data

2D or 3D?

The most "science-ready" output of a survey pipeline consists of multidimensional vector data: the catalog. In feature space, each point corresponds to an individual detection. For both quality control and scientific assessment, a powerful visualisation tool for such data is mandatory.

Compared to the traditional 2D plotting packages, programs that use 3D vector or particle projection provide a virtual extra-dimension that proves extremely useful in many cases, for instance with magnitude-colour-colour diagrams or magnitude-colour-stellarity. A coarse 4th dimension can be added using colour coding, and a 5th one can be obtained through animation, although the latter proves to be mostly useful for virtual flyby's or revolution around 3D point clouds.

Display engine

Given the amount of source data produced by wide-field instruments — a single exposure yields typically 10^5 sources —, the display engine must be quite efficient. For realtime display, reasonable ($\geq$ 10 fps) frame rates can only be achieved through 3D hardware rendering. Thanks to the rapid development of 3D gaming, cheap 3D hardware solutions has become widely available on PC. Actually, *all* graphic cards found on modern PCs come with excellent 3D capabilities: in theory they are able to handle from 10^6 to 10^8 vertices or particles per second. There is now both software and hardware OpenGL support in Linux (see, e.g., `http://www.mesa3d.org` or `http://www.nvidia.com`), and Windows, so a 3D plotting tool is essentially a matter of fairly high-level software development.

Concerning a web-interface: apart from the VRML standard, there is not yet any really efficient way of making real-time 3D visualization work through a web-browser. Hence the possibilities of VRML concerning large data-sets should be investigated. A proprietary OpenGL plug-in tool might prove necessary.

A tour of existing OpenGL solutions

Commercial packages with OpenGL support for 3D-plots available on Unix/Linux include

- MATLAB (`http://www.mathworks.com`): a standard in the industry; although it does much more than this, it is quite efficient at manipulating 3D plots with less than 10^6 particles.
- IDL (`http://www.rsinc.com/idl`): another standard package popular among physicians. We have not measured its 3D plotting performance. It is probably similar to that of MATLAB
- AMIRA (`http://www.tgs.com/index.htm?pro_div/amira_main.htmm̃ain`): this product is fairly popular in medical applications.

Free packages available on Unix/Linux include

- XGOBI/GGOBI (`http://www.research.att.com/areas/stat/xgobi`): although it does not seem to have OpenGL support, it is quite fast for displaying large data sets (up to 10^6 particles on a fast machine) using an isometric projection, and very easy to manipulate.
- OPENDX (`http://www.opendx.org`): This free package features very high quality rendering and offers an impressive set of functions. Thanks to its support of OpenGL, performance is very good (up to 2.10^6 Phong-shaded-triangles/s). Unfortunately, it does not seem very comfortable with very large datasets, and the learning curve is unusually high (it requires a few hours for a beginner to create a decent scatterplot).
- PERLDL (`http://pdl.perl.org`): The PerlDL library has OpenGL support for 3D plotting. Although the display is quite crude, it is exceedingly easy to use and performance is top notch: we measure 10^7 points/s on an nVidia GeForce3 Ti200 graphic card)

2.1.3 Graphical database query tools

Most of the indexed fields in the pipeline/science database are populated with continuous numerical values; those can be used as coordinates in parameter space. A graphical interface tool therefore provides a particularly efficient mean of querying (groups of) data for all kinds of purposes. This kind of data exploration tool is typical "Virtual-Observatory technology".

Query by sky coordinates ("Astrographical query")

The most intuitive way of querying data is to use sky coordinates, by clicking on a representation of the celestial sphere. Two approaches can be identified at this point:

Clickable maps through a web browser A good example of this concept applied at small angular scales can be found at `http://astrowww.phys.uvic.ca/grads/gwyn/pz/hdfn/spindex.html`: this page proposes a true-colour display of the Hubble Deep Field; by clicking over a given source, one gets access to a new page that lists various measurements that concern this source (Fig. 2.3). Such a page could be generated dynamically from the database. On a larger scale, the SDSS EXPLORE tool (`http://skyserver.fnal.gov/en/tools/explore/#` provides another example of clickable maps directly connected to a database: the user can "browse" celestial maps and have

access to detailed information on the nearest survey detections. The maps are dynamically updated as new observations get registered in the database (Fig. 2.3).

The clickable-map approach works on all existing browsers without the need for any additional plug-in. However, the display cannot be updated very fast while zooming and panning, which makes the query rather slow and uncomfortable in a context of intensive use.

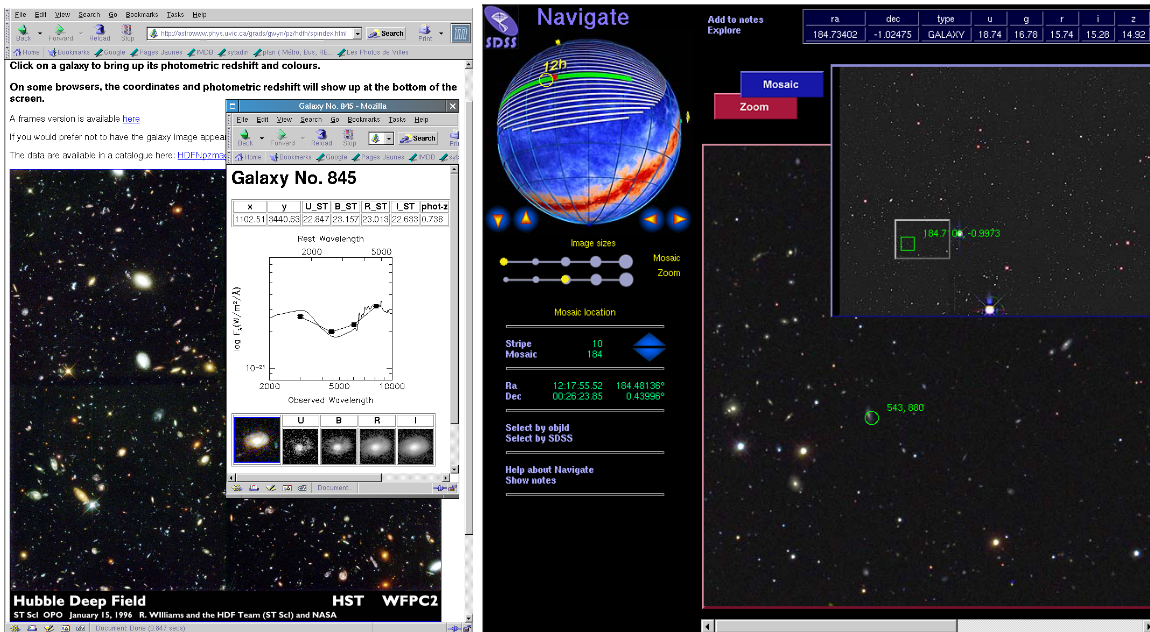

Figure 2.3: Two working examples of clickable maps linked to a database. *Left:* The Hubble Deep Field (courtesy S.Gwyn, UVIC). *Right:* The SDDS Explore tool (SkyServer website).

A dedicated OpenGL graphical tool Using OpenGL, it is quite simple to display interactively, in a window, a real-time projection of the night sky (Fig. 2.4). It should be easy to add clickable footprints of all observations that satisfy arbitrary constraints. Such a tool responds instantaneously to zooming and panning requests, which makes the query fast and comfortable for intensive use.

Basic graphical requirements

The purpose of the multi-dimensional visualization tool is to generate on demand plots based on pipeline or science data (obviously, it is operated through the graphical query tool described above). Such plots hold a significant amount of information, hence it is necessary for the tool to provide at least high-quality, printable plots (e.g. Postscript), and low-resolution ones for display through a WWW browser (e.g. GIF/PNG bitmaps). If a suitable 2D/3D-display engine is available (e.g. OpenGL plug-in), the low-resolution plots can be made scrollable and zoomable in real-time.

The plotting engine must have support for the following basic properties:

- Color encoding to highlight different properties
- Choice of symbols with arbitrary size
- Different fonts and character sizes and styles

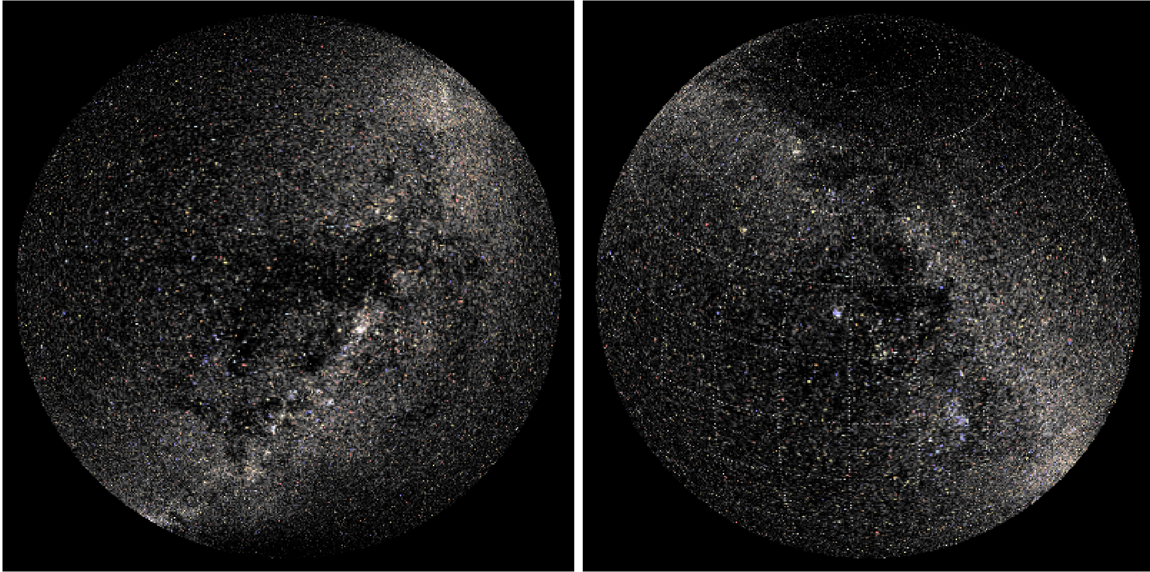

Figure 2.4: Interactively rotatable celestial spheres made with the `xplanet` software, using OpenGL (`http://xplanet.sourceforge.net`). The sphere is textured with a map generated from the Tycho star catalog (`http://maps.jpl.nasa.gov/stars.html`).

- Line thickness parameter

...and the following graph types:

- 2-D and 3-D scatter plots with dots
- Histograms
- Points with error bars
- Connected lines
- 2-D and 3-D density plots (e.g. for very large data sets)
- Celestial maps using popular sky projections, including Aitoff(all sky), polar equal area and tangent plane.

When dealing with very large data-sets, it is desireable to be able to use density plots (instead of scatter plots) with zoom-in capabilitity, which will let the user grab data-sets of a "reasonable" number of elements.

Browser integration

The web-interface could consist of a java-based catalog browser, allowing one to generate plots interactively from a catalog extracted from selected points and user-defined quantities. This java tool would download data from the server to the user's computer (in compressed mode) and interact with the database engine via CGI, to request other pieces of information.

2.2 Source extraction/ morphological classification tools

Source catalogs extracted from the astronomical images are required at various stages of the pipelines, especially for astrometric and photometric calibrations. Source extraction is also needed at the very end of the process for the final science catalog.

2.2.1 SExtractor

SExtractor, or Source-Extractor (Bertin & Arnouts 1996) is a simple, yet fairly robust source extraction program. Since its introduction in 1995, the SExtractor package has gained wide spread use among the astronomical community, which helped in developing new features and tracking down bugs. It is now used in many survey pipelines including TERAPIX (IAP) and EIS (ESO).

Besides the most basic features expected from a source extraction code (modeling and subtraction of the background, image filtering, thresholding and measurement), two main features of particular interest for survey pipeline operations have been added over the years:

- "Double-image" mode: detection is made on an image, while pixels from another image are used for measurements. This technique provides the most consistent colour measurements on registered images.
- Pixel-weighting: image pixels are weighted according to the values stored in "weight-maps" provided as additional inputs to SExtractor. This allows one to obtain constant detection reliability on images with variable background noise variance (this is almost always the case on wide-field survey images).

PSF fitting

More recently, a Point-Spread-Function fitting module has been added (PSFEx). Although it has already been used with great success in a few stellar photometry programs (e.g. Kalirai et al. 2001, Moreau et al. 2002), it is still experimental and needs further development. This module is essential to provide the best possible stellar photometry, especially in dense fields.

Correlated noise

The background noise on the resampled and co-added images produced by SWarp (or any other similar software) is correlated on small scales (2-3 pixels). This correlation will extend to larger scales once PSF homogenization is applied to the data. In the current implementation of SExtractor, background noise is assumed to be white (uncorrelated at all scales). Both the matched filtering (detection phase) and error estimates (measurement phase) are biased by noise correlations. A future version of SExtractor should automatically measure the background-noise autocorrelation function and take it into account at all stages of the source extraction process.

2.3 Quality checks/ human interface to data/ data mining

For the OmegaCAM instrument the *green* coloured boxes in Fig. 1.1 indicate quality control operations which are near the data aquisition (Paranal and ESO Headquarters). These quality checks go beyond the ASTRO-WISE operations. Many of the *yellow* labeled QC0 and QC1 quality checks and calibration file derivations will be operated by ESO-DMD in standard production mode, however these processes can also be executed and reconstructed within the ASTRO-WISE environment. All these quality control operations are extensively described in the OmegaCAM documentation and will not be repeated here.

The image pipeline also includes some standard quality checks which are described in the OmegaCAM documentation.

2.3.1 Data browser/ plotting / trend-analysis

Trend analysis addresses the instrument behaviour as a function of time. This information can and should be used by the quality controllers to model the instrument's behaviour aiming to inprove the final results and the end users (astronomers) to chose data which may meet their needs once enough data is stored in the archive.

For QC and astronomers it is important to have a highly flexible tool to select any subset from this data based on specific criteria, and perform basic operations such as the creation of plots and histograms in order to give a snapshot of the properties of any of the involved parameters (fig. 2.5). Flexibility means being able to set selection constraints on any database variable or combination of variables. The constrains should be combined with any logical operators in any fashion (ie. the system should be able to handle complex logical expressions).

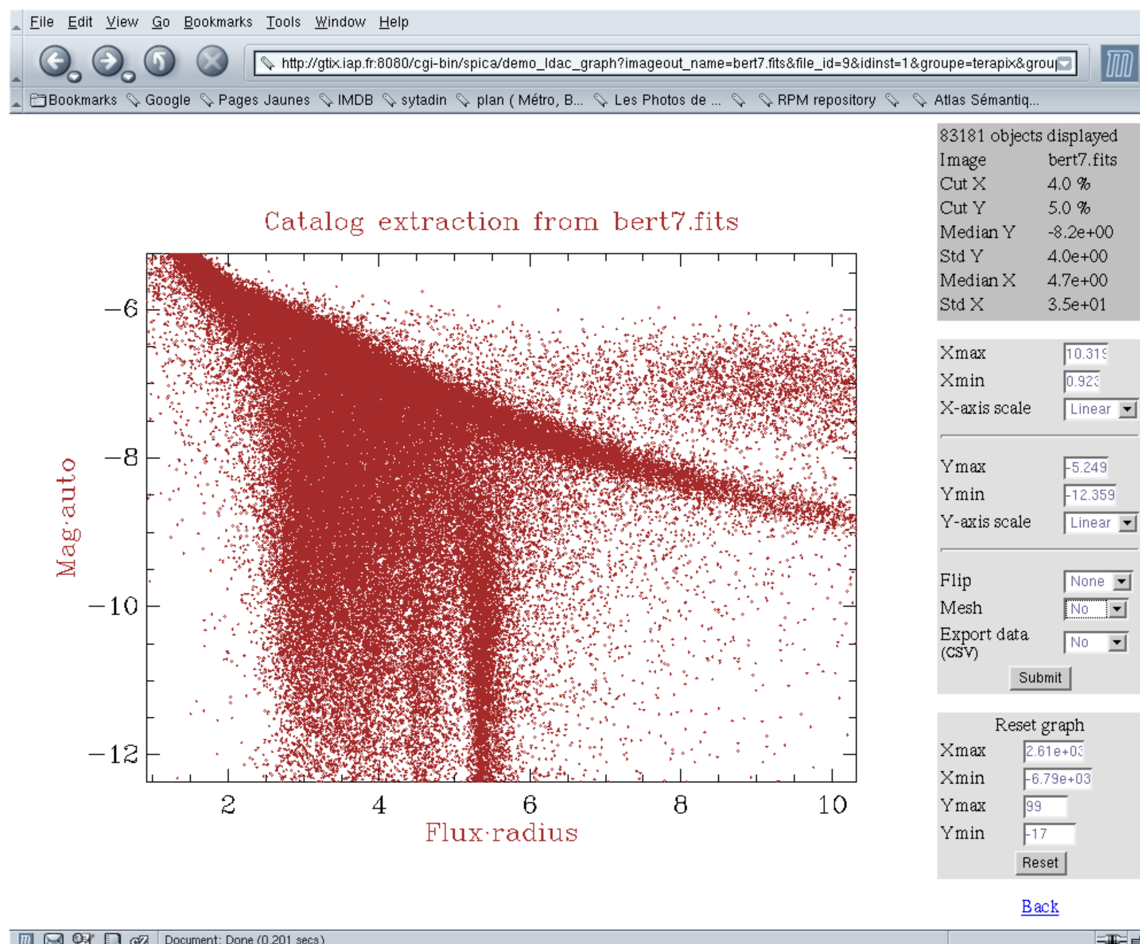

Figure 2.5: Interactive data visualisation prototype developed at TERAPIX. The user communicates with the database through the web server, and can plot in 2D any set of measurements from the extracted sources.

An interactive tool will be available to let the user generate plots involving database variables,

select/mark data on them, show the selected data in plots with different axes, and allow the user to recover all properties of these data points by invoking the database.

The ability to visualize 3 or more dimensions/variables (eg, using color) is expected, including the possibility to generate animations.

2.4 Near/inside database access tools

2.4.1 Basic set of queries

- basic mechanism

The basic mechanism for querying the database is provided by means of definitions in the Python scripting language, which passes the request to the database through SQL statements(see Appendix A). For example:

selection on time: flatfield = isotime('14-MAR-2002') < Flat.time_start < isotime('21-MAR-2002')

selection of sources within an area of the sky (box in α and δ)
sources = SourceList.within(20.0,-60.0, 24, -58)

complex selection:
SourceList.reduced_frame.flat.time_start < isotime('21-AUG-2003') < SourceList.reduced_frame.flat.time_end

The user can construct his own queries and send these to the db.

-some templates for frequent use

More frequent and more complex queries will be scripted and provided as a tool to the user, eg queries which make use of associations made in the database between different source lists:
for example, find red objects (B-R) when seeing is better than a given value, and the atmospheric conditions were very stable, requires access to:
– two different source lists which are associated by
– the sAssociate tools (providing a crosslink table) and
– atmospheric extinction calibration files
can be Python scripted and provided to the user.

A list of such standard queries is TBD, but will evolve continously during the project:

Summarizing, the database access tool or set of tools will provide the possibility to navigate through and retrieve from the database the different data products, or resolve queries on multiple dataproducts which have complex interrelations (like the coupling between atmospheric conditions and source lists in the example above).

2.4.2 Variability tool

The ASTRO-WISE system will allow the search for objects that show photometric variability on short time-scales, on the order of minutes-hours, provided by the images of the individual dither exposures.

This time domain has been very poorly studied in previous surveys, normally the search for variable objects is done only on the final mosaic images.

There are many software packages using either catalogs or image subtractions (like ISIS). Our scope is not to duplicate these programs but to create a simple, robust and efficient tool which runs automatically in the image pipeline.

Methods

The variability tool uses the catalogs that are already extracted from single dither exposures for astrometry, using SExtractor. The only additional requirement is that these catalogs should be deeper (S/N$\leq$5 instead of 50–100) in order to detect also faint objects. The next step will be to normalize the fluxes (or magnitudes) in the different catalogs in order to compensate for different air-masses and transparency variations. At this point a matching of the catalogs (which will be done through the Sassociate tool) will identify the objects which are photometrically stable or show brightness variations. The variability tool flags the variable candidates in the catalogs and optionally creates a simple light curve of these objects. The major challenge is to filter the large number of false detections that will be certainly produced.

2.4.3 Associate source lists

A tool (Sassociate) will be developed to associate source lists with each other, on the basis of the spatial relation in order to be able to inspect the peculiarities of a given source, particularly as it emerges from multiple exposures. The tool is meant as a workhorse for finding and identifying the same object on mulitiple images. This can be used to find excess colour objects, variable objects (e.g. supernova) or moving objects. The tool can be a Python script, a special purpose program, or a database query. The tool will store the associated sources in a so called *associate list* which contains database pointers to the original source list entries. Various prototypes of this tool have been explored, in the Objectivity environment and in the Oracle 9i environment, the latter using Oracle 9i spatial query components. Some problems with astronomical coordinate systems were encountered and are under investigation at Oracle Headquarters. As this tool as a major workhorse for spotting special objects in Terabyte volume source lists, its performance is crucial and is focal point of the prototypes.

An interface to this tool will be created which allows the user to select parts of different images and associate the sources found within the image selections.

2.4.4 Tell me everything within ASTRO-WISE tool (basic ingredients provided by WP1 and WP3)

When an astronomer identifies a certain object (in a source list or in an image), s/he should be given the possibility to trace any bit of information that was involved in the derivation of that source info, and s/he should be directed to all available information within the database relevant for the same sky position. A general tool will facilitate this. The tool uses a general functionality to support this at the database level. This functionality will form the core of tracing history and relations to other data items, which will also be used for inspecting and triggering on the-fly-reprocessing

Typically, the functionality will output the following data items identifiers and links: Table TBD

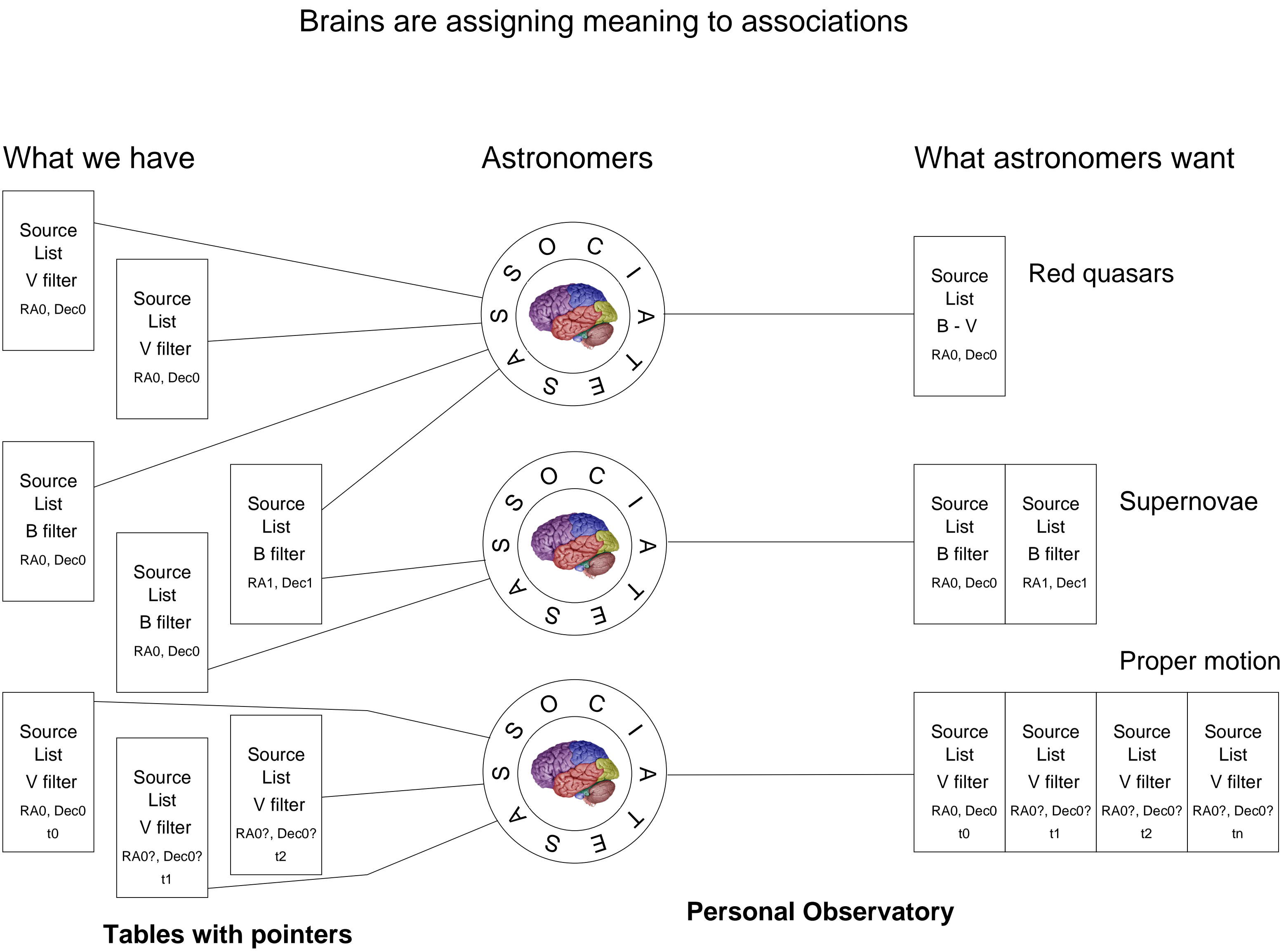

Figure 2.6: The ASTRO-WISE back end.

Property	nature	origin
Object type	scalar	from accumulated knowledge
Position	scalar	from reference image
Positions	array	from images involved if $\mu > 0$
V magnitude	scalar	from V image(s?)
Color indices	array	from same epoch images
Observing conditions	array	from each involved image
Reduction parameters	array	from the reduction DB
Light curve per filter	array	from any relevant observation
References (bibcode?)	array	Literature, may refer to relevant images
Thumbnails in various filters	array	From pre-processed images
Links to external DBs (VO-like)	array	???

Tying together images and catalogs should be achieved. Two approaches can be expected: **a)** description of objects found in an image, and **b)** retrieve images once an object was selected from the catalog space; both should be implemented. Images in several filters or epochs are particularly relevant to have a quick look at the properties of any object. Catalog query tools associated with Panorapix should be easy to implement, in which case a click on an object viewed by a user in an image (with a non visible position code) should pop up a window where all the properties of such object are shown.

It is likely that **b)** would be the starting point in many cases, but **a)** will become important once the user has a clickable image in front of his/her eyes, and what is described in the energy distribution section should also be incorporated into this functionality.

The tool will get history tracking data from the ASTRO-WISE database through a common interface. This interface can be used in a command line environment to show everything related to a certain object. Everything means all data (or pointers to that data) and software that was used to create the object, and that which might be used to recreate the object with a different setting. The same interface can be used in a graphical user environment.

In addition, an interface to perform introspection of the data in the database is defined that allows access to the metadata in the database in a portable (XML) form. This will make it possible to hide database details from the "tell me everything" tool, which might therefore be used with different databases or data providers that implement the interface.

There are a few possibilities how a tool like this will offer the required access to the database in a structured way.

For navigation through the ASTRO-WISE data model one can think of webbrowser or explorer type functionality for navigation in a browser or treelike way.

For more complex visualizations that require the combination of different parts of the datamodel (e.g. all source parameters together with flatfield, bias, atmosphere condition), one can think of a spreadsheetlike way of operation. A spreadsheet view of the underlying database allows the manipulation and visualisation of the data in a way that people are used to. Another benefit of such a view is that more complex operations can remain understandable.

2.5 Spectral Energy Distribution fitting tool

The USM will contribute a Spectral Energy Distribution (SED) fitting program that will allow the interpretation of multi-colour (from the UV to the NIR) photometric catalogues. The routine will deliver best-fitting spectral types, photometric redshifts, and lists of specific object candidates for follow-up observations. A large spectral library, including stellar, galactic and active galactic nuclei objects, and taking into account dust and intergalactic absorption, is convolved with the given filter bandpasses and matched to the dataset in a maximum likelihood sense. Additional a priori constraints (coming from possibly known object densities, etc.) can be taken into account.

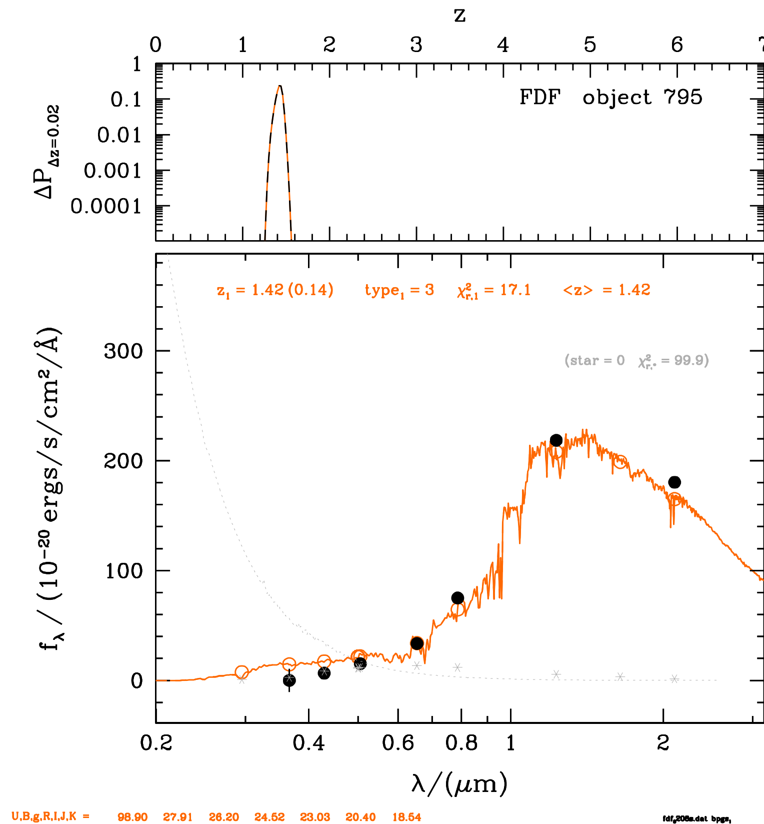

Figure 2.7: A sample output of the SED fitting routine as applied to a source in the FORS Deep Field (FDF). Based on flux in the U, B, g, R, I, J, and K-bands the program does a maximum-likelihood match of a library of template spectra (in this case of an elliptical galaxy) (bottom frame), to which is also associated a redshift probability (top frame).

The current version of this SED fitting routine exists as a stand-alone program, and will be modified to fit within the ASTRO-WISE data structure using an interface that is scripted with Python . For large source catalogues a direct access to the database is clearly required. Whether this is done within one of the visualization tools provided for ASTRO-WISE (see section 2.1.3 and, in particular, figure 2.3 which shows the SED output GUI created for the Hubble Deep Field), or whether an independent GUI is necessary remains to be determined. It should be noted that a reliable SED fit requires a sufficiently large spectral coverage (e.g. at least 5 filters for a reasonable source identification and spectroscopic redshift for most types of objects) and, therefore, these routines will only be relevant to a smaller fraction of the science programs proposed. Furthermore,

experience with the MUNICS survey and the FORS Deep Field revealed that adequate accuracy in the photometric redshifts can only be achieved with homogeneous PSF's across the different passbands sampled.

2.5.1 PSF Homogenization

In order to obtain accurate colors from comparisons between catalogs obtained in different filters, requires homogenous PSFs within the different passbands. Experience has shown that for small PSF differences ($<$ 30% in FWHM) a simple convolution with a Gaussian kernel suffices. One should take into account that such homogenization leads to correlated noise and that Sextractor will underestimate (photometric) errors in such images.

PSF measurement is a standard part of the image pipeline. We will provide a tool for image convolution and a tool to correct error measurements from Sextractor in homogenized images.

Chapter 3

WP3: Provide Federated database

3.1 Introduction

The aim of workpackage 3 is:

to design and implement a federated database environment to distribute administrative, and calibration data, as well as documentation and software over the participating sites in support of WP1 and WP2. In addition, the database will deliver or point data items such as raw and processed image data and will contain all source lists.

The ASTRO-WISE database will fully administrate all I/O of the various pipelines, irrespective whether they are run in "standard production mode" or "interactive analysis type of mode" by individual users.

ASTRO-WISE will process and store data on multiple sites, and these data should be transparently accessible from all sites. Hence the ASTRO-WISE database environment will be a **federated** database environment, distributing the data through a Wide Area Network (WAN).

A baseline objective is that all partners sites have equal access, privileges and authority over the database, though some control mechanism will have to be build in the system in order to guarantee data property privileges, in particular for what concerns the GTO (Guarantee Time Observations).

Compromises will have to be made in distributing *all* the raw data and *all* the processed science data. It is foreseen that different sites will focus on different sub-sets of the total datavolume. The database should transparently "know" about the various local archives, where the amount of replication of bulk data will be a continuous compromise between available network speeds and available local direct access storage volumes.

WP5 addresses the on-going ASTRO-WISE activities in expanding and upgrading direct access storage media. It is envisaged that most WAN services will be provided by the implementation of the db in WP3. Network speeds are assumed to be a free parameter in the present concept, but in the ideal system no data is replicated and the db always knows how to point and collect the relevant Input. In practice, replication will be needed preferably using smart caching, such as available in Oracle 8i. Whatever compromises between network access, replication and distribution of bulk data volumes are made the ASTRO-WISE db will control this process and register all data-items.
For the wide-field imaging data it is useful to discriminate between

1. **image data** (raw or processed) which in a few years will enter the 100 Terabyte regime, but which is dominated by noise,

2. and the **event** data of extracted source parameters, which exclusively contain useful data

items, for which nominal data volumes are of the order of 1 Terabyte/year, while exceptional programmes (galactic plane, crowded fields, monitoring programmes) might acquire Terabytes of event data in a period of less than a month.

The ASTRO-WISE database environment will store and provide access to:

1. All administrative data of the pipeline (WP1), including all process parameters and a full description of all processed data (meta data).
2. All "event data" produced by the pipeline (WP1) and tools (WP2), scalable to Terabyte useful "event" data.
3. All calibration data
4. All documentation and software
5. The WAN distribution of all raw and processed "image data"

Although it is not a priori necessary to use the same database technology to store the different kinds of data, **it is imperative that the logical relation between these data items is preserved at all times**. For example: a zeropoint, will be derived by a pipeline program, controlled by several parameters, from a number of source lists, obtained from standard star observations calibrated with a certain flatfield. Hence, this operation combines administrative data, sourcelists, calibration data and software. Only by combining all these data can we reproduce how the zeropoint was obtained.

In section 3.2 we discuss in detail the requirements posed by these different data items on the ASTRO-WISE db environment. We also discuss which capabilities the database environment could provide for these data items to facilitate and/or support the implementation of the software described in WP1 and WP2.

To facilitate implementation, ASTRO-WISE uses a single object model to represent all data, except documentation and software (in fact, the software *fits* the data model). This model would be most easily supported by a federated, object oriented database. However, to support science operations, one would like maximum flexibility in querying, which is much better supported by relational databases. These conflicting requirements suggest a hybrid solution, an object-relational database. In section 3.3 we discuss Oracle 9i, an object-relational database, as a back-end, for the ASTRO-WISE database environment. We pay particular attention to the federation aspects of this solution.

As stated previously, software and documentation will not be stored/distributed through the same object oriented database. One advantage is that this avoids the bootstrap problem (needing software to access a database to retrieve that software). In section 3.4 we discuss the implications of CVS, our method of choice to provide access to the software, both in terms of source code and in terms of bootstrapping the executables in the pipeline/db environment.

The interface/handshake between the CVS repositories as far as source code is concerned and the ASTRO-WISE db is outlined in section 3.4.

The distribution of raw data is made possible by the use of an **ingest** procedure which turns this data into persistent objects of the proper class and stores the corresponding file on a dataserver.

The ingest procedure ensures that the raw data is federated, meaning both the raw image data and the corresponding persistent object can be accessed from all sites participating in the federation (taking into account proprietary data privileges).

The procedure takes raw data, identifies it if possible, stores the metadata it reads from the file in the database and stores the file on the dataserver. If required, the ingest procedure also takes care of splitting and storing the splitted files.

3.2 Provide database engine which supports the following operations

3.2.1 Administration of pipeline operations as done under WP1

The administration of the pipeline comprehends administration for

i. All pipeline I/O, under version and time control. Python to OCI interface- see also WP1

ii. All pipeline modules including plug-ins.

iii. The free input parameters and input data for on-the fly reprocessing.

To achieve this goal the database implements the persistency as required by WP1. This means that there is a direct mapping between the objects that are marked as persistent by the pipeline and their instantiation in the database. More specifically, also each attribute that is marked as persistent in a pipeline object will have an identical counterpart in the database. Attributes that are not marked persistent in a persistent pipeline object are considered transient and are not stored in the database.

In addition to the objects as identified by the pipeline references between these objects are also stored in the database. That ensures navigation between objects that are linked by such a reference.

3.2.2 Storage for all source list data

Fast selection and manipulation of source list data is central to ASTRO-WISE . The database design has to support this. All source list data is stored as persistent objects in the database in the representation that allows the fastest retrieval and association—see WP2—possible. The source lists and the sources they contain are defined as persistent objects in the same way as persistent pipeline objects are defined.

There are several representations for source lists. In Oracle 9i they can be represented as nested tables, varrays or partitioned tables.

3.2.3 User programs, contexts, permissions

Several different users of the database can be identified:

- The pipeline operator who runs the standard pipeline.
- The astronomer who wants to run the standard pipeline with the latest standard calibration data.
- The astronomer who wants to run a modified standard pipeline, possibly with private calibration data.
- The astronomer who wants to run personal software with personal data.

To accommodate this the database and the ASTRO-WISE software have to be used following strict rules specified hereafter. The database and the standard pipeline will enforce these rules. Any personal software that doesn't adhere to the rules should have read access to the data, but only to data that are public. If such software produces results that should contribute to the ASTRO-WISE environment it can only do so by using the supplied interfaces and guidelines.

The contexts as defined in 1.1 are similar to Virtual Private Databases in Oracle. These appear to the user as single logical database but are in fact subsets of a much larger database. This means it can be used to impose access-restrictions by allowing a user access to certain contexts. At the same time this mechanism can be used to hide data that is not of any interest for a particular use.

3.2.4 Provide basic set of queries, including complex queries

A basic set of queries is provided that shields end users from implementation details. The reason for this is that even questions as phrased in simple English by an astronomer can require a lengthy and complex sentence in a standard query language. For the most common questions that arise from calibration or data mining precooked queries give easy and optimal access to the needed information. Complex queries involve combinations of data described in 3.2.1 and in 3.2.2.

3.3 Provide WAN for ASTRO-WISE

3.3.1 Introduction

The ASTRO-WISE database is seen as a single logical database system that is distributed over geographically distant physical database systems.

Ideally, transparent access to the logical ASTRO-WISE database should be provided by the database implementation itself. Although some databases come close in certain area of the federation aspect, currently no database exists that gives complete federated use over a WAN.

Some sort of middleware has to be created or used to simulate a single logical federated database[1].

3.3.2 The context of the database operations

In the logical database there is one single data definition for each class/type. This implies that for the different physical databases there is also one single data definition. Consequently, this has to be enforced by the database interface or middleware.

Insertions into the database are local operations and requires write/insert permission for the physical database that is local to the user. Retrieving from and searching the database can be remote operations and require read/select permission to the physical databases that are both local and remote to the user. In figure 3.1 the situation is shown for a single node in the federation. All other nodes have identical behaviour.

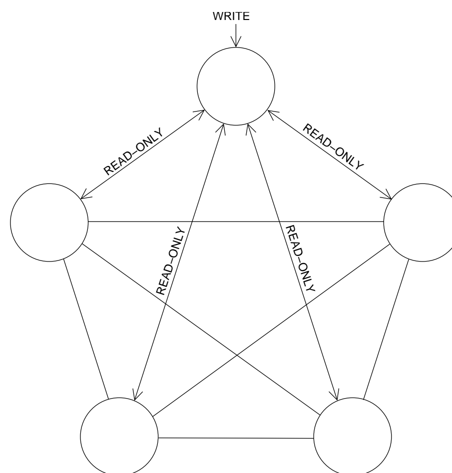

Figure 3.1: ASTRO-WISE geography and I/O - all nodes behave identically.

[1] See also http://www.eu-datagrid.org/

It is interesting to note that this way of operating is very similar to the world wide web. The main additions for database operations are that searching occurs in parallel across the different nodes and that results from searching different nodes can be *combined* to produce a single answer.

3.3.3 Network speeds

Considerations

– distributing all OmegaCAM data within 24 hours as taken by the telescope during one night would require a modest average network speed of 5 Mbps.

– realistic top network speeds should be of the same order as disk access speeds, 200 Mbps. (25 sec for one OmegaCAM image)

These considerations lead to various operational models. Having a reliable network with unlimited speed makes all machines appear as local machines. For most I/O-bound operations "unlimited network speed" is roughly equivalent to maximum harddisk I/O speed. This is by far the simplest operational model since there is no distinction between a collection of local machines and a mixture of local and remote machines.

In the case of modest networks speeds the major question is which data needs to be copied from the remote to the local site and how to minimize this amount in order not to saturate the network. When remote data is needed often it is worth keeping a local copy. For some data it is clear beforehand whether it will be accessed frequently. In that case, it could be worth making copies on a regular (e.g. nightly) basis. Clearly, the most general and automatic way of copying data would be to have some caching mechanism. One has to keep in mind, however, that such a smart mechanism might not use available bandwidth if the caching is not triggered by a process requesting data. Even if a cache is in use, it is useful to see how to optimize the use of available average bandwidth by selective automatic copying of data.

3.3.4 What are the options?

As explained, the optimal solution would be that the database can itself be distributed. A system that mimicked this by means of an intelligent caching mechanism between databases was available in the Oracle 9i Application Server.

Version 9i-second of Oracle introduces STREAMS, a concept that makes it possible to copy data in a smart way from Oracle databases, databases from other vendors or even data that is not part of any database. It will be very useful to find out how to mould streams in a way such that the standard pipeline and other database operations work in a seemingly federated way.

All versions of Oracle have several ways of replicating parts of databases in a coarse manner.

- Transaction logs can be used to synchronize with all the additions that have been made to all remote databases in the federation.
- Transportable tablespaces are another mechanism to copy parts of the remote databases to the local database.
- Views can be used to make data on the remote databases appear as local data and materialized views can be used to instantiate such view as physical copies in a local database.
- There may be existing - now overlooked - mechanisms or technology to give the impression of a single logical database.

The DataGrid tries to solve the connection of distributed databases in a more general way by providing middleware and interfaces to connect existing database. Since this is a more general approach, it is also valid for the ASTRO-WISE database. At the same time it is clear that having a

single system can provide a more straightforward and easy solution than is necessary for a hybrid federation.

Given that a transparent federation does not exist, the other options depend on trade-off between simplicity of implementation, simplicity of maintenance network speed, reliability of the network all in combination with performance.

3.4 The database interface

Access to the database is provided through Python . This means that Python is used for both the Data Definition Language and the Data Manipulation Language.

The database interface maps class definitions that are marked as persistent in Python to the corresponding SQL data definition statements. Likewise, any instantiation of a class is translated to the corresponding SQL data manipulation statement. In case a persistent Python object is retrieved from the database the corresponding SQL selection statement is performed.

It is important to note the following points. All translations from Python to SQL and vice versa are done transparently and on the fly. The user merely marks a class and all attributes that should be made persistent as such. Inheritance is preserved in the database; classes that inherit from a persistent parent class in Python map to types in SQL that inherit from the SQL counterpart of that parent class.

The database interface is described in more detail in Appendix A.

In addition to the Python objects themselves also their method need to be made persistent. This means that a connection between the Python source that was used to create an object and this object needs to be made. Like for the object data, also for the object methods a mapping between Python and SQL could be created. Given the complexity of automatic translation from Python methods to SQL this is not feasible or desirable. Also there is the need to keep track of Python sources that are not methods of persistent objects. Therefore the connection between persistent Python objects and the Python source used to create them is handled in a different way.

The source text is stored in a CVS database described in the next section. CVS knows about versions of source text, but not about the meaning of its content. I.e. it does not know that a couple of lines of text together form a method in Python . It is the connection between this method and the created objects that we want to keep track of. By assigning versions to methods and keeping track of which methods created an object its history can be traced at the best level of detail. Relying on the version of the complete source text in which the method is defined is not sufficient.

3.5 CVS database for federating

All source text, documents, web pages and manuals are stored in a central repository under CVS.

At the Kapteyn Institute a fileserver hosts the central repository. All partner sites “commit” and “update” under the CVS protocol.

All objects in the federated database as created by the ASTRO-WISE software are created with the source text as stored in CVS. For reprocessing it is necessary that the version of the software with which an object is created is known. Versions of CVS source text shall have a counterpart in the federated database. To achieve this it is sufficient to store the location and version of a source file in the federation.

For reasons of consistency a checksum is calculated for each version of a source file in CVS. This checksum is stored in the federation along with the location of the source file and its version. Introspection of a source file allows the detection of changes in the version the source file thinks it has, e.g. caused by editing of the source file.

3.6 Interoperability with WP2

3.6.1 Introduction

Interoperability of WP3 with WP2 is regarded an extension of interoperability with WP1. The interface between WP1 and WP3 is defined in a way that allows it to be extended, hence accommodating WP2 in a natural way. Anything that applies to the WP1 interface applies to the WP2 interface.

3.6.2 Data types

The database interface shall provide all basic and complex data types that are needed for WP2. In addition the database itself will store the data types in the most appropriate way. The database shall handle all translations between the database type definition and its counterpart to Python . The database interface is foreseen to handle all data types already needed by WP1. These are basic types such as integer, floating point, string and time and arrays of those and more complex types that combine (arrays of) basic types (i.e. class definitions). Additionally references to complex types are allowed.

For WP2 another data type, a "virtual data type", may be useful. This type is not stored, but calculated on the fly. One can think of coordinates in different systems or epochs or projections. A more complex example would be a virtual data type that is defined as the combination of existing types. The database interface should know that for this virtual type the data is not stored.

A data type might have a unit attached to it for WP2, this is undefined in the WP1 context.

3.6.3 Querying

As for WP1, Python is used as the main language for queries. In addition an interface is defined that accepts SQL directly. This interface is only for the end-user. Any tools or software layer will have to use the standard Python database.

The database will support spatial queries as required by the core of ASTRO-WISE tools. It should be possible to answer queries to find all sources and their parameters within a certain area on the sky. More complicated questions, like e.g. trying to find the areas of the sky where the source density is highest or lowest should be solvable by the database. The (spatial) association of two or more source lists is considered a complete subject in itself and is covered as such by `SAssociate`.

Examples of five line scripts (5LS) which provide powerful queries in near pipeline environment are given in Section 1.2.

An interface to allow introspection beyond what is already possible for WP1 allows the creation of more general WP2 tools. This obviates the need for intimate knowledge of the data that a WP2 tool tries to visualize.

3.6.4 Representation

Besides Python and in SQL, it should be possible to represent both the data definition and the data itself in XML. One can see XML as just another data definition and manipulation language. Because of that and because the database interface already handles Python and SQL representations, the database interface is a suitable place to handle conversions to XML.

XML will be used to interface with systems external to ASTRO-WISE . To expose data to VO's the standard interface defined for VO's is used. The standards and tools to exchange data are discussed by the Interoperability Working Group[2]. Regardless of the interface, a context (see 1.1) will be used to define which selection of the data is visible to VO's.

[2] `http://cdsweb.u-strasbg.fr/Interoperabilitywg.htx`

3.6.5 Tell me everything within ASTRO-WISE

The tell-me-everything-within-ASTRO-WISE tool provided by WP2 and described in the WP2-tools chapter relies on functionality provided by WP1 and WP3. The federated database achieves this by storing

- All persistent objects.
- The connections/links between the persistent objects.

The persistent classes defining the persistent objects and the relation between them are defined by WP1.

The Oracle 9i implementation of the database interface uses user-defined types and references to provide the needed tell-me-everything functionality. User-defined types and references are Oracle concepts. The user-defined type is used to define a persistent class in the database. The name of the user-defined type is the name of the persistent class and the name of the persistent attributes become the names of the attributes in the user-defined type. The type of the persistent attribute translates to an equivalent atomic Oracle type. A persistent link attribute in particular translates into an Oracle reference to a user-defined type.

To store the instances of the user-defined types a so-called object table is used. This is a table consisting of objects only (unlike conventional tables in relational databases, which consists of columns and rows).

In addition to the type and the object table an object view is created for each persistent class. This object view does not contain objects. Instead this view defines a selection of the objects in the corresponding object table and *ALL* objects in tables of persistent class that is derived from the corresponding. For example, if classes X and Y are both derived from Z, then the object view for Z will show all instances of both X and Y.

The query syntax in Oracle 9i supports the object model also when selecting data in the database which means there is very close match between the Python query syntax and the database syntax.

Both the Python interface and the database support resolution of links/references as required or in queries.

In figure 3.2 a tree-like view is shown of all ancestors of a reduced scienceframe. Leafs in the tree correspond to attributes of persistent classes and branches correspond to links/references to objects. By expanding a branch one can have a more detailed look at the referenced object. In the figure this is done for the 'bias', 'filter' and 'image' attributes. When all branches are expanded the complete history of an object is shown. This is true for all objects in the database.

Object Viewer

Attribute	Value
AIRMEND	0.0
AIRMSTRT	0.0
DATE	1999-06-13 02:57:32.00
DATE_OBS	1999-06-13 02:44:58.00
EXPTIME	719.9138
filename	reduced.tmp
globalname	None
object_id	'BDB51DD4ED482A29E0307D81790666E2'
process_status	0
quality_flags	0
+ astrom	
+ astrom_corr	
- bias	
creation_date	2003-05-14 12:48:12.00
filename	2000-04-21cal541_ccd50.fits
globalname	None
object_id	'BDA03AB94FAD119FE0307D81790659EC'
process_status	1
quality_flags	0
timestamp_end	2000-04-22 12:00:00.00
timestamp_start	2000-04-21 12:00:00.00
+ raw_bias_frames	
+ chip	
+ imstat	
+ instrument	
+ observing_block	
+ prev	
+ process_params	
+ chip	
- filter	
central_wavelength	4562.52
has_fringes	0
mag_id	JohnsonB
name	#842
object_id	'BD69A157307916ADE0307D8179066F85'
+ flat	
+ fringe	
- imstat	
max	17394370.0
max_x	1013
max_y	3005
mean	475.018469777
median	325.891143799
min	7547381.0

Figure 3.2: History of a single reduced scienceframe

Chapter 4

WP4: Provide parallel processing power to ASTRO-WISE

4.1 Introduction

The aim of WP4 is to provide the necessary computing power to the prime data centers and install and operate Linux-Beowulf 32 Gigabit PC parallel clusters.

4.2 Work description

The distributed nature of the astro-wise consortium naturally provides different environments and different boundary conditions to start from. Initially the expertise build at the different sites will yield different ideas of how to build a parallel cluster. Therefore at the beginning of the project a diversity of implementations will be available for testing and evaluation.

Currently the below mentioned sites have an operating configuration which will be upgraded and export to the other main sites according to the results of the investigations mentioned in the next section.

A generic hardware configuration will be described in general terms based on performance (processing speed, network speed, disk space requirements, infrastructure layout, etc.). It will not be a precise description of which hardware vendors, types, or versions a parallel system should consist off.

4.3 Investigations

The boundary conditions for processing the data in the ASTRO-WISE context are well established and will not evolve dramatically during the course of the project. The evolution of computer hardware during the project period, however, will be dramatic. What now seems barely possible will be easy, by factors of four, by the end of the project. We need to explore a working parallel processing environment based on the current hardware possibilities, but will incorporate the expected hardware evolution. To facilitate the exploration the following important aspects concerning the choices of hardware need to be addressed.

Currently and even more so in the future the ASTRO-WISE data processing is I/O bandwidth limited. Given the curve of growth in terms of CPU speed and I/O bandwidth this limitation will remain. There are several aspects to the I/O bandwidth problem dealing with different parts of the computer hardware. The three major parts to consider here are network speed, machine bus

throughput, and memory speed. Each has its own limitation and speed enhancements with time. For the ASTRO-WISE data flow the I/O speed of each part must be evaluated, future developments must be extrapolated from past experience and implementation choices must be made. Especially the expected future changes and possibly shifting of relative bottlenecks of the different I/O bandwidth components need to be addressed.

To evaluate the possible hardware implementations the processing boundary conditions must be clearly defined. To do this several aspects of parallelization of the software need to be investigated. Important aspects concerning the parallelization choices of software involve questions to the level at which the distributed processing will take place. The simplest form of parallel processing is to run individual pipelines on individual machines. This so called embarrassingly parallelized processing is very simple in terms of its operation, but takes little advantage of the intrinsic parallel nature of some of the pipeline processing steps. A somewhat more complex, yet still quite simple form of parallelization can be obtained on the individual CCD level, where automatically 32 to 40 different parallel processes can be defined.

Some pipeline processes are, by the nature of the algorithms used, not embarrassingly parallel and therefore either need to be executed on a single CPU, or need to be recoded to allow distribution across a cluster of CPU's. With the parallelization at the CCD level and the deeper algorithmic parallelization, several software and hardware aspects dealing with system resources and synchronization of pipeline streams become important. The cost of software development to support higher levels of parallelization in terms of the overall pipeline speed up needs to be weighted against the costs of acquiring additional hardware to facilitate a similar speed up, if ever this is possible given additional hardware.

To make parallel processing automatic and without human intervention pipelines software development must be done to deal with the resource management in such conditions. All resources that have direct impact on the processing speed, see above, need to be incorporated. In addition, possible variations in the implementation details must be coped with.

4.4 Relation with WP5

The concepts in this work packages are strongly correlated with those of work package 5. Because data needs to be fed to the processing hardware the data flow to, from and within the processing hardware is a vital part in the total execution time of the pipeline components. This work package concentrates on the processing aspects of the data reduction task but should yield boundary conditions for the data ingest/output speeds as well.

4.5 AstroBench

As outlined in previous paragraphs, in order to make the best hardware choice it is necessary to benchmark different hardware solution for specific astronomical application. Also, in order to identify bottleneck and so try to optimize both hardware and software, a profiling tool to monitor the usage of the different hardware component during the tasks execution is needed. To address these requirements Astro-Bench has been developed to perform a set of astronomical application benchmarks in order to estimate the performance of single/multi processor/s platforms. At the moment four astronomical tests are done: a production of a master bias using wfi-masterbias (eclipse) out of five WFI@2.2 bias frames, a production of a master flat field out of five dome flats and five sky flats using wfi-ff., a run of SExtractor on one wfi image and a run of SWarp on four wfi images. In this last test only background subtraction and re-sampling are executed. The tool also give the possibility (for linux platforms only) to profile the runs in the sense that the /proc files are sampled at a settable frequency. In this way it is possible to know the CPU(s) usage, to measure

the network and the disks traffic. Of course other applications can be easily added to Astro-Bench as well as we plan to include in the tool the whole ASTRO-WISE pipeline.

The software has an html based graphical interface as shown in figure 4.1 and can be freely downloaded at the URL: http://www.na.astro.it/beowulf/.

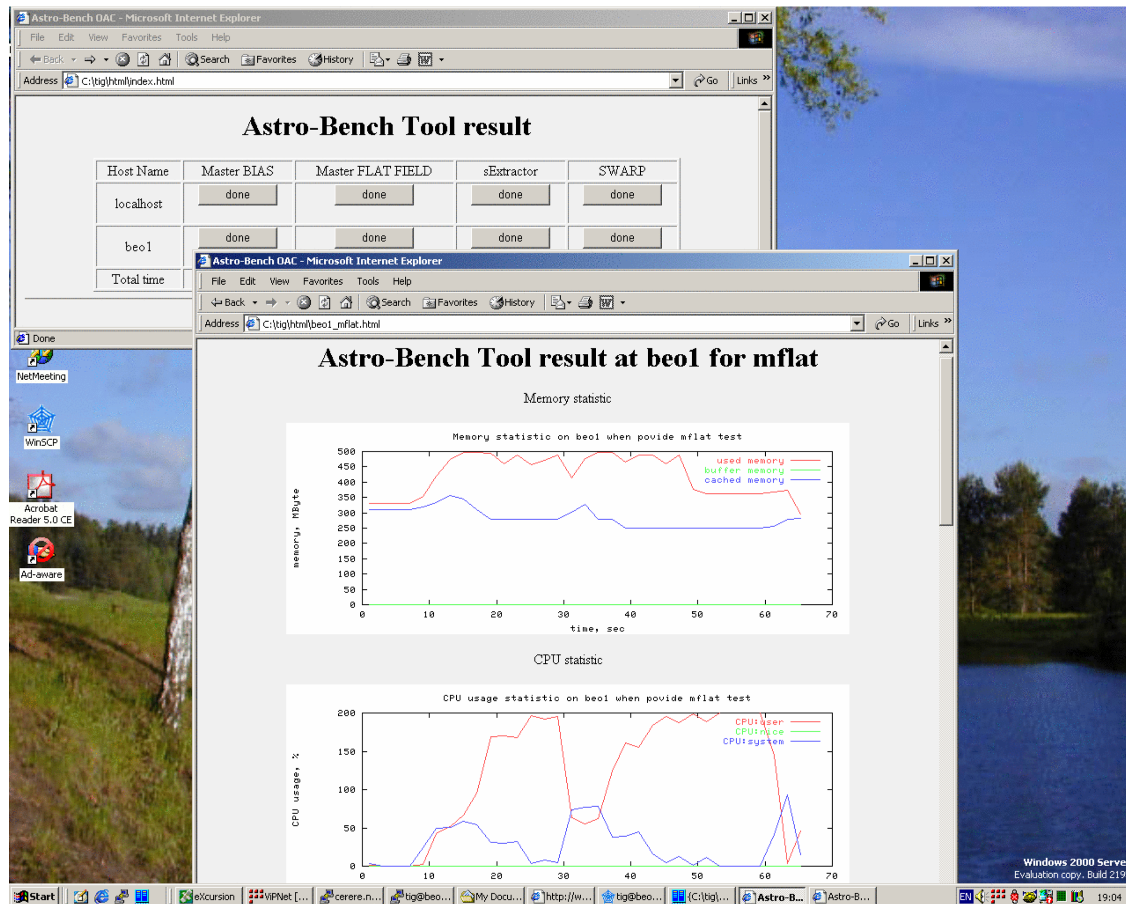

Figure 4.1: Snap-shot of Astro-Bench graphical interface.

Astro-Bench results obtained with the beowulf cluster at some of the astro-wise data center are reported in table 4.1

4.6 Hardware at Groningen / Leiden

Beowulf system at Groningen

To run the OmegaCAM pipeline a 32-Processor cluster (16 node Beowulf system, each node having 2 processors) linked at very high speed (1Gbit/s) is optimal. Such a system allows the processing of huge data volumes which is necessary to handle the expected data stream of the instrument.

Currently a fileserver connects the harddisk towers to a dedicated Oracle server and an 8-node computer cluster at the Computingcenter of the University of Groningen through a dedicated glasfiber link over a distance of 300 meters. In this experimental setup the astro-wise pipeline runs on the remote cluster. The metadata is obtained from and updated on the local Oracle server,

Institute	Master Bias	Master Flatfield	SExtractor	SWarp
NOVA	8	23	5	68
OAC	34	82	17	158
USM	19	43	11	142

Table 4.1: Astro-Bench results (in seconds) obtained with the astro-wise data center beowulf clusters

the raw data is retrieved from and the reduced data is stored on the local harddisk towers of the fileserver (see WP5, Data management and Hardware at Groningen). Each node of the beowulf cluster at the computing center has a 1.7GHz Pentium IV processor with a cache size of 256 kB and 512 MB internal memory with a 20 GB internal IDE disk.

4.7 Hardware at Paris

Beowulf system at TeraPix

- hardware type client nodes Bi-AthlonXP at 1.53 GHz, 2GB or RAM at 266MHz
- number of processors 10 processors total, divided amount 5 towers with dual CPU's
- network type, layout client nodes Two interfaces, 1 Gigabit, 1 Fast Ethernet. The Fast Ethernet is form external network connections and to provide a command network. the Gigabit network is meant for data transport.
- hardware type Server node Bi-AthlonXP at 1.53 GHz, 2GB or RAM at 266MHz
- network type, layout server node Four 1 Gigabit and 1 fast ethernet interface. The four gigabit cards each connect to one client node. The fast Ethernet card is part of a star-node network.
- storage capacity and distribution Client nodes each have 2 x 400 GB RAID arrays and the server node has 840 GB of RAID5 disk
- estimate of cost acquisition Computing units Euro 24k, Server unit Euro 7k, Auxiliary equipment Euro 3k, which gives a total of Euro 34k.

4.8 Hardware at USM

Beowulf system at München

The following Beowulf PC cluster is currently installed and operating at the USM:

- Compaq ProLiant Rack-mounted PC cluster based on Pentium III 1.1 GHz processors, each with 3 GB RAM.
- 16 processors total, divided among 8 racks with dual CPU's. Our rack and Myrinet switch has been chosen to allow expansion to 32 processors in the future.
- the network layout is based on a Myrinet switch with fiber ports for inter-node communication at 1.5 GB rates.

- storage is momentarily centrally distributed on the master node and comprises 28 x 72.8 GB = 2.04 TB in a RAID configuration. Furthermore, we have 4 x 815.5 GB spread over 4 workstation PC's. Our rack and RAID controller will allow a future expansion to 10 TB total storage.
- the hardware contract was awarded to a local company (Lantec) and purchased as a "turn-key" solution. Hardware costs totalled 123,442 Euro, with a further 4,090 Euro for installation, maintenance, and an extended warranty.

Ancillary Hardware:

Aside from the Beowulf PC cluster, we have also installed 4 individual PC workstations that will, eventually, be connected to the cluster to give the user a local storage and work area. Each PC has a RAID array with 815.5 GB at RAID level 3/5.

4.9 Hardware at Capodimonte

PartenoBEO Beowulf cluster at OAC

Hardware description

1 Master with two processors + 8 Nodes with single processor

Master configuration:

- CASE cabinet rack mount 4U 19"
- MB TYAN THUNDER HE SL (S2567) dual processor
  - Up to 4GB PC133
  - Four 64 bit 33 MHz PCI
  - Two 64 bit 66 MHz PCI
  - Dual-channel Ultra 160 SCSI
  - Serverset HE-SL chipset support 100/133 MHz FSB
- CPU two PIII 1 GHz
- RAM 1 GB ECC registered
- DISKS 2 HD 75 GB eide Model=IBM-DTLA-307075 + 1 HD 18 GB IBM Model: DDYS-T18350N
- NET Gigabit Ethernet module INTEL PRO/1000

Slaves configuration:

- CASE cabinet rack mount 2U 19"
- MB TYAN TIGER 200T dual processor
  - Up to 2 GB of PC100 or 1.5 GB of PC133
  - Five 32 bit PCI
  - Promise FastTrak 100 IDE RAID
  - support for Tualatin CPU

  - Dual LAN controller
  - CPU one PIII 1 GHz
  - RAM 512 MB ECC registered
  - DISK 1 HD 40 GB eide IC35L040AVER07-0

NETWORK

The master is connected to a network switch through Gigabit ethernet, from the switch to the nodes there are fast ethernet connections

SWITCH

- Allied Telesyn 24 ports 100TBase
- 1 port Gigabit Ethernet
- Number of processors: ten PIII's 1 GHz
- Network type, layout: 1 Gigabit ethernet to the master - switch - fast ethernet to the nodes.
- Storage capacity and distribution: 168 GB on the master, 40 GB disk on each node
- estimate of cost acquisition for Master,nodes,network,switch,KVM,switch,rack: 16 KEU
- estimate of cost implementation/maintenance: it is a bit difficult to estimate such costs. The installation in total lasts few days of two people. The R&D time to extend to the 32 nodes configuration is quite difficult to estimate and than the maintenance when the system and the pipeline will be stable will require a person not full time

On date 22/04/2002 the order for an upgrade of the Beowulf has been placed. It consists on putting the second CPU on the nodes, changing the case of the master from 4U to 5 U and add a RAID controller 3ware 7850 with 7 EIDE hard disk of 40 GB each. A seconf master with the same configuration of the previous one has also been ordered with a second Gigabit module for the switch. In total the cluster will have two dual processor masters with 350 GB of disk space each, both connected with Gigabit ethernet to eigth dual processor nodes. With the up grade to two masters we plan to test parallel file system with Gigabit ethernet. Promising results have been achieved using PVFS with fast ethernet.

The up-grade cost is about 11 KEU.

Planning

OAC plans to extend the present cluster to 32 nodes.

Chapter 5

WP5: Provide data storage to ASTRO-WISE

5.1 Introduction

The aim of WP5 is to provide the necessary direct access data storage to the prime data centers; typically 1 Tbyte/ site direct access, up gradable to 10 Tbyte/site within two years.

5.2 Work description

Comparable to WP4 the distributed nature of the ASTRO-WISE consortium provides different environments and different boundary conditions to the storage requirements. However, the fact that data may reside on any or all of the different nodes in the consortium may be appealing to the concept of a distributed environment but must be evaluated against the data access speeds of data retrieval in terms of data reduction or archival research.

The development of a direct access data storage implementation at first was a diverse process where different nodes investigated the storage implementation that is best-suited to their parallel environment. Experience and in depth investigation of the different aspects mentioned below provided a generic best implementation concept.

In addition a maintenance plan will be stipulated and implemented across the ASTRO-WISE sites.

5.3 Investigations

To define proper environments for data storage facilities for the future the several hardware aspects needed to be tackled in close correspondence with the implementation choices made for the data processing hardware.

5.3.1 Hardware

Questions as to whether a centralized or distributed data storage implementation are preferred involved on the one hand the I/O requirements for the data processing pipeline and on the other hand the I/O requirements for data mining purposes. Because these two components are very different in nature, and because the concept of a final dataset does not exist in the ASTRO-WISE context, they are the major concern of this Work Package. From existing pipeline implementations

and known boundary conditions the I/O requirements of the first component can be well defined. This is not the case for the second component because, first, our knowledge of true data mining is still sparse and, second, by the very fact that data mining is such a dynamic and versatile process.

The investigations with respect to the implementation detail to such things as the type of the storage media. For hard drives this meant investigating the size (largest types) and their access speeds (read/write). For tape media this would be the permanency (duration, life time) and possible hierarchical storage facilities and total storage capacity. Most of the storage hardware needs to be intelligent and contains or must be connected to a CPU. Either one automatically builds a Beowulf type of system while evolving to larger disk farms or one piggy-backs on the Beowulf system to provide these CPU's.

It turns out that data storage units are becoming specialized systems on their own and are not well equipped for the parallel processing task as viewed from a pipleline perspective. Given local high speed networks a data storage facility can reside next to a processing facility in a LAN environment. Data that is to be processed in an automated pipeline fashion can then be ingested on the local data store while data processing necessary for data mining purposes may reside on non-local data stores.

5.3.2 Software

A data storage unit has no value without a means of access and content description. These two aspects are usually provided by software and databases. Therefore, it is vital for an operational data storage facility to be described by software and data to be retrievable through software. Important aspects concerning the choices of software involved such things as global and local administration where distributed data (among the different nodes) is uniformly accessible. For example a data retrieval query at one node should result in a number of distributed queries across the ASTRO-WISE nodes.

This administrative software in its simplest form hides the UNIX files structure behind the pipeline file administration. It needs decision mechanisms to distribute data on intake, redistribute data for processing efficiency, and redistribute data for data mining efficiency. This is very similar to a DBMS were the files are DBobjects. The actual important concept is the decision mechanism that has knowledge of the underlying hardware for distribution data across local environments, such as the Beowulf clusters, but also among the ASTRO-WISE nodes, involving internet connections. As a first approach data to be processed as part of the ASTRO-WISE node allocated observing time will be ingested at that ASTRO-WISE node. It can then, by default, be the master data store for that particular data.

The above concepts worked out in more detail result in the following description.

Data identification

As the telescope acquired data is ingested into the processing system, it is given a unique identification within the federated database environment because the associated DBObject is stored there and the unique identification from the DBObjects is used. The place where the actual data will reside is the place of ingest. This is the simplest means of defining the master of this data item. Upon ingest the data is stored in a permanent storage environment from which it can not be erased. The federated database has provided a global identification (GID) for the ingested data item so that it can be uniquely referenced using this GID.

Data management

The manager of a particular data store is the dataserver. It is both a server of the data responding to *get* requests and *put* requests as well as a database manager that translates the GID to a local

UNIX filename (or any other storage configuration). The caretaker role information is stored in the federated database where DBObjects are stored. For human readability the GID translates to a UNIX filename which is constructed using the data description information as given by the ASTRO-WISE data model.

Federated data store

Each data-store manager cooperates with the other data-store managers in such a way that they communicate data and information among each other so that the ensemble of data-store managers in itself is a federated data store. To illustrate this fact, when a client data processing unit requests a data item by means of a GID, it will do this request to the local dataserver. In principle a local dataserver can be running on one node of a parallel computing cluster, a single data storage computer environment near the parallel computing cluster or even at some more remote site. The data item retrieve request will be answered by the local dataserver either directly as the requested data is part of the dataserver master data, or indirectly. In the latter case the local dataserver, which does not have the data item stored locally, will peer-to-peer communicate with the other dataservers in the federated data store to download the data item from the remote data server that is the master of the requested data item. The downloaded data item will reside in a caching area on the local data server, which then serves the data item to the processing client. This way often requested data items will always be stored locally. A scratching period for the local cache area will have to be established.

In fact it is even possible to build a hierarchical dataserver structure with for instance dataservers running on each node of a parallel processing cluster, one central dataserver serving all these node based dataservers. This central dataserver could then be part of the federated data store. There is no obligation for a dataserver to be the master of any data so that it only has data in its cache area while its permanent area is empty.

As for the storage of derived data the local dataserver will be the master of this newly created data. This functionality operates in the same way as the ingest procedure for the raw observed data. One might, in a hierarchical data store environment, setup a structure defining master data stores for particular data items. For instance, processed images created at the Leiden parallel processing cluster will have as their master data store the Groningen dataserver. This uploading procedure will also be a multi step mechanism with data uploaded from the parallel cluster going to the cache of the local data server and then at some time disjunct from the data processing be uploaded to the actual designated master data store.

The federated nature of the dataservers has the advantage that the data store can dynamically grow. A dataserver is the master of a particular set of data which might be limited as the data storage hardware has capacity limitations. When a data storage hardware environment (e.g. a Linux based computer with a few TB of disk space) is build and filled, one can just add another such environment with its own dataserver running, making it the master of some new data items, thus dynamically enlarging the federated data store.

Unit of data item

For the purpose of data storage and retrieval the unit of a data item is the FITS file. This means that one can only retrieve and store complete FITS files, not separated FITS file extensions. This has the consequence that multi-extension FITS files such as those coming from the telescope will have to be split before they are ingested into the data store. The data processing pipeline operates on single chip data. The data items are accessed through their GIDs, but the associated DBObject in the federated database is the place where to access the information items (such as FITS header information).

5.3.3 Prototype

Given above considerations a dataserver prototype has been build. This dataserver program is written in Python and uses the peer-to-peer paradigme for its communication mechanism. A dataserver can become part of an ensemble of dataservers by giving the program a reference to one of the other dataservers in the ensemble. At startup the new dataserver will announce its presence as part of the ensemble. The dataserver is a multithreaded daemon process that listens for requests.

Communication is based on http post and get authenticated requests for files. The communication routines are implemented as a client - server library.

A request for data is placed by the client side (processing program) to a dataserver, usually one that is nearby in terms of network performance. If the dataserver is not the master of the requested data and if the requested data is not in the cache, the dataserver queries all other dataservers in the ensemble for the data. When another dataserver has the requested data in its master or cache area, it serves this data to the requesting dataserver, which stores it in it cache area and forwards it to the requesting client program. The next time the same data is requested a cache copy can quickly be delivered.

The prototype lacks a GID to filename conversion and only has a rudimentary cache cleanup facility.

5.4 Hardware at Groningen / Leiden

5.4.1 - dataserver for CVS, dbI, observational data

The huge amount of data obtained with OmegaCAM requires a powerful computer platform, able to process and maintain huge data volumes and the derived products on-line. This is achieved with a Beowulf system (discussed in WP4), with a host-based RAID disk network providing the multi-TByte storage. A typical 'storage unit' consists of a cpu and a number of inexpensive disks (1 TByte/host in 2002, 2 TByte/host in 2003). This is currently the most cost-effective solution, with the important advantage that it is scalable: the storage capacity can be upgraded over time by simply adding more units to the network. Thus we can expand the system as the data volume grows, allowing us to always buy the most effective components. The Beowulf cluster plus storage units will be located at the Kapteyn Institute. The scalability of the system allows the expansion and purchasing of components on 'the last minute' when data accumulation forces us to up-scale.

The acquiring of storage units has already started: Early 2002 a 0.5 TByte fileserver came into operation. This machine is used primarily as CVS data server and for data storage of the informational data (meta-data) in Oracle. In january 2003 a 2 TBytes RAID was acquired to store the observational (test) data. Next we will procure another 2 TBytes in the summer and 4 TBytes in the fall of 2003, 20 TBytes in the fall of 2004 and 30 TBytes at the end of 2005.

Currently each storage unit of 2TB consists of 12 200GB (7200rpm, 8MB cache) disks which are connected to a 3ware parallel ata controller running a RAID5 system with one spare disk. The net result is a disk of in total 2TB which is quite well manageable via the 3ware web-tools. Experience has shown us that the 3ware solution is faster, more reliable and better maintainable than many other RAID solutions like for instance the Promise SX6000 controller. Each storage unit is connected to the net via a 1 Gbit/s NIC.

5.5 Hardware at Paris

A distributed data storage capability is installed at TERAPIX, with currently 10+1 nodes. Depending on when they where bought (2001, 2002 or 2003), the client nodes are equipped with 700GB, 2.5TB and 3.5TB of RAID5 securized disk space, for a total of 20TB online storage. The server

node has 2 2.5TB RAID5 systems for storing temporarily both incoming and outcoming (reduced) data. These RAIDs are used as a buffer for exchanging data with the archiving data centers, via internet. The amount of available online disk space is expected to grow as more observations become available, in order to have everything online and be able to quickly reprocess any data at anytime. One year of observing data with MEGACAM requires about 10TB of storage space.

5.6 Hardware at USM

A distributed data storage capability has been incorporated at the USM. The server node of the Beowulf cluster has a 1 TB RAID5 disk array that will be devoted to the direct pipeline reduction of data. We foresee a total expansion of this storage to 10 TB, and have ensured that the current rack can accommodate this volume of disks. Furthermore, four workstation PC's will be connected to the Beowulf cluster to give the user a local storage and work area. Each PC has a RAID3/5 array with 815.5 GB of storage.

5.7 Hardware at Capodimonte

For what concern the data storage, no decision has been taken yet. Our feeling at the moment is that in order to have a data storage system with enough reliability we need to separate it from the cluster. Several disk storage commercial solution are available on the market also if not cheap compared with an "home made" disk farm. Presently we are investigating both the solutions. The final choice will depends also on the data distribution strategy that will be adopted within ASTRO-WISE .

Chapter 6

WP6: Coordination

6.1 Project organization

WP6 is coordinating this project. The view is that keys to the success of the system involve:

- extensive design and setting standards.
- setting up federations of source code, documentation and observational data
- frequent communications of partners sites by
  - 2-weekly telekons
  - mail boxes/newsletters, websites etc
  - quarterly meetings
  - regular work visits

WP3, providing federated databases such as CVS, Oracle and web-pages, plays a double role in the project: it supports the administration of scientific processing, but it also provides the workhorse for the various partners to collaborate on development and exploitation of the system, for running scientific projects and surveys.

During the development work each partner site will provide at least a local contact person and a local data base administrator (fraction of time), next to the various software developers.

A deliverable of the project is the emanation of the system to satellite sites. During the development of the system it is expected that the maintenance of such satellite sites requires about 0.2 fte.

ASTRO-WISE can deliver data products to the Astrophysical observatory (AVO) and close contact with the VO's and the Grid projects, like AstroGRID and DataGRID, is considered very desirable, particularly for the inter-operability issues.

A liaison to VISTA is organized as part of the project.

Other ASTRO-WISE associates are Sterrewacht Leiden, RuG Rekencentrum Groningen, Oracle NL, Université de Liège, Sternwarte Bonn, ESA- GAIA.

Appendix A

Persistency Interfaces

A.1 Introduction

This document describes the specification and Python implementation of persistent objects on top of a relational database back end. The aim of this implementation is twofold:

1. Provide a transparent mapping from a definition of a persistent class to a table in a relational database, preserving inheritance relationships, and allowing attributes to refer to other persistent objects.

2. Provide a native Python syntax to express queries, and leverage the advantages of the relational model (SQL) when using persistent objects.

In this paper we will first introduce a number of concepts from Object Oriented Programming (OOP) and Relational Database Management Systems (RDBMS), in order to clarify the problem we wish to solve. We will then provide the specification of the database interface provide by the ASTRO-WISE prototype. Finally, we will clarify some of the implementation issues addressed by the current prototype.

A.2 Background

A.2.1 Object Oriented Programming

It is difficult to give a meaningful definition of "object". However, the following "definition" introduces some intimately related terms that will be used throughout this document:

object An object is something that comprises **type**, **identity** and **state**. The type of an object, specifies what kind of object it is, specifically what kind of behavior the object is capable of. The identity is what distinguishes one object from another. The state of an object specifies the values of the properties of the object.

In Object Oriented Programming (OOP) we have an operational definition of objects:

object An object is an **instance** of a **class**, and **encapsulates** both data and behavior

The class defines what operations (**methods**) can be performed on its instances, and what **attributes** those instances will have. In general 'class' and 'type' are synonymous, as are 'instance'

and ‘object’. That is, when we talk about the type of an object we mean the class of which it is an instance.

It is important to note that the values of the attributes of an object will themselves be objects, although most programming languages distinguish between (instances of) primitive data types (integers, strings, etc) and instances of classes.

Inheritance is the mechanism by which one can use the definition of existing classes to build new classes. A child (derived) class will inherit behavior from its parent (base) class. In defining the child class the programmer has the opportunity to extend the child class with new methods and attributes, and/or modify the implementation of methods defined in the parent class. However, the child class is expected to conform to the interface (specification) of the parent class, to the extent that instances of the child class can behave as if they are instances of the parent class. In particular it is expected that procedures taking an object of a base type as argument, should also work when given a derived type as argument. This key property of objects is called **polymorphism**

A.2.2 Persistency

An object is said to be **persistent** if it is able to ‘remember’ its state across program boundaries. This concept should not be confused with the concept of a program saving and restoring its data (or state). Rather, persistency, implies that object identity is meaningful across program boundaries, and can be used to recover object state.

Persistency is usually implemented by an explicit mapping from (user-defined) object identities to object states and by then saving and restoring this mapping. However, this implementation assume that the object identity of the object one is interested in can be independently and easily obtained. For many applications this is not the case. On the contrary, one usually has a (partial) specification of the state, and are interested in the corresponding objects that satisfy this specification. That is, many interesting applications depend on a mapping of a partially specified object state to object identity (and then to object). This is the domain of the relational database.

A.2.3 Relational Databases

A relational database management system (RDBMS) stores, updates and retrieves data, and manages the relation between different data. A RDBMS has no concept of objects, inheritance and polymorphism, and it is therefore not a-priory obvious that one would like to use such a database to implement object persistence. However, using the following mapping

$$\begin{array}{rcl} \textbf{type} & \longleftrightarrow & \textbf{table} \\ \textbf{identity} & \longleftrightarrow & \textbf{row index} \\ \textbf{state} & \longleftrightarrow & \textbf{row value} \end{array}$$

it is (hopefully) obvious that one might, at least in principle, implement object persistency using a relational database. That is, given a type and object identity, one can store and retrieve state from the specified row in the corresponding table.

Relational databases provide a powerful tool to view and represent their content using structured queries. It would be extremely useful if we were able to leverage this power to efficiently search for object whose state matches certain criteria. Special consideration has to be given to inheritance in this case.

Assume, for example, that we define a persistent type `DomeFlatImage`, derived from a more general type `FlatfieldImage`. A query for all R-band flatfield images, should result in a set including all R-band domeflat images. This behavior of queries is what inheritance means in a relational database context. Hence, a query for objects of a certain type maps to queries (returning row indices/object identities) on the tables corresponding to that type, and all of its subtypes. The

results of these queries are then combined in to a single set of all objects, of that type or one of its sub types, that satisfy the selection.

A.3 Problem specification

The implementation of the interface (should) address(es) the following issues:

defining a persistent class Defining a persistent class (type), will give its instances the property of being persistent. The class definition should provide sufficient information about the attributes (possible state) of the objects to build the corresponding database table. This table should be present in the database when the first object of the class is instantiated. Presently, this is achieved by dynamically creating the table (if it doesn't yet exist), when processing the class definition[1]

retrieving state of persistent object Instantiating a persistent object with an existing object identity should result in retrieval of state from the database.

saving state of persistent objects Persistent objects, whose state has been modified, should save their state to the database before they cease to exist.

references persistent objects will contain references to (read: instances of) other persistent objects. Care has to be taken that instantiation of a persistent object does not recursively instantiates all objects it refers to. Only when the attribute corresponding to the reference are accessed should the corresponding object be instantiated.

expressing selections It should be possible to express selections of the form

$$\{x|x \in X \wedge (x.attr1 \in A \wedge x.attr2 \in B \vee x.attr3 \in C...)\} \tag{A.1}$$

i.e.: the set of all objects of type X whose attributes have certain properties. This set should be translated in to an SQL query to the database, and result in an iterable sequence of objects satisfying the selection.

In addition, the following issues need to be addressed, though not necessarily by the interface to persistent objects.

managing database connections The interface does not specify how or when the database connection is established.

transactions The interface doesn't specify if and how transactions are implemented

efficiency No effort has yet been made to maximize performance and/or scalability. Initial efforts has focussed on a demonstration of technology and simplicity of implementation.

A.4 Interface Specification

In this section we describe how to implement and use persistent objects, using the interface defined in the astro-wise prototype. This section includes Python source code fragments. For those not familiar with Python we advise that they have a look at the main web site at `http://www.python.org/` and at the Python tutorial at `http://www.python.org/doc/current/tut/tut.html`

[1] This implementation neatly avoids the problem of having to maintain both the class hierarchy and the corresponding database schema

A.4.1 Persistent classes

Persistent objects are instances of persistent classes, which specify explicitly which attributes (properties) are saved in the database. We call these attributes persistent properties. Executing a program defining

Defining persistent classes

A new persistent class is defined by deriving from an existing persistent class, or by deriving from the root persistent class DBObject. E.g.:

```
#example1.py
from astro.database.DBMain import DBObject
class A(DBObject):
    pass
class B(A):
    pass
```

specifies two persistent classes (A and B). Neither of them extends their parent classes, so instances of A and B will behave exactly like instances of DBObject.

Defining persistent properties

A persistent property is defined by using the following expression in the class definition:

```
prop_name = persistent(prop_docs, prop_type, prop_default),
```

where, `prop_name` is the name of the persistent property, and `persistent` is constructed using three arguments: the property documentation, the type of the property, and the default value for the property respectively. For example:

```
#example2.py
from astro.database.DBMain import DBObject, persistent
class Address(DBObject):
      street = persistent('The street', str, '')
      number = persistent('The house number', int, 0)
```

This program defines a persistent class 'Address', with two persistent properties, 'street' and 'number', of type str(ing) and int(eger) respectively.

We distinguish between 5 different types of persistent properties, based on the signature of the arguments to `persistent()`

descriptors If the type of the persistent property is a basic (built-in) type, then we call the persistent property a descriptor. Valid types are: integers (int), floating point numbers (float), date-time objects (DateTime), and strings (str).

descriptor lists Persistent properties can also be homogeneous variable length arrays of basic built in types, called descriptor lists. Valid types are the same as those for descriptors. descriptor lists are distinguished from descriptors by the property default. If the default is a python list, the the property is descriptor list, else it is a simple descriptor.

links Persistent objects can refer to other persistent objects. The corresponding properties are called links. If the type of the persistent property is a subclass of DBObject, then the property is a link.

link lists Persistent properties can also refer to arrays of persistent objects, in which case they are called link lists. Link lists are distinguished from links by the property default. If the default is a python list, the the property is link list.

self-links A special case of links are links to other objects of the same type. These are called self-links. if no type and default are specified for the call to `persistent`, then the property is a self-link.

Keys

It is possible to use persistent properties as alternative object identifiers for the default object identifier (`object_id`). Only descriptors can be used as keys. Keys are alway unique and indexed.

The special attribute `keys` contains a list of attributes and tuples of attributes tuples, each specifying one key. For example:

```
#example3.py
class Employee(DBObject):
    ssi = pesistent('Social Security Number', str, '')
    name = persistent('Name', str, '')
    birth = persistent('Birth data', DateTime, None)
    keys = [('ssi',), ('name', 'birth')]
```

In this example `ssi` is a key. The pair of attributes `('name', 'birth')` is also a key.

Indices

Databases use indices to optimize queries. It is possible to specify which persistent properties should be used as indices.Only descriptors can be used as indices.

The special attribute `indices` contains a list of attributes which should be indexed. E.g.:

```
# example4.py
class Example(DBObject):
    attr = persistent('A measurement', float, 0.0)
    indices = ['attr']
```

A.4.2 Persistent Objects

Having specified persistent classes, we can now use these classes to instantiate and manipulate persistent objects. In most respects these objects behave just like instances of ordinary classes. There are two exceptions: special rules for instantiation, and special rules for assigning values to persistent properties.

Object instantiation

We can distinguish between three different modes of instantiating a persistent object.

New We are creating a new persistent object, for which the `object_id` needs to be generated. This can be accomplished by instantiating an object without specifying `object_id`.

Existing We are using an existing object. If the object has already been instantiated in this application we want a copy to its reference, otherwise we want an instance, whose state has been retrieved from the database. This can be accomplished by instantiating the object with an existing `object_id`.

Transient it may be useful to build an object of a persistent type that is not itself persistent (whose state, will not be save to the database). This can be accomplished by instantiating the object with an `object_id` equal to 0 (zero)

or, in code:

```
a = MyObject()               # A new instance of MyObject
b = MyObject(object_id=1000) # An existing instance of MyObject
c = MyObject(object_id=0)    # A transient instance of MyObject
```

In practice, objects are rarely instantiated with an explicit `object_id`, because, we will generally not know the `object_id` of the objects we are interested in. Rather, objects are instantiated using keys or as the result of a query (see below)

Instantiating an object using a key, will result a restored object (if an object of that key did exist before) or a new object. In code:

```
class Filter(DBObject):
    band = persistent('the band name', str, '')
    keys = ['band']

f = Filter(band='V')       # The V-band filter
```

Assigning values to properties

Python is a dynamically typed language. This means that there is no such thing as the type of a variable. However, since database values (e.g. columns) are statically typed, the interface performs type checks when binding values to object attributes. The type is specified in the property definition, as outlined earlier.

A.4.3 Queries

In order to represent selections in native Python code, we have defined a notation that is based on the idea that a class is in some sense equivalent to the set of all its instances. To illustrate the concept, let us give a few examples.

Given a persistent class `X` with persistent property `y`, then the expression

```
X.y == 5
```

represents the set of all instances `x` of `X`, or subclasses of `X`, for which `x.y==5` is true. To obtain these objects the expression needs to be evaluated, which can be done by passing it to the `select` function, which returns a list of objects satisfying the selection.

Given a class `X` with a descriptor `desc`, a descriptor list `dsc_lst`, and a link `lnk`, then

```
select(X.desc > 2.0 && X.dsc_lst[2]=='abc' and X.lnk.attr == 5)
```

will return a list of instances `x` of `X`, or subclasses of `X`, for which

```
x.desc > 2.0 and x.dsc_lst[2]=='abc' and x.lnk.attr == 5
```

is true.

A.4.4 Functionality not addressed by the interface

New persistent objects may have an owner. The owner can defined as the user running the process in which the persistent object is created or it can be defined as an attribute of the persistent object. In either case, it is the responsibility of the implementation of the interface for a certain database to handle ownership of persistent objects.

Appendix B

Quality Control: possible problems and solutions

For what concerns the Quality Control (QC) issues, the first step is to identify a list of possible problems that may affect the OmegaCAM data. The table below contains a first version of this list. It is clear however that not all these problems can be detected in a simple way. Therefore the second step is to identify, among this list, those problems that can be detected through simple, specific and robust tools which run in an automatic way when the pipeline is running. On the other hand, the remaining problems, considered as very rare and/or too complex, will not be detected in an automatic way by the pipeline.

For what concerns in particular the calibration pipeline, the QC tools produce, as output, a numerical quality index (QI). When the QC is applied to raw frames, the frames are accepted (or discarded) depending whether the QIs are smaller (or greater) than a tunable threshold. Although the QC tools might not be visible directly by the user when the pipeline is running, all the QC outputs (not only QIs but also other numbers, tables and figures that can be produced) will be saved in the DB, in order to keep track of the whole process.

Finally the list below does NOT contain the problems related to:

– Instrumental problems (loss of focus, pointing-tracking problems, CCD sensitivity degrade over time, optical/mechanical/electronic problems)

– Data description problems, bad header information (lacking headers, wrong headers, data different from what they were planned to be);

– Observational constraints (e.g. Service Mode observations can have seeing constraints that are not satisfied by the data);

– Empty/Crowded field observations (possible failure of Sextractor background subtraction and de-blending, possible failure of astrometric calibration);

– Possibility to use only a fraction of the mosaic (e.g. if one chip contains a really bright (mag<8) object, we might want to be able to use the other 31);

– PSF homogenization in different bands (needed to obtain accurate colors)

Table B.1: Possible problems and solutions

Description	Problem	Solution	Tool	Comments
Calibration frames (general)	strange statist. distribution	check statistical distribution (create histogram of pixel values and compare with Gaussian)	histo.py[1]	under testing
BIAS (raw)	uncorrelated vs. correlated noise	compare with Gaussian stat.	Signtest.py[2] (applied to the the difference between two BIAS frames)	working
	short-term time variability (t ∼30 min)	compare with previous	Signtest (see previous point)	⇒ Overscan correction with y dependence desirable
BIAS (master)	long-term time stability (from one night to the next)	compare with previous	Signtest	DB Trend Analysis
Dome FF (raw)	under/over exposures	measure median and reject underexposed/ saturated frames		already implemented in the pipeline
	strange pixels (outliers)	check number of outliers and reject bad frames	count_outliers[3]	under testing
	non-flatness	measure RMS of 2d cubic spline fit (through windowing) and reject bad frames	imsurfit_stat	under testing
	short-term time stability (from one frame to the next)	compare with previous		low effect expected; probably not a pipeline issue
Dome FF (master)	long-term time stability (from one night to the next)	compare with previous		DB Trend Analysis

[1] *This tool can be used for any calibration frame and will be particularly useful during the commissioning phase. It could also be linked to the graphical user interface in order to show the statistical distribution of the pixels in each raw frame, when the pipeline is running*
[2] *The sign test tool can be used in principle for any calibration frame provided that the noise is mainly uncorrelated. It might be used also for DB trend analysis*
[3] *Also this tool can be used in principle for any calibration frame; in particular it can be used to produce bad pixel masks. It might be used also for DB trend analysis*

Table B.2: Possible problems and solutions (continue)

Description	Problem	Solution	Tool	Comments
Twilight FF (raw)	under/over exposures	measure median and reject underexposed/ saturated frames		already implemented in the pipeline
	strange pixels (outliers)	check number of outliers and reject bad frames	count_outliers	under testing
	non-flatness	measure RMS of 2d cubic spline fit (through windowing) and reject bad frames	imsurfit_stat	under testing
	bright objects	SExtractor detection + removal/masking	Derfotron	with 10 twil. FFs and large telescope displacements
		Or use comparison with previous frame	count_outliers	bright objects should be removed in any case
Twilight FF (master)	long-term time stability (from one night to the next)	compare with previous		DB Trend Analysis
Super-flat	bright/large objects in raw scientific frames	SExtractor detection and automatic masking	Derfotron	
Fringing	check quality of fringing removal	measure residuals	Derfotron	might be not necessary if the fringing correction works fine in most cases
	check stability of fringing pattern in time	compare with previous		DB Trend Analysis (check z-band, lunar phase)
Science frames (raw)	saturated stars	SExtractor detection + automatic masking Manual masking	Derfotron	
	stellar diffraction spikes	SExtractor detection + automatic masking Manual masking	Derfotron	
	satellite trails	SExtractor detection + automatic masking Manual masking	Derfotron	
	reflections/ ghosts/halos (oblique scattering)	SExtractor detection + automatic masking Manual masking	Derfotron	
	bad cosmic rays removal	check number of detections vs expected	TBD	the USM tool performs also some verification based on the number of detections

Table B.3: Possible problems and solutions (continue)

Description	Problem	Solution	Tool	Comments
"Sky concentration" (Illumination correction)	zero-point depends on x-y position due to additional defocussed light on the focal plane	independent zero-points in each CCD		x-y spatial mapping of the zero point through standard fields during commissioning
Photometric calibration	standard fields must be reduced exactly in the same way as the science frames	impose that same calibration master frames are used: compare descriptors		If requirement is relaxed, specify and check acceptable differences
	transparency variations between standard and science fields	compare statistics		It would be extremely useful to save the counts of the guide stars as it was proposed some time ago
	insufficient number of stars in each CCD	check number of stars; compare zero points with previous ones		
	saturated stars in standard fields	exclude saturated stars from the fit through sigma clip algorithm		
Science frames (reduced)	check final background	track background level and backgr. noise	Derfotron (Terapix)	
	check final astrometry	check residuals		
	check final photometry	check star and galaxy counts	Derfotron (Terapix)	
	PSF homogeneity	measure PSF across the field	Derfotron (Terapix)	
	PSF shape	ellipticity, skewness and kurtosis stat.	Derfotron	
Radial wavelength dependence	λ_0 changes from center to edges (for interference filters only)			commissioning