Insert here your thesis' task.

Czech Technical University in Prague

Faculty of Information Technology

Department of Software Engineering

Master's thesis

# Cross-matching Engine for Incremental Photometric Sky Survey

***Bc. Jiří Nádvorník***

Supervisor: RNDr. Petr Škoda CSc.

4th May 2015

# Acknowledgements

Most of all I would like to thank my supervisor Petr Škoda for patience and willingness to help me with astronomical aspects of this thesis. I would also like to thank Filip Hroch for his support in Munipack usage as well as in theoretical aspects of my thesis. The next person that supported it very helpfully is Martin Reinecke from the HEALPix support and last but not least I would like to thank Markus Demleitner from GAVO for his support far exceeding the scope of GAVO DaCHS package we are using. We also acknowledge grant GACR 13-08195S.

# Declaration

In Prague on 4th May 2015 .....................

## Citation of this thesis

Nádvorník, Jiří. *Cross-matching Engine for Incremental Photometric Sky Survey.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

# Abstrakt

Pro získávání světelných křivek je dnes potřeba předem naplánovat přehlídku oblohy, kde jsou pevně dané souřadnice expozic, které se v průběhu přehlídky nemění. Tato práce ukazuje, že to není nutné a že lze vytěžit světelné křivky z astronomických dat, která k tomu vůbec nebyla původně určena. Tímto způsobem můžeme zrecyklovat všechny fotometrické přehlídky na světě a vytvořit k nim světelné křivky pozorovaných objektů.

Tato práce se zabývá zejména způsobem generování katalogu objektů, který je nutný pro výše zmíněné světelné křivky. V praxi se soustředí na jeden z největších problémů v astroinformatice. Jedná o klastrování datových objemů na úrovni Big Data, kde většina tradičních technik selhává a my musíme hledat nové cesty k dosažení cíle. Zabýváme se širokou škálou možných řešení z pohledu výkonu, škálovatelnosti, distribuovatelnosti, atd. Definujeme kritéria pro časovou a paměťovou složitost, která jsme vyhodnotili u všech testovaných řešení. Dále si vytvoříme kvalitativní nároky, které také zohledňujeme v hodnocení výsledků.

Používáme relační databáze jako výchozí bod implementace a srovnáváme je s nejnovějšími technologiemi potenciálně použitelnými pro řešení problému. To mohou být noSQL Array databáze nebo přesunutí výpočetně náročných fází na superpočítače s použitím paralelismu.

**Klíčová slova** astronomie, paralelismus, klastrování, big data, databáze

# Abstract

For light curve generation, a preplanned photometry survey is needed nowadays, where all of the exposure coordinates have to be given and don't change during the survey. This thesis shows it is not required and we can data-mine these light curves from astronomical data that was never meant for this purpose. With this approach, we can recycle all of the photometric surveys in the world and generate light curves of observed objects for them.

This thesis is addressing mostly the catalog generation process, which is needed for creating the light curves. In practice, it focuses on one of the most important problems in astroinformatics. This is clustering data volumes on Big Data scale where most of the traditional techniques stagger and we have to search for new paths to achieve our goal. We consider a wide variety of possible solutions from the view of performance, scalability, distributability, etc. We define criteria for time and memory complexity which we evaluated for all of the tested solutions. Furthermore, we create quality standards which we also take into account when evaluating the results.

We are using relational databases as a starting point of our implementation and compare them with the newest technologies potentially usable for solving our problem. These can be noSQL Array databases or transferring the heavy computations of clustering towers supercomputers by using parallelism.

**Keywords** astronomy, parallelism, clustering, big data, database

# Contents

# List of Figures

Chapter 1

# Introduction

In this chapter we will define the topic of this thesis, it's contents, justification and purpose. As already said in the abstract, the main part of this thesis is to take a set of observations[1] and cluster them based on Euclidean and other metrics.

The ultimate goal is to provide light curves[2] of astronomical objects to the end user, with as much quality as possible. For that, we need to generate our own catalog[3] of objects. The main problem with cross-matching[4] matching only an existing catalog is that we won't be able to create light curves for all of our observations. If we create our own catalog, however, we can guarantee, that all of our observations will be part of a light curve. The As-Is state can be seen on Fig. 1.1 and To-Be on Fig. 1.2.

## 1.1 Motivation

In this chapter we will justify the motivation for our thesis. We will explain the usefulness of the work and the benefit it will bring to the end users. The end users for our system are astronomers, but the nature of our data makes it very interesting for geospatial science too.

The high level motivation is to create a light curve catalog for OSPS (Ondřejov Southern Sky Photometry Survey) project [1], with the data originating form the Danish 1.54-m Telescope [2]. This photometry survey contains hundreds of thousands of images, but almost all of them are observing the

[1]By the term observation, we mean a light dot identified on an image of the sky

[2]Light curve is a graph of an observation's brightness based on time when the observations where taken.

[3]Catalog of astronomical objects is list of celestial objects identified by their coordinates. These can be planets, stars, galaxies, quasars, etc.

[4]Cross-matching in astronomy is understood as a process of matching one data set with another. The criterium is mostly distance between the data points, which means a point from set A will only be matched to a point from set B if their distance is smaller than the criterium.

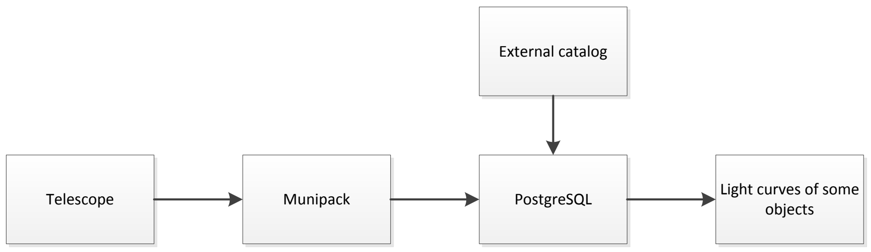


Figure 1.1: As-Is state

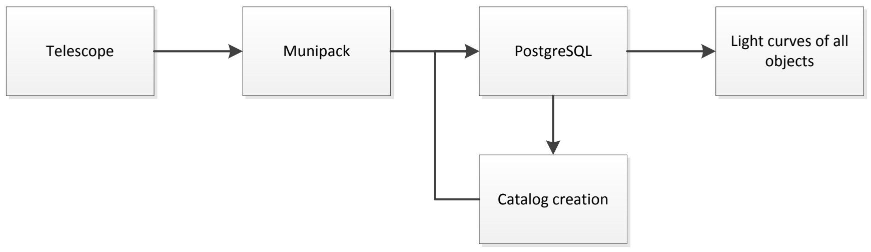


Figure 1.2: To-Be state

same region on the sky. Identifying celestial objects on these images is a complicated process of astrometry[5] and photometry[6]. The output of this process is a set of observations of all identified objects[7] for each image.

Our data used for producing the light curves is not actually meant for that originally and that's why we differ from the standard solutions nowadays. This aspect will be brought in detail in 1.2.

## 1.2 Data structure

There is a lot of sky surveys which clearly had to solve this problem already. To answer this question decisively, we need to introduce the structure of our data.

[5]Astrometry is a process of measuring exact positions and movements of celestial objects

[6]Photometry is a process of measuring the flux, or intensity of an astronomical objects electromagnetic radiation

[7]An observation of individual object is defined by their sky coordinates, photon flux, and lots of other parameters

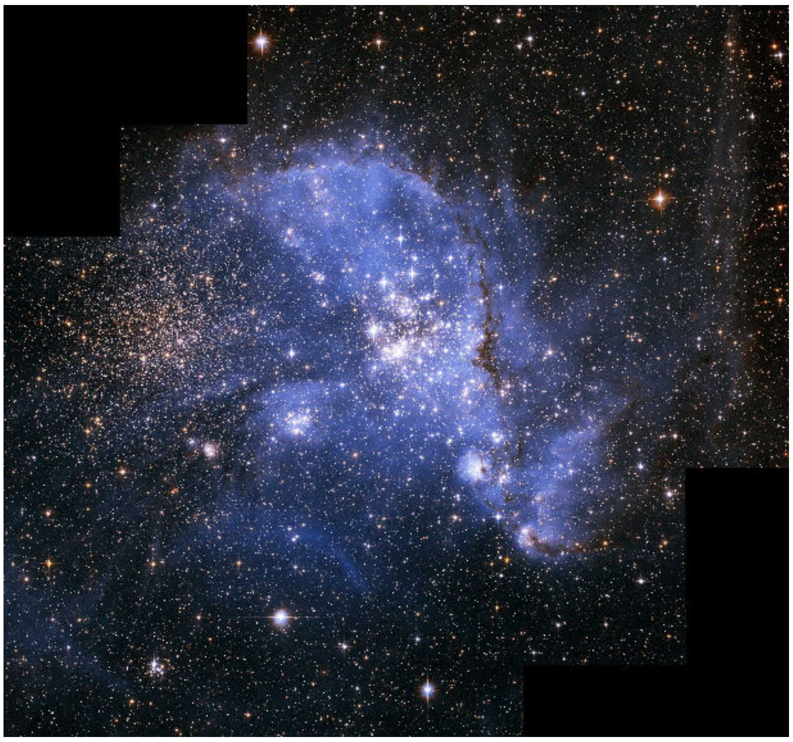


Figure 1.3: Small Magellanic Cloud

As most of the fields are located in the area of Small Magellanic cloud[8] seen on Fig. 1.3, they are very dense. On one image we have cca 5000 - 25000 identified observations. For comparison, on an average region on the sky with the same coverage and deepness, there will be between 2000 and 5000 observations. We have cca one hundred thousand images in our dataset, with a total of four hundred million identified observations. The dataset will still grow until the end of the project when we expect to have six hundred million identified observations which we need to assign to celestial objects. A typical image looks like NGC330[9] on Fig. 1.4.

### 1.2.1 Typical versus our data

In this section we will compare the typical approach to creating a light curve catalog with the one we had to choose for ourselves. The reason for it is the different structure of typical and our data.

[8]Small Magellanic Cloud is a dwarf iregular galaxy, one of the closest neighbors to our Milky Way

[9]NGC330 is a open star cluster in Small Magellanic Cloud

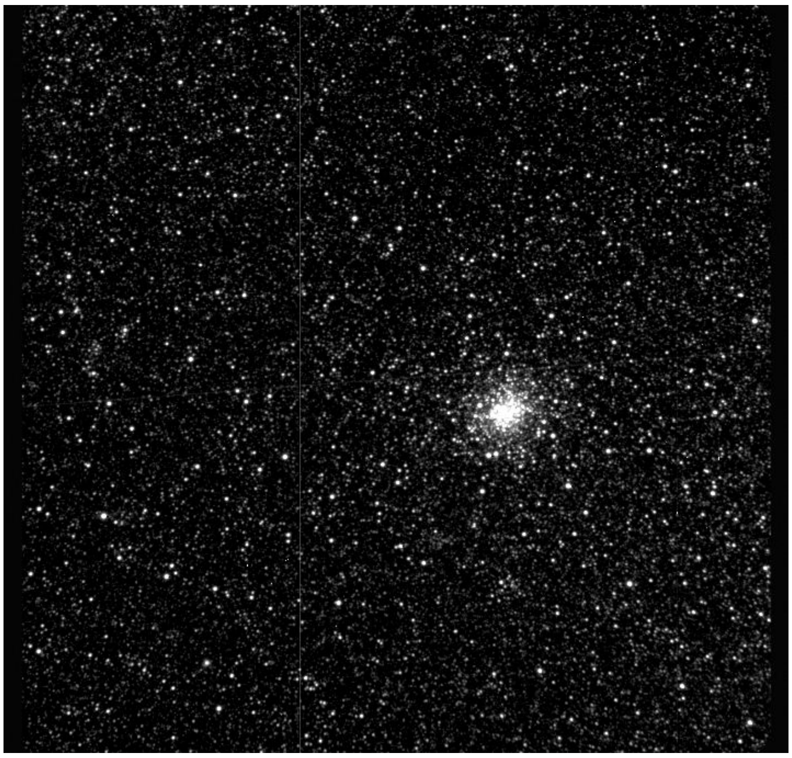


Figure 1.4: NGC330 image

#### 1.2.1.1 Typical data

A typical light curve survey will have a before-hand defined grid of image areas which will be observed repeatedly in regular time intervals. This grid will not change during the project. This means that two images taken at different times which cover the same region will overlap almost entirely (the intersection of their coverages will be almost as big as the whole image). This means that a differential photometry and astrometry can be used. The most crucial condition for a differential astrometry to to be successful when comparing two images is to have enough common coverage, so we can match their positions. The difference between absolute and differential astrometry is shown at the beginning of this presentation [3].

#### 1.2.1.2 Our data

Unfortunately, we cannot use differential astrometry for our survey. The images taken in OSPS have mostly only few objects of interest (a planet, asteroid, or a Be star, ...), which means two things. First, 99 % of the data is not originally meant to be used and second, there is no grid for restricting the image positions. This is best seen on Fig. 1.2.1.2 - here we can see the chaotic spread of our images which makes differential astrometry and photometry stagger, because the images won't have enough referential stars in common.

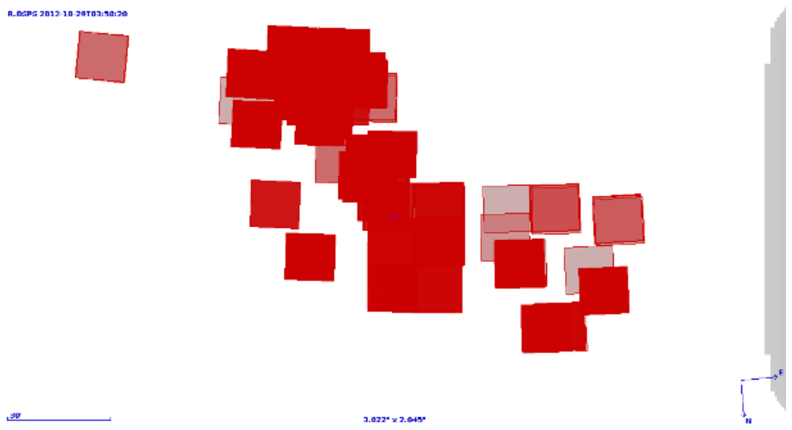


Figure 1.5: OSPS coverage

## 1.3 Our solution

We would like to use this 99% of our data which is just thrown out, but has no less quality and can easily lead to new discoveries. We believe that our survey is not a rare case when most of the data is unused. With our approach, we can actually recycle all of the images in the world even if they were not originally meant for producing light curves and data-mine much information from them.

Another reason why we'd like another approach is that we want our survey to be incremental. That is not always the case for standard surveys, as when they are closed, it would be very complicated to add later (or sooner) taken images to the survey. When new images are taken, the light curves are just updated, not generated anew.

Our astrometry is quite similar to the differential one, but we are not comparing our images directly. It is done with the help of package Munipack [4]. Instead of computing the astrometry and photometry for all images at once, we create our own set of calibration stars selected from an on-line catalog (currenty UCAC4 [5]) and try to calibrate our image's coordinates with these. We do the astrometry separately for each image, so at this point we don't mind whether the images actually overlap with each other or not.

This creates a small random error, which will cause all observations of one immovable object create a group of points with a Gaussian distribution of coordinates. The accuracy of our astrometry is around 0.2 - 0.3 arcsec, which is about half the size of our pixels as can be seen on Fig. 1.6. These groups of points are our clusters, and assigning the cluster identifiers to these points observed on the sky lets us to query for a light curve of one individual object on the sky.

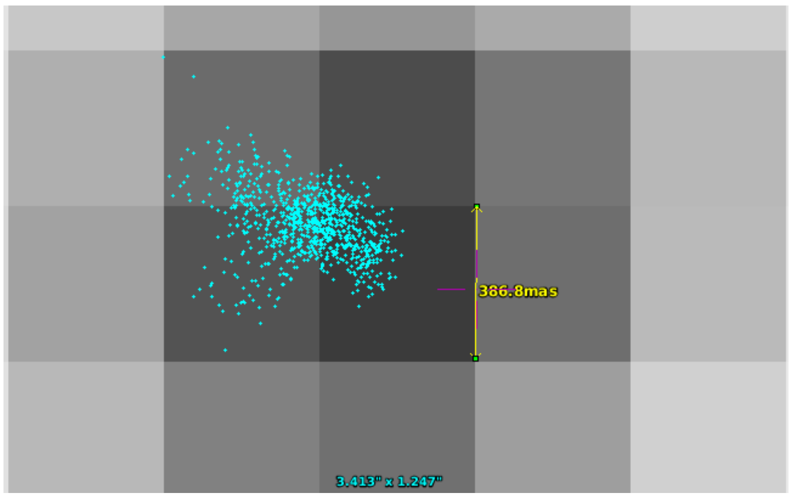


Figure 1.6: Astrometry accuracy

The product of the astrometry is just a binary table of observations for each image. When we take this data from all of the images, we have four hundred million observations which we need to cluster and assign to real physical objects (cluster them). And how we do that is the topic of this thesis.

### 1.3.1 Incremental survey and Transients

With our research, we can even create incremental survey. On the first iteration, we cluster all of the observations we have and create the first version of our catalog. The individual star observations on each image are assigned to objects in this catalog during the process.

After our survey has expanded, we take the images that were not processed yet and try to cross-match them to our previously created catalog. For the ones that did not match, we just run the catalog generation process separately, update the catalog with new identifiers, and assign the individual star observations.

This way we can even detect transients, such as supernovas, for which we didn't have a catalog identifier before. This is a great advantage against usual approach, when we just try cross-match against an on-line catalog and throw away the observations of objects, which are not in the catalog.

### 1.3.2 Publication

The results of this project will be presented on the IVOA Interoperability Workshop – Spring 2015 meeting [6].

Chapter 2

# Review of possible solutions, prove of concepts

As there exist no solutions which are solving exactly our problem, this chapter will not be a typical analysis. Instead we have to at least partially implement each possible solution as a prove of concept and then evaluate the result and decid whether this particular solution is feasible.

At the beginning of this chapter we define the background environment where we have to apply our solutions. There will be also all of the solutions which failed to some aspects. These solutions are analyzed carefully so we don't throw away an already partially or fully implemented solution.

The background needed will be defined in sections 2.1 and 2.2, all inacceptable solutions in sections Pure SQL 2.3, Array Databases 2.4, Apache Spark 2.5 and the final accepted solution is at the end of this chapter in section 2.6.

## 2.1 General background

There are many possibilities of how to store astronomical data and publish it to the world. The infrastructure we are using is inherited from my Bachelor's thesis [7]. The general data flow can be seen on Fig. 1.2. A more detailed view can be seen further on Fig. 3.2.

The main thing we will be focusing on in our thesis is how to properly cluster the astronomical data to produce desired light curves. We can see a star on Fig. 2.1 with cca 300 observations distributed around the star center. There is a closer look on Fig. 2.2, where we can see observations of one object over a period of time. The error in the astrometry here is around 0.3 arcsec, forming a cluster with a diameter cca 0.5 arcsec, as pointed out with the gauging line.

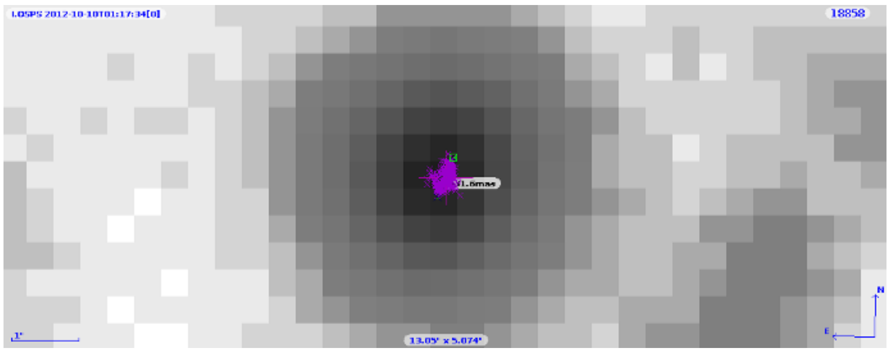


Figure 2.1: Typical star observation

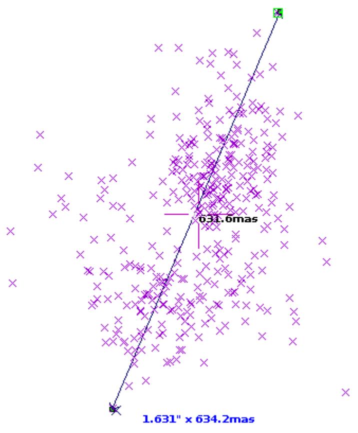


Figure 2.2: Cluster of star observations

## 2.2 Database background

The database model of the underlying architecture can be seen on Fig. 2.3. Each image is represented by 1 row in exposure table. Observations identified on this image will be kept in observation table, tracked by *obsname_id* to the exposure they were taken with. Then we will have to create identifiers for the actual objects and write them to *objcat* table. Each observation should be assigned to a catalog object via *id_cat* foreign key.

The *objobs_complete* view is used for easy access to the complete information about an observation. It joins data from exposure, observation and objcat tables, effectively linking image data (when the observation was taken, with which filter) with the actual observation (point on the image with all it's identified attributes) and the catalog identifier (to which real object this observation corresponds).

The *objobs_lightcurves* table based on the *objobs_complete* table and is used with the SSA protocol [8] to publish the light curve. This process is described in detail in my Bachelor's thesis [7].

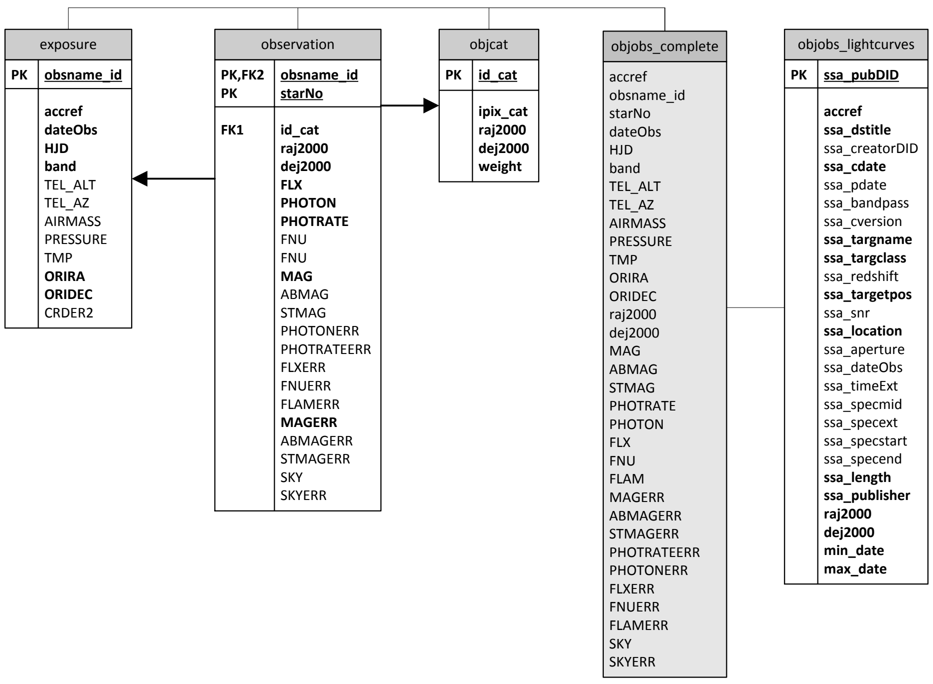


Figure 2.3: DB Model

## 2.3 Pure SQL solution

For summing it up, we are using PostgreSQL database for storing all the observation data as it is nicely supported and used in the astronomy area, has multiple sphere indexing algorithms implemented and can easily handle the amounts of data we are using. The main reason, however, why we are using this architecture is the GAVO DaCHS [9] package, which we are using for the data ingestion and publishing, and this package is built on the PostgreSQL database.

In this section we will describe all of the solutions which try to process all of the work inside the PostgreSQL database.

### 2.3.1 PPMXL Catalog

In the original solution described in my Bachelor's thesis [7], we did not create our own identifiers. Instead, we took our observations and tried to cross-match them with a deep enough on-line catalog. The idea is illustrated on image 2.4.

The individual observations on the left (one cluster is the same one as on Fig. 2.2) are cross-matched to catalog objects on the right.If an observation has no matching object in the catalog, it has no other way of assigning itself to a light curve and will be forgotten.

The best results were produced with the help of PPMXL catalog [10]. We managed to identify cca 70 % of our observations and within the 70 % there were still errors mostly caused by duplicate entries in the catalog. Throwing away more than 30% of our data is alone an unacceptable drawback.

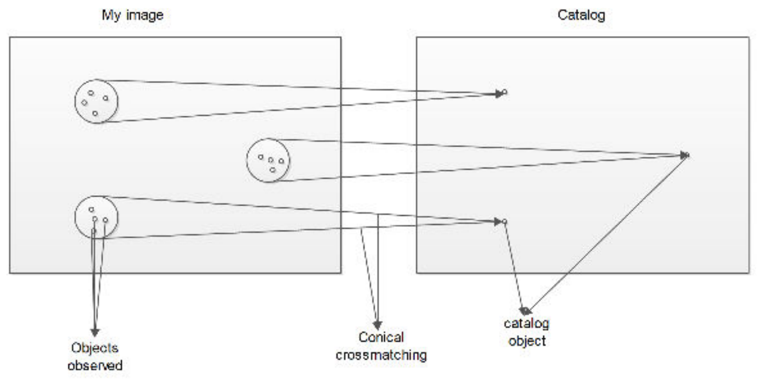


Figure 2.4: Catalog cross-match

### 2.3.2 ”Silver bullet” query

So we need to create our own catalog. The possible ”integration friendly” solution is to write a simple SQL query that would extract the identifiers we need. Because we already have the data ingested into a relational DB (PostgreSQL), it would be very convenient if we could create these identifiers inside the database, without the need of moving big amounts of data to external applications.

Thus, we created a *”silver bullet”* query, which is just taking the data from *observation* table, grouping them on some criteria, and writing the catalog identifiers to *objcat* table and the assignments of points via *cat_it* field back to the *observation* table.

#### 2.3.2.1 One big query

The query is based on the fact, that our clusters have cca 1 arcsec diameter. So we are grouping the data on a condition, that all of the points in one group can be connected by distances smaller than 1 arcsec. In practice this is implemented by a self-join which iterates over all points and for each one of them creates a group of points which are closer than 1 arcsec. Then if we iterate over groups of these points (members of my group), we are just observing this cluster from different points of view. From these we can choose the one that has the "best" view - meaning he sees the most points - has the largest group. This means just that the point is the closest to the center of the cluster and if we create an average of coordinates of all his neighbors, it will lie precisely in the center of this cluster. This approach can solve the corner cases too (e.g. three points in one line or a equilateral triangle).

This clustering algorithm can be implemented in PL/SQL, runs fast and has a very high precision for most of the patterns the observations can create on the image.

If we look at the complexity of this algorithm, we see that it is highly dependent on number of points in one cluster. If we are unlucky and we have a cluster of 1000 points which have exactly the same coordinates, for each one of them we get 999 coordinates with a distance zero. We have an $O(n^2)$ complexity where n is the number of points in one cluster. This counts for both computational and memory complexity.

Aside from the fact that our data has high density (on one image), it has also high density in the sense that we have a lot of overlapping images. We have at most cca 1000 points per cluster, at average cca 100. On the example of the whole dataset which has cca four hundred million points, one most basic point represented by 16B (right ascension, declination double precision), that means cca $6GB * 100^2 = 600TB$ of intermediate results. As we would like our algorithm to work for even bigger data sets, this is a major drawback of *"silver bullet"* query algorithm.

#### 2.3.2.2 Parallel smaller queries

Because the complexity problem is fatal only for the memory, parallelizing the query for smaller chunks of data will actually solve this problem. However, that creates another problem - how to divide the data into chunks?

The Q3C [11] IPIXes[10], which we are using for our observations, can be used for sorting the data and then just slicing the chunks sequentially. The spatial locality of these identifiers is quite good, but is not guaranteed (close points on the sky will have their IDs usually close to each other, but not always). We also have high probability of slicing the clusters on the edges of the chunks, and together with not guaranteed locality for the IDs, we will produce duplicate clusters close to each other. This problem can be reduced, if we iterate over points from the chunks only and look for their neighbors in the whole *observation* table.

If we were to accept these duplicate errors and continue with the testing, we will encounter the final bottleneck of this solution. The smaller the chunks are, the less memory we will use (and the more threads we can actually use on

[10]IPIX is a Q3C identifier for one point on the sky. It is a long integer.

one machine). But as we are already pushing PostgreSQL to the limits with the actual implementation (PL/SQL functions, looping over the table manually), the planner is very confused and will not produce any reasonable query plan (which would be loading the actually processed chunk and the whole observation table index into memory). Instead, it will do a lot of random disk seeks when searching the index for close neighbors and then using the whole memory for storing intermediate results. This query realization cannot be changed easily without changing the PostgreSQL source code. And distributing a PostgreSQL database over several disk nodes is very complicated and with already limited query plan quality, would not probably solve the problem.

#### 2.3.2.3 "Silver bullet" query summary

The *"silver bullet"* query solution is working really well with small data, but we encounter very big problems with scalability. This solution is limited either by disk capacity (storing intermediate results) or by the disk speed of random seeks (when parallelizing the query into smaller parts) and these limitations cannot be overcome, so we need to search for other solutions.

### 2.3.3 Iterative query

The limitation of *"Silver bullet"* query is not a problem of the SQL, it is just too high complexity of the actual clustering algorithm used. We can create a streaming algorithm which can process the data with a linear complexity. We will call this approach an *Iterative query.*

The idea is a quite naive sort of K-means algorithm. We iterate over all of the points in the database and for each one process the following condition.

Do I have a catalog identifier in a given range within my coordinates? If no, then I am a new cluster and the identifier is me. If yes, I add myself to that cluster and just update the cluster coordinates by a weighted average of mine coordinates and the ones already in that cluster.

This approach does not solve the corner cases when we don't iterate over the points in the right order. Example pattern here can be a cluster of directly 500 mas diameter. We start with a point completely on the left and create new cluster. Then it happens we take the second point completely on the right. The distance to the previously created catalog ID is exactly 1 arcsec far away, so we have to create a new cluster. Then as we iterate over the rest of the points, some of them will be assigned to the left cluster, some to the right. In the end we have 2 clusters close to each other instead of one whole.

These cases are in practice quite rare though, so we can accept them as error rate of this implementation. There is, however, a bigger drawback here which comes from using SQL. This language is just not built for looping over each single row of the data and doing operations on such level. For each point we have to do a couple of random seeks in the spherical database indexes before we find all of the points closer than the distance limit. With an average of couple ms per one cycle (which is really fast for close neighbors lookup in a relational database), it takes us weeks to process our whole dataset with very high disk usage the whole time. Under such load, we even discovered that PostgreSQL is quite unstable, because it was simply not meant for such usage.

#### 2.3.3.1 Iterative query summary

Even if all other arguments were beneficial, we simply cannot accept instable solution. Another option is of quite different matter.

### 2.3.4 IPIX iterating

One quite different approach still operating on the database level is the following. Instead of iterating over the points we can iterate over some measure defining the clusters itself. If we create a grid of squares with the same resolution as the cluster size, we can actually iterate over the grid and just ask what points are in this column. A sky indexing plugin in called Q3C [11] can actually do that quite efficiently. We call the strategy *IPIX iterating.*

On this Fig. 2.5, we can see how the sphere is partitioned to "squares" by the Q3C algorithm. It is based on quad tree cube partitioning of the sphere as can be seen on image 2.5.

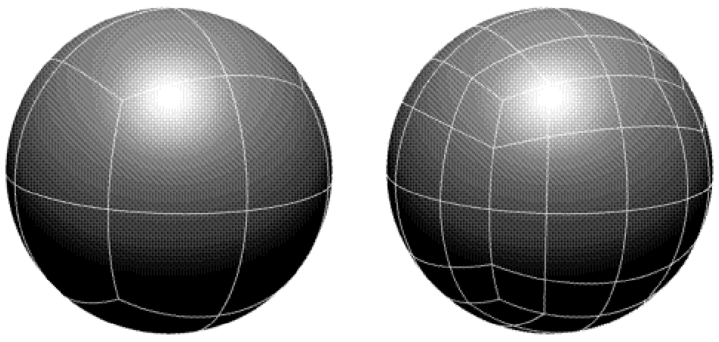


Figure 2.5: Q3C Pixels

After assigning points to the squares in the grid, we can join the ones which have points assigned and are next to each other, as that probably means that we sliced a cluster which lies on the borders of these squares. This approach however costs us precision in higher density areas where the cluster distances are comparable with the cluster sizes as we join more clusters together under one catalog identifier.

The big plus for the IPIX iterating method is it's speed. For the whole dataset (cca four hundred million observations) , it runs cca 40 minutes. It's complexity is $O(n)$ where n is the number of Q3C squares used for the sphere partitioning.

The accuracy is about the same as if we used the on-line catalog - around 70% of the object identifiers are identified correctly.

### 2.3.5 Combination

It would be very nice, if we could take advantage of such a fast algorithm, which provides high precision for cca 70% of our observations and falls off only where the star fields take up in density (clusters closer to each other). We could separate the rest of 30% and process them in some other slower algorithm. The problem is, that 30% will still take days to process with the iterative query 2.3.3 link and with the *"silver bullet"* query described in chapter 2.3.2 we wouldn't help ourselves, as it already solves sparse fields easily and has problems with the dense ones.

The biggest problem is that telling whether we actually joined the intermediate results correctly is actually as hard as the clustering problem itself. We tried several heuristics to separate the erroneous results, but every time with worse results than with other non-combination approaches. So combining the *IPIX iteration* with other sorts of clustering algorithms is a nice idea, but we didn't manage to solve to create a simple fast metric which would select all of the incorrectly assigned points or incorrectly identified clusters.

## 2.4 Array Databases

That's it, we are done with the relational databases. Clearly they are not meant for such kind of work and the argument of not transferring big amounts of data seems not that important after all. With freeing ourselves from the bonds of standard database, we can look for more exotic ones.

The array databases have very nice way of storing data. In general we are talking about column store (instead of standard row store) of data. The data is logically stored as sparse matrices, where we can search for close neighbors very effectively. That suits our data really well, as we have at least two dimensions (right ascension, declination coordinates) where we would really benefit from this strength. The array databases are even more effective, if we have more dimensions of the data (so we can add more parameters for our clustering). Another strength of array databases is that the sparse matrices can be easily sliced and partitioned between multiple physical database nodes.

The biggest advantage is that most of the solutions are highly tuned to keep most of the data in memory rather then disk, so we would have no more problems with disk random seeks when searching for closest neighbors.

We could even group our data spatially, assign them to concrete partitions and then run the clustering algorithm separately, knowing that all of the points we need are actually in our physical partition. When we add a small overlap to these regions on the sky, we will be sure, that no clusters will be split on the edges.

With this approach we could even use the high support for extensibility of for example *MonetDB* and write the clustering algorithm as an extension to the database core and call it directly from SQL, leaving the parallelisation management to the database itself. Or we could use *SciDB* with a simple C++ program that would transfer the data from database node, process it, and return the results. The database would again care about the parallelization over database nodes, where only a small geometrical chunk of sky will be stored and processed.

We will now have a look at some most famous array databases used nowadays. The *MonetDB* [12] and *SciDB* [13]. We will not make a detailed comparison of these databases in the sense of benchmarking or some detailed analysis. We will compare them only from our point of view, which is usability for storing, publishing and most of all clustering astronomical data.

### 2.4.1 Requirements

The main points which we demand from the array databases can be summarized into the following:

- Installation - As we are using Debian for our servers and that is not something to be changed in near future, we need a support for this system.
- Storage for astronomical data - We need to be able to store astronomical data at least as well as in PostgreSQL Q3C [11] spatial indexing schemes.
- Publishing of astronomical data - If we decide to migrate to array databases, we have two possibilities. We might migrate only part of our data and functionality, or we migrate all of the functionality. It would be very complicated to integrate the array database with PostgreSQL in order to keep part of the data here and part of the data there. When we decide to migrate fully though, we need to implement the protocols for publishing data on top of an array database.

  We are currently relying ourselves on the GAVO DaCHS package [9] which implements majority of the IVOA VO [14] protocols and works currently with PostgreSQL only. If we cannot transfer the data model without any significant changes, it will be very difficult to accommodate the processes above them accordingly.

  In the end, we decided not to use the array databases only partially, as that would bring us more troubles then benefits. As this decision was also connected with other aspects, we won't terminate our analysis here, but try to compare other aspects needed for our cause too.

- Data migration from PostgreSQL - We need an easy data migration from our data centers which are currently running on PostgreSQL.
- Extensibility for clustering - We need to be able to write an extension to the database or call a program from the database which actually clusters the data. Otherwise we would not benefit from the fact, that database can handle the parallel processing on each data partition (i.e. database node) for us.
- Fast neighbor lookup - This is not needed only for the clustering algorithm, as obviously it can run outside the database as a stand-alone program. But we actually need it for any kind of data publishing to the outside world, as most of the astronomical queries will be spatially based.
- Easy data partitioning - We need a simple way to control the partitioning of our data across the database nodes. To be the distributed clustering algorithm truly effective, we need to ensure high spatial locality on the data. Each partition has to represent an area on the sky. In other words, one cluster is permitted to have it's members stored only at one particular data partition,i.e.,physical database node. We would be relying on that if we decided to process the data in smaller chunks. It would be much easier if we could process each one separately without the need of communication with the others.

### 2.4.2 *MonetDB*

We made the most detailed analysis for *MonetDB* [12] as it is really close to what we need.The *MonetDB* is written in C, supports mainly Linux systems and is very easy to install on Linux.

*MonetDB* is quite advanced as it has been developed since 1993. Amongst its biggest strengths belongs high performance vertical fragmentation, automatic and adaptive indices and run-time query optimization. In other words, we don't have to worry for example about indexing the data correctly, *MonetDB* will create the indices itself, statistically based on the queries which will be requesting the data.

#### 2.4.2.1 Installation

If we want to install the database on one server, it is very easy. For a Linux-based system, we can install it similarly to any relational database - as a distributed package.

#### 2.4.2.2 Storage for astronomical data

*MonetDB* is based on a an exotic way for data storage. It stores tables using vertical fragmentation (storing each column as one table), called *Binary Association Table (BAT*. Each table is stored using a key-value mechanism, where the keys are always a dense sorted list. Both the keys and values are stored as memory mapped files, which ensures very high performance with data access times. In the values we can store anything - in case of variable-width types the value is separated into into a reference (offset) and the real value of variable length.

For the right indexing (which in case of *MonetDB* means just sorting) the data, we would need a sphere partitioning algorithm. We would also need to implement the geometrical queries ourselves, whereas for PostgreSQL we can use already tested and reliable plugins like Q3C [11].

There are actually several User Defined Functions in *MonetDB* under LSST [15] package, based on HTM spatial indexing [16]. For all out publishing of astronomical data, however, they are incomplete and last but not least poorly documented. We would not choose HTM indexing for our data either, because it's complexity climbes with the indexing resolution and it does not guarantee the pixels to be of same size.

There is also a project which tried to use *MonetDB* for SDSS survey SkyServer [17], where they actually implemented spatial queries based on zones. This algorithm is described in article [18], but these methods deviate quite significantly from our current solutions and integrating them into our clustering algorithms would be very difficult.

#### 2.4.2.3 Publishing of astronomical data

Here comes the real problem. As we mentioned above, transforming the data may be difficult, but it can be done. But it means we would have to rewrite the implementation of the protocols responsible for publishing of our data.

As I have not found any good enough alternative to GAVO DaCHS [9], it would mean implementing the support for another database layer to this package directly, which is simply more effort than we would like to invest only for the sake of better clustering results when all other functionalities we already have in the current implementation using PostgreSQL.

#### 2.4.2.4 Data migration from PostgreSQL

We can migrate the data through two channels. First one is through *pg_dump* utility, which gives us a database dump of all the tables and data. As not all the data types in *MonetDB* and PostgreSQL match directly, we have to manually repair the dump file according to the *MonetDB* data types. There are some open source tools which do that automatically, but for bad experience with such things, we chose another, safer approach.

Second approach is based on export to simple text (e.g. CSV) files, define the tables in *MonetDB* manually, and just ingest the data from CSV. This approach is safer, as we have complete control over the data types used in *MonetDB* and we solve all the conflicts before-hand, separated from the data.

Each PostgreSQL and *MonetDB* have built-in tools which can export and import CSV files, so there is no problem with this requirement.

#### 2.4.2.5 Extensibility for clustering

Construct named *User Defined Functions (UDFs)* are used in *MonetDB*. These functions are mostly written in pure C and wrapped by *MonetDB* inner assembly-like language called MAL. We can write the functions directly in MAL, but that is not advised, as it is not an easily debuggable language. The MAL instruction to which we can link a C function has to be mapped to an SQL function, which can call the functionality directly from the SQL front end.

This architecture is quite complicated, but it allows us to call the BAT functions directly from the C code. It is very convenient, as now we can just pass points which we want to cluster to the C function, and it can write the results directly to several other tables (e.g. cluster IDs and cluster assignments of the original points). These functions are then stored with the *MonetDB* source code, and using the bootstrapping algorithms, they can be included in the whole build and made part of the *MonetDB* distribution.

Originally, we also thought we could use the very interesting *MonetDB* functionality - the actual integration of R to the database itself. As R contains a lot of clustering functions and modules, it would be very nice if we could cluster the data in the database directly. There is also an example of how to use K-means in *MonetDB* using the R module [19].

But - as the conventional clustering algorithms fall off with both time (high complexity) and quality (usually merging clusters which should not be merged) for Big data, we would have to divide the data into smaller chunks which could be computed in parallel. This would be very nice when used in synergy with the database partitioning. We could run the K-means in parallel for small parts of the data where we could ensure high IO bandwidth if each partition would be saved on a separate hard disk.

#### 2.4.2.6 Fast neighbor lookup

As we already mentioned in the *Storage for astronomical data* argument, the data is stored in separate tables for each column. If we take the two most important ones for clustering - the coordinates right ascension and declination - we can see that the neighbor lookup will be fast even without explicit indexing.

As each table is stored as a separated head which contains sorted IDs of the actual values, if the data is sorted geometrically the lookup will be very fast. Ensuring this data locality is quite a hard problem, but there is nowadays a number of sky or sphere indexing algorithms, which we can use.

#### 2.4.2.7 Easy data partitioning

In the official documentation of *MonetDB* [12] there is no word of partitioning support. There are mentions of possibilities and experiments in the SDSS SkyServer paper [17].

We still can use *MonetDB* for clustering the data using the *Iterative query* approach mentioned in section 2.3.3.

We can still sort the data on disk accordingly to the geometrical locality on sky, as this helps us with finding the nearest neighbors. We decided to use HEALPix [20] library to take care of that. It has a very nice future that the pixels partitioning the sphere are of equal size, thus the data will be distributed uniformly in the grid. We will be talking about that in the separate chapter 3.2.2.

We discovered later that such synergy with *MonetDB* is not actually possible, or at least very complicated. The *MonetDB* is written entirely in C so the bootstrapping of the *User Defined Functions* mentioned in *Extensibility for clustering* point can only work with pure C functions. HEALPix library has APIs for C++ as well as for pure C, however, a key functionality needed for the data partitioning is only available in C++. The thing we need here is to be able to ask for neighbors of a pixel to compute overlaps (see chapter 3.2.2).

We could declare the API functions of our C++ program extern C then, but that would still mean we have to build and distribute our C++ application separately from the *MonetDB* code. This is a very high cost we would have to pay and collides with the point *Extensibility for clustering* too.

#### 2.4.2.8 Conclusion

We have discussed the individual points we need from the database above and here comes the conclusion. Strong points of *MonetDB* are the Installation, data Migration and the fast neighbor lookup. The storage of astronomical data is fine, but we would have to implement our own plugin for querying the astronomical data. We could base this plugin on HEALPix [20] data indexing but we could not distribute the plugin as a part of *MonetDB*, as it cannot be written in pure C.

In the end, all comes down to the *Extensibility for clustering* requirement. As much as *MonetDB* is opened and easily extensible by pure C functions, it cannot be extended by C++ code.

Overall, *MonetDB* seems quite mature, but the fact of inextensibility by C++ (which we need for HEALPix, more info in chapter 3.2.2) and the lack of horizontal partitioning make it inappropriate for our case.

### 2.4.3 *SciDB*

Another option we considered here is the SciDB [13]. As we realized quite quickly that it does not fit for our cause, the analysis will only be that much thorough to justify our decision.

The *SciDB* is a partially commercial project where the linear algebra module (which we could make very good use of) is available only in the commercial version. In the scientific open-source version there are all of standard functionalities we need though so we will not dismiss it too quickly.

#### 2.4.3.1 Installation

*SciDB* has a very nice installation automation script, which can install it to multiple database nodes (even on different servers) easily. However, the *SciDB* packaged distribution is very sensitive to the OS version. Installation from packages requires RedHat / CentOS, and Ubuntu to the 14.9 release. If we have another system version (in our case Debian 7), we have to install manually from the sources. As we are considering extending these sources, this is not too big obstacle for us.

#### 2.4.3.2 Storage for astronomical data

*SciDB* has been founded as a part of the LSST [15] project originally. In the flow of time, however, they deviated from this original purpose and formed a stand alone partially commercial project. LSST chose to use massive distributed data center based on MySQL nodes thereafter.

The examples of using *SciDB* for geospatial as well as astronomical data can be found over the internet. The logical storage of data is that we have a multi-dimensional array of cells, which can hold an arbitrary number of attributes. These attributes can be of any (even user-defined) type.

Data transformation will be needed, but no more complicated than when using *MonetDB*. Actually it will be a lot easier, as we can define the cells as coordinates and any attributes assigned to those coordinates can be hold in an array of attributes directly.

#### 2.4.3.3 Publishing of astronomical data

The real issue here again comes with the support for IVOA VO [14] protocols to publish the data. The problem is that even if we managed to change the data model to the better, we simply cannot afford to invest time connected with adding another database support to the GAVO DaCHS [9] package.

#### 2.4.3.4 Data migration from PostgreSQL

As we already decided to use CSV export from PostgreSQL, *SciDB* supports this future fully. The *pg_dump* file would require corrections of types, the same as with *MonetDB*. See *Data migration* in previous chapter 2.4.2.4.

#### 2.4.3.5 Extensibility for clustering

The *SciDB* supports multiple types of extensions - user-defined aggregates, user-defined array operators and user defined functions. The last are the ones that interest us most. These are scalar functions which accept variable number of arguments of different data types, and return another data type.

We would have to define our own types for passing the input data (observations) to the function, as well as for the returned value, which would be the original points grouped into clusters. Using parallelism inside this function shouldn't be a problem and *SciDB* currently supports only C++ for user defined functions, which perfectly suits us, as we need it for our HEALPix [20] library.

#### 2.4.3.6 Fast neighbor lookup

*SciDB* storage model ensures, that neighbor lookup will be fast as long as the neighbors will be physically close in the multidimensional matrix. We are talking about the cells in this matrix, which are representing point coordinates in the sky.

As we already mentioned above, to enable this fast lookup by ensuring data locality, we need to be able to sort the data accordingly. Thus, using a sphere indexing algorithm like HEALPix [20] is needed and the extensibility of *SciDB* allows us to do that.

#### 2.4.3.7 Easy data partitioning

*SciDB* has a very transparent system of logical chunks, which are dividing the multi-dimensional arrays into groups. These can be easily partitioned over separated media. There can be also defined overlaps between these, so if we can implement a clustering algorithm working on one node, it will not require data transfer from other nodes.

#### 2.4.3.8 Conclusion

The strong points of *SciDB* are the data partitioning, extensibility and fast neighbor lookup points. Other points are acceptable, but very controversial for us is the publishing of the actual data. As we don't like to store part of the data in PostgreSQL and part of it in *SciDB*, we would have to integrate *SciDB* with implementation of IVOA VO [14] protocols.

### 2.4.4 Array databases conclusion

As much as we liked the idea of moving a part or our whole data center to array databases because they work with astronomical data in a more natural way than relational databases, the costs are too high. We are not discontent with the nowadays functionality provided by standard database such as PostgreSQL along with it's sky indexing plugins, the only reason we are trying something different is their inappropriateness for clustering algorithms.

The original thought was that we could migrate only the clustering algorithm to the array database, but that is such an overkill if we compare it to a stand alone C++ program. As we agreed it is not profitable to move all of our functionality to the array databases, they are not useful for solving our problem at all.

## 2.5 Apache Spark

Next thing we considered is using the Apache Spark [21] language, as it can be used very efficiently for parallel cluster computing (here by the term cluster we mean server cluster). We dismissed this language because it's linear algebra module has small support for clustering algorithms (in fact there is only K-means algorithm) and if we are using a lower level language, we would like to do a benchmark for multiple clustering algorithms on the data too.

## 2.6 C++ application

So the last resort for us are the oldest, but still most efficient tools available. We will implement the clustering algorithm as a stand alone C++ application, to which we will transfer the data. PostgreSQL is very fast with CSV exports/imports, so we stick with that idea and the data will be transfered in this format. We chose C++ explicitelly, because it is fast, relatively easy compared to pure C and HEALPix [20] library supports it, which we will bring up in chapter 3.2.2

We will not change the original database model, so the input will be only one table of observations, which can be represented by right ascension and declination, plus some unique identifier tracking the observation back to the image file it was identified on.

The output will be a list of catalog coordinates of our light curves with some catalog IDs and a list of the original observations assignments.

As we don't have to limit us because of the database possibilities anymore, we can implement any clustering algorithm we like. We decided to implement the *Incremental query* defined in 2.3.3 as a base for our benchmarks, because if we manage to load all the data set into the memory, there will be no problem with iterating over each particular point. This algorithm has it's own limits though, and as there are lots of C++ clustering libraries available, we can try to integrate and test them for hopefully better results.

### 2.6.1 Dividing the work

The main problem of most of the clustering algorithms (K-means, Expectation Maximization, Hierarchical algorithms, etc.) is their complexity. With growing number of clusters in the data, most of these algorithms run very long and, specifically for our data, tend to join more clusters together. With an average number of one hundred points per one cluster, we have cca four hundred million clusters in our dataset. None of the more complex clustering algorithms can process such data with a reasonable result - at least not as whole.

So the key here is to divide the data into small chunks not only for helping with the complexity, but also for improving the actual results of the clustering algorithms. When each instance of the clustering algorithm runs for a data set with 5 clusters, each with 100-1000 points, it still provides very good results. It would be even better, if we could be sure, that each one of these chunks actually contains all of the points of it's clusters, that we cannot find any sliced cluster on the edges of these.

There is a quite simple solution because of one fact we know of our data. The actual errors in the observation coordinates come from the inaccuracy of astrometry computed for these objects defined in chapter 1.3, i.e., the size of clusters is smaller than this inaccuracy. This technique is the same for all of our data, so we can define a threshold which is greater than the known maxium size of a cluster and use it to define overlaps at the edges of our chunks, which will be wider than this threshold. That ensures with one hundred percent assurance, that we cannot slice a cluster. We will show this in the Realization chapter in more detail in chapter 4.

### 2.6.2 GPU computing

There is also the possibility to accelerate the computational process by moving the most expensive processing to GPUs instead of processor. However, to be this enhancement really efficient, this requires very intensive memory micro-management. Also, the advantage of GPUs originates from the ability to process high amount of small tasks which require very little communication with the memory. The clustering algorithm has to be deeply analyzed and bottlenecks quite different from standard processor computing have to be identified and addressed.

The GPU computing can be actually implemented in an existing C or C++ application, transforming only small parts of the program, where we know from the testing it is efficient. This is possible for example with an nVidia CUDA Toolkit [22]. This toolkit of libraries can be called as simple C++ functions.

There are several solutions implementing K-means using a GPU acceleration. We are planning to take a little different approach, where we want to decrease the number of points for one K-means run and parallelize these tasks. Parallelization inside these tasks would not be that efficient, as it is better if we effectively decrease the complexity by assigning less points per one K-means instance, than to parallelize the one K-means instance for a bigger data set.

After considering all of the above mentioned facts we decided against implementing GPU optimization in our application.

Chapter 3

# Design

All of the aspects of possible solutions and their results, mentioned in the previous chapter, were carefully considered and a final design decision was made. The accepted and fully implemented solution will be described in this chapter.

We will implement the catalog generation as a stand alone C++ application. This application will take data from PostgreSQL, produce results, and these will be transfered back to the database. With this approach we don't have to change the underlying architecture of the data manipulation at all.

## 3.1 MPI vs OpenMP

The dimensionality of our data is not very high (2 coordinates, photon flux or magnitude possible as 3rd dimension), so we can load the whole data into memory and process it there. There is a possibility of streaming the intermediate parallel task distributions on disk, but the problem is that we cannot be sure that a task is complete before we iterate over all points in the data set.

The processing cost for the individual tasks will be small in the terms of memory. It depends on the task size, but compared to memory amount needed for constructing these tasks, it is very small. The benefit of distributing the work over multiple nodes is not high, as the tasks are supposed to be small. Another reason why we'd like to try the threads approach is that it is easier and faster to implement. And the easies way to implement a thread-based producer consumer application is OpenMP library [23], so that is our final decision for parallelization.

The parallel tasks will be processed by an arbitrary number of threads and the results collected by a master thread and written back to disk.

## 3.2 Design details

We can see the complete pipeline used for the light curve generation on Fig. 3.2. The main part of our thesis is about the Clustering swim-lane.

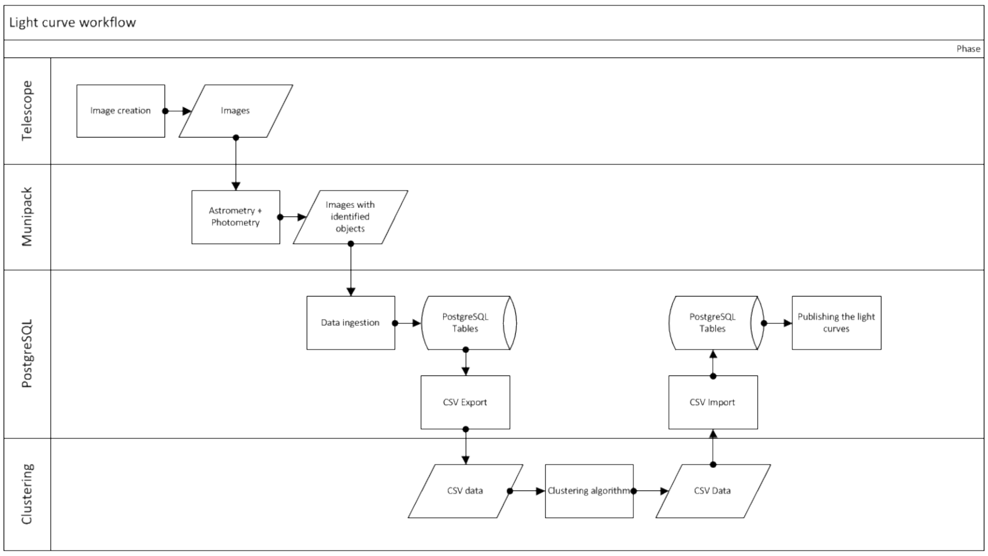


Figure 3.1: Light curve generation workflow

### 3.2.1 Clustering algorithms

We also had to analyze multiple clustering algorithms and choose which ones will be used within the C++ application. The possibilities here were a simple K-means, Expectation maximization based techniques such as Gaussian mixtures, agglomerative or divisive hierarchical clustering or last but not least some clustering algorithm of our own.

We also considered integrating R [24] with the Seamless R and C++ Integration [25]. There is a very large variety of clustering algorithms implemented in R. However, most of these are usable only for testing on small amounts of data, their complexity grows very quick. We also don't have any possibility to influence the behavior of these R modules, they have to be taken as-is.

Final reason why we favored C++ library over the integrating R modules was the usability for big amounts of data. C++ is way faster than R in equal computations and has the advantage, that we can debug and potentially change or even repair the library we are using.

In the end, we decided to implement the *Iterative query* algorithm defined in chapter 2.3.3, which will be used as a benchmark to other clustering algorithms. Then we decided to use a library for randomized K-means algorithm [26] and the Armadillo C++ library [27]for EM algorithm based on Mahalanobis distance.

### 3.2.2 HEALPix [20]

We also had to consider which sphere indexing library will we use to work with the spatial data. This cannot be done without a library because we need a complex way of building the chunks used for parallel processing. Another reason is a pre-implemented optimized way of computing angular distance between two points on a sphere.

The possibilities here were HTM indexing algorithm [16], HEALPix or Quad Tree cube algorithm [11]. We will not do a detailed benchmark here, as all of these algorithms are quite effective. The decision was simple in the end.

We chose HEALPix because of these major reasons:

1. HTM [16] has higher complexity for higher resolutions of the sphere tessellation, whereas HEALPix [20] has constant complexity in this direction. We will work generally only with the high resolutions with the clustering algorithm. The pixels have different sizes at different areas of the sphere.

2. Quad Tree Cube [11] will also produce different sizes of the individual pixels, thus it would be more complicated to distribute the data into chunks uniformly.

3. HEALPix [20] can distribute the data uniformly, because it's pixels are of equal sizes on the same resolution. It is also very popular lately and has been implemented into Aladin VO [28] client, which makes it much easier to test and compare our results.

## 3.3 Functional requirements

We can summarize the results of our analysis to the following requirements on our application.

1. Data input will be processed from CSV format. This input file will have 4 columns identifying an observation. The *obsnameID* and *starNo* are used as a composite primary key for the observation table. The exported columns will be the following:

    a) *Right ascension*

    b) *Declination*

    c) *imageID*

    d) *starNo*

2. Data ouptut will be again in CSV format and will take place in two files. First one will represent the catalog with following columns:

    a) *Catalog ID*

    b) *Right ascension*

    c) *Declination*

    And the second one will represent the observation mappings to these catalog files, having the following columns. Again, *obsnameID* and *starNo* are used to identify observation in database. The columns defining our catalog are:

    a) *Catalog ID*

    b) *imageID*

    c) *starNo*

3. The program will be able to create a groups of the observations, which can be processed in parallel completely separately without degrading the results. This will be accomplished via the HEALPix library [20].

4. It will support several clustering strategies for comparison. Specifically these will be:

    a) Our own incremental strategy described in 4.1.3.1

    b) Several variants of K-means from KMLocal library [26].

    c) Expectation Maximisation strategy from Armadillo C++ library [27].

5. The solution will be integrated into our current data center. This integration will be implemented by scripts for exporting and importing CSV data for PostgreSQL database.

## 3.4 Application model

The UML diagram for our application can be seen on Fig. 3.2. We will explain the design on a typical run of our application.

The main part of the application logic is contained in class *ClusteringController*. The entry point function just parses the command line arguments and passes them to the the controller. Then it calls the run function. The *ClusteringController* class holds instances of two other controllers - the *CsvOperator* and *ChunkOperator*.

*CsvOperator* is responsible for working with input and output CSV files. The *CsvOperator* will parse the input file into a vector of *Coordinates* and pass it to the *ClusteringController*. This vector is logically owned by *ChunkOperator* which works with this data most as with *obsCoords* vector.

Then, the *buildChunksFromCoordinates* method is called on *ChunkOperator*, and the result is stored in two maps - the *obsInCells* and *obsInOverlaps.* In these maps, *Coordinates* are assigned to their HEALPix [20] pixel IDs of selected resolution, along with their overlaps if there are any. This function is described more in chapter 4.1.2.

For each of these HEALPix pixels, a *ClusteringTask is created.* This clustering task has a strategy by which it is to be solved. Each strategy is holding a list of pointers to the *Coordinates* it has to process. The *ClusteringController* ensures that tasks are processed in parallel by a number of threads, which has been allocated to this run of our program. Each clustering task will have it's own results stored in *cluster_map* type, which is a map of *clusterIDs* pointing to a list of it's members. The *clusterID* is too an instance of *Coordinate* struct representing the cluster medoid.

These individual task's results have to be collected into an overall result of the clustering algorithm. This is done sequentially after the parallel phase of actual clustering has been finished. We iterate over the tasks, throw out the duplicate results of neighboring tasks, or merge them into common results.

The overall result is kept in the *ClusteringController's* result, where we ensure, that a cluster will not have duplicate members, i.e., the list of cluster members is a set of *Coordinates.*

### 3.4.1 Configuration

The application will take the following configuration. The values values here are optimal according to the testing results. The rest of configuration, such as K-means configuration, will be listed later.

```
[parallelOptions]
bigPixelNsideExp=15      ;Parallel task resolution
overlapPixelNsideExp=18     ;Overlap pixel resolution

[resultOptions]
IDNsideExp = 29          ;Catalog ID generation resolution
clusterDuplicatesArcSec = 0.5 ;Distance at which two clusters will
     be identified as duplicates <arcsec>

[incrementalStrategy]
catalogIndexNsideExp = 17   ;Resolution of the index inside one
    task
clusterRadiusArcSec = 1        ;Distance at which I will add a
    point to the cluster <arcsec>
```

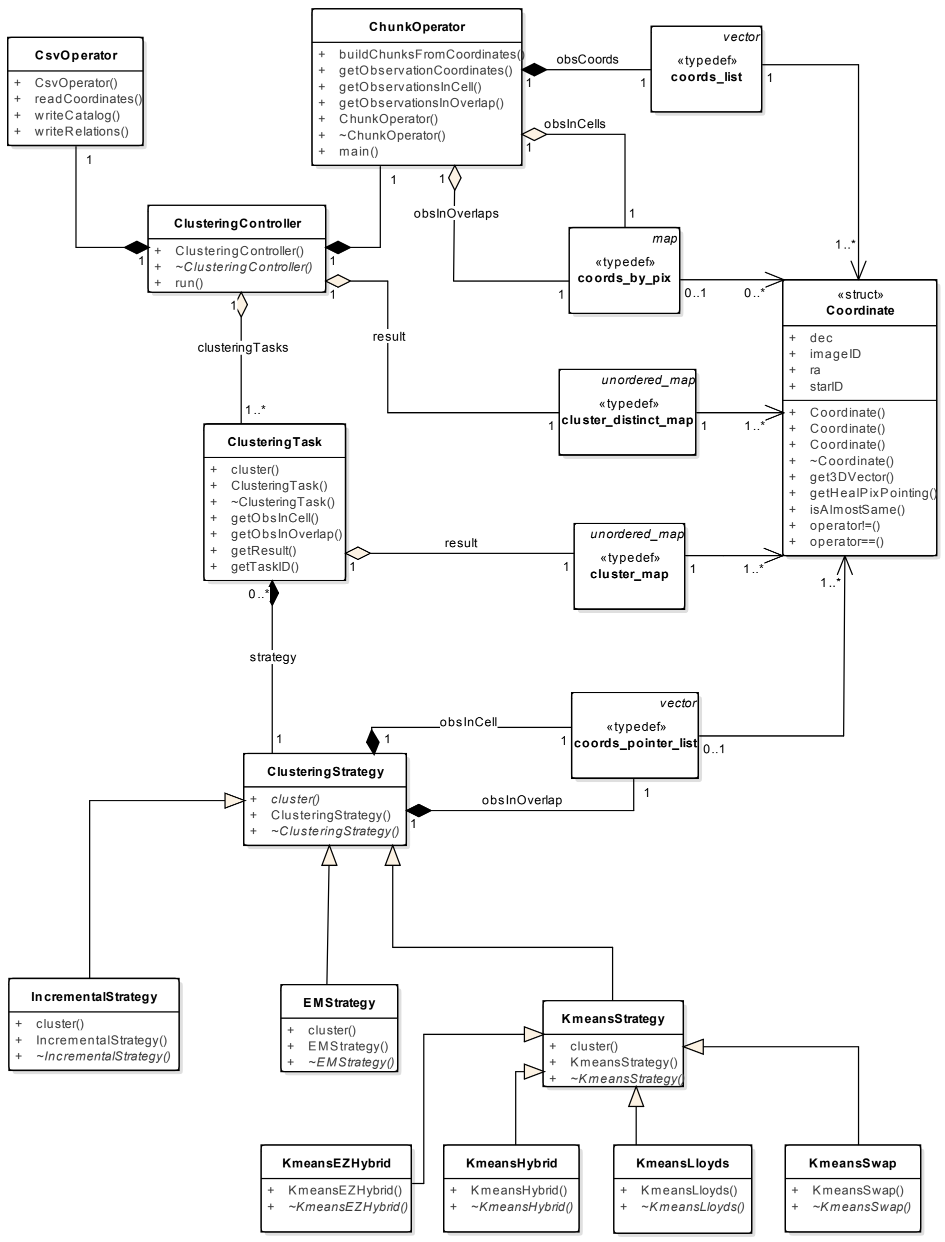


Figure 3.2: C++ application UML class diagram

## 3.5 Time Complexity

The most crucial application phases have the following complexity:

1. Building chunks - This phase has a $O(n)$ complexity, as it touches every observation once.
2. Clustering phase - Dependent on a clustering algorithm, this is either $O(n)$ for a simple strategy, or $O(n^2)$ for other strategies implemented (K-means, EM)
3. Collecting results - $O(n^2)$ for merging the results. Depending on the task size, we have to check the neighboring tasks for duplicate clusters and in worst case, merge them all.

## 3.6 Scalability

The scalability of our solution results from the chosen architecture. As we are using an in-memory solution based on threads rather than individual processes, we are favoring a shared-memory architecture. The memory limitations for the whole application run are around 8 GB per 100 million points. This relation is linear, so for 1 billion points we would need an 80 GB RAM machine with one processor to do the division into parallel tasks. These individual task's complexities depend on the strategy used for solving these tasks, but the idea is to keep them small for better efficiency and result quality of the clustering algorithms beneath.

Chapter 4

# Realisation

As we already deduced above, the most appropriate solution to our problem is a stand alone C++ program, which will be integrated into our current data center. We will comment the implementation in this chapter, pointing out the interesting parts, which solved the problems the previously mentioned solutions could not. We will also document here the way of integration with our current solution of the original data ingesting and publishing the light curves.

## 4.1 Implementation

In this chapter we will present the interesting parts of our solution, with examples of C++ code.

### 4.1.1 HEALPix usage

We are using the HEALPix library [20] to create the parallel tasks, which can be processed individually, without need of further communication between themselves. The key aspect for defining such tasks is the *Nside* parameter of the HEALPix grid. The *NSide* is specifying the size of HEALPix pixel. The number is always $2^N$, where N can range from 1 to 29. For our case, the most convenient are around 18, which specifies a pixel approximately the size of one cluster. We use these pixels for defining the overlapping the region of our parallel tasks, which are equivalent to the area of a HEALPix pixel of a $Nside = 18$. The *Nside* used for the parallel tasks is meaningful between 10 and 15 (for 16 and more the overlap is actually bigger than the task area itself).

For simplicity, the *"task size"* term will be used for the exponent of an *Nside*. For $Nside2^15$, the *task size* will be 15, for $2^29$ the *task size* will be 29. Whenever we will talk about the *task size*, we will refer to this exponent, not the actual *Nside* value.

### 4.1.2 Creating spatial chunks

This section is describing functionality in *ChunkOperator's buildChunksFromCoordinates* function. We are mentioning here the most interesting parts of the code. For the whole function, you can look at appendix C.1.

First we transform our coordinates to the system HEALPix [20] is using and then comes the interesting part. We create two HEALPix bases, first one with the resolution of the task spatial partitioning (i.e. How big will be the area for one individual task) and the second one with the resolution of the overlaps.

```
HEALPix_Base2 base1(base1Nside, NEST, SET_NSIDE);
HEALPix_Base2 base2(base2Nside, NEST, SET_NSIDE);
```

Then we iterate over observations and decide in which cell it lies.

```
int64 idx_lores = base1.ang2pix(observations[i]);
```

Then we check, whether this point does not belong to an overlap (based on the base2 finer grid) of the neighboring cell (based on the base1 rougher grid). The big cell (i.e. task) can have up to 8 neighbors in the HEALPix geometry.

```
// now check whether the surroundings of the observation touch
    neighboring jobs
        int64 idx_hires = base2.ang2pix(observations[i]);

        fix_arr<int64, 8> neighbors;
        base2.neighbors(idx_hires, neighbors);
```

We iterate over these 8 neighbors and if it happens that the *nbidx_lores* (i.e. big pixel ID next to our small pixel used for overlaps) is not actually the current *idx_lores* then we add our current point to this neighboring big pixel's overlapping region.

```
int64 nbidx_lores = base1.ang2pix(base2.pix2ang(neighbors[j]));

if (nbidx_lores != idx_lores) { // touches a neighbour cell
  if ((*obsInOverlap)[nbidx_lores].empty() ||
          (*obsInOverlap)[nbidx_lores].back() != &(*obsCoords)[i])
              {
        (*obsInOverlap)[nbidx_lores].push_back(&(*obsCoords)[i]);
  }
}
```

By this algorithm we can distribute any kind of spherical coordinates into chunks of equal size (equal area on the sphere) with overlaps of variable size in a linear complexity (touching each observation only once). This was the biggest obstacle in parallel clustering of our data.

### 4.1.3 Actual clustering

Each task remembers the strategy[11] by which it should be solved.

The parallel processing of each task is resolved by OpenMP library [23]. The clustering loop looks like this. The *omp_set_dynamic(0)* is used to enforce that the number of threads we provide will be actually used for the computing.

```
omp_set_dynamic(0); // Explicitly disable dynamic teams
omp_set_num_threads(this->noThreads);
#pragma omp parallel for if (this->noThreads > 1)
    for (size_t i = 0; i < clusteringTasks.size(); i++) {
      ClusteringTask *currTask = clusteringTasks[i];
      currTask->cluster();
}
```

#### 4.1.3.1 Incremental strategy

We wrote this strategy as a benchmark for the other strategies such as K-means. Those were taken from 3rd party libraries mostly, so we will not document them here.

This strategy is basically re-written *Incremental query* from chapter 2.3.3. We iterate over observations in our cell, then we iterate over the ones in our overlap in the same way and finally remove clusters we decide to be incomplete (i.e. they were sliced on the edges of the overlap region). We remove the incomplete clusters when using other strategies too.

```
void IncrementalStrategy::cluster(cluster_map &taskResult) {
    processObservations(obsInCell, taskResult);
    processObservations(obsInOverlap, taskResult);
    removeIncompleteClusters(taskResult);
}
```

The code for processing each individual observation is simple too. For each observation, we check whether we don't have a catalog ID close to it already. If there is one, we just update it and add ourselves to that catalog ID. This is done in *findAndUpdateNeighbor* function.

If we could not update the catalog, that means we have to create a new identifier with the same coordinates as the current observation. We set the *imageID* and *starID* to -1 as this observation is not originally from database. The *catalogIndex* is used for indexing already processed observations of this task in memory for faster neighbor lookup.

```
void IncrementalStrategy::processObservations(coords_pointer_list
    *obsList,
        cluster_map &taskResult) {
    for (coords_list_it it = obsList->begin(); it != obsList->end
        (); it++) {
```

[11]Strategy is a software design pattern used for solving same task by different ways of doing it

```
        Coordinate * currObs = *it;
        int64 indexID = indexBase.ang2pix(currObs->
            getHEALPixPointing());

        bool catalogUpdated = false;
        catalogUpdated = findAndUpdateNeighbor(indexID, currObs,
            taskResult);

        if (!catalogUpdated) {
            Coordinate * clusterID = new Coordinate(currObs->ra,
                currObs->dec, -1, -1);
            taskResult[clusterID].push_back(currObs);
            catalogIndex[indexID].push_back(clusterID);
        }
    }
}
```

The *findAndUpdateNeighbor* method gets close points from the *catalogIndex* (it does not search all of the points, only the ones in neighboring pixels). Then it iterates over these coordinates and for each one computes distance to it. If the distance is smaller than a configured value, it is added to that cluster. If not, we return false and a new cluster is based on it.

```
bool IncrementalStrategy::findAndUpdateNeighbor(int64 indexID,
        Coordinate * currObs, cluster_map &taskResult) {
    coords_pointer_list clustersCloseToMe;
    getClosePoints(clustersCloseToMe, indexID);

    for (coords_list_it it = clustersCloseToMe.begin(); it !=
        clustersCloseToMe.end();
            it++) {
        Coordinate * currNeighborClusterID = *it;
        double distance = HEALPixHelper::computeDistance(currObs,
            currNeighborClusterID);
        if (distance < Config::clusterRadius) {
            updateNeighboringCluster(currNeighborClusterID,
                currObs, taskResult);
            return true;
        }
    }
    return false;
}
```

### 4.1.3.2 K-means strategy

We used the K-means algorithm from library KMlocal library [23]. The metric used for optimization here is the Euclidean distance of our coordinates to the K-means centroids called distortion. We are using average distortion for better compatibility with the elbow method. Along with minimizing this function, we also try to minimize the number of clusters. The algorithm is enhanced with simmulated annealing. The randomization it brings gives us far better results with the right parameters, as it greatly reduces the risk of getting stuck in a local minimum.

A K-means algorithm has to have a parameter K - the number of clusters. We don't have this number, however, and need to estimate it. The elbow method is a standard method of choosing the right K. It is based on iterating the algorithm for different K parameters and choosing the one with the best ratio of minimizing K as well as the average distortion.

As our data is forming clusters of approximately the same size, we can get away with a simple identification of the elbow in our graph of average distortions, using just the ratio of average distortion of previous K to our current K.

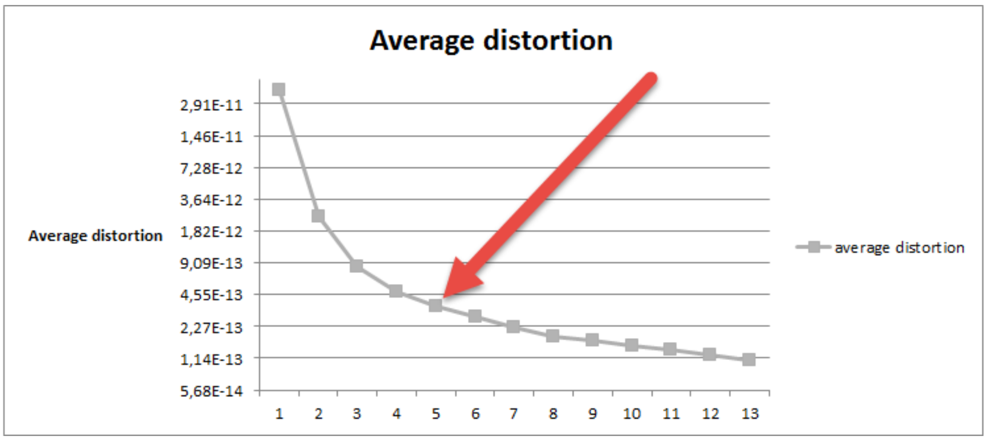


Figure 4.1: Elbow method

#### 4.1.3.3 Parameters

```
[K–meansLocalStrategy]
maxClusters = 10000       ;maximum number of tested clusters per 1
    task
maxTotStageVec0 = 100
maxTotStageVec1 = 10
maxTotStageVec2 = 2
maxTotStageVec3 = 1       ;number of stages = a + (b*k + c*n)^d
minConsecRDL= 0.10        ;min consec RDL
minAccumRDL = 0.10    ;min accum RDL
maxRunStage = 3           ;max run Stages
initProbAccept = 0.5      ;init probability of acceptance
tempRunLengt = 10         ;temp. run length
tempReducFact = 0.95      ;temp. reduction factor
elbowFact = 2         ;elbow Method aceptance factor
```

The number of maxClusters is not that important, as the *task sizes* for the K-means should be much smaller than this number, where the elbow method will just choose the right K and end the computation. If we want to use bigger *task sizes* with thousands of clusters, this algorithm will run much slower, but it will run nonetheless[12]. The $O(n^2)$ complexity for each individual task depends on the actual best K for that cluster according to our elbow method, not on the maxClusters parameter.

Most important is the number of stages (iterations) for the whole clustering algorithms. This represents the total number of randomized starts of the algorithm to get the best result possible. Choose this too small and the quality will drop off, choose this too high and the clustering will be very slow. The complexity for each task is O(n*m), where n is the number of running stages and m the actual complexity of the individual K-means run.

The elbowFact is the coefficient of $d_{K-1}/d_K$, where K is the actual tested number of clusters and d the average distortion, which needs to be satisfied to accept the solution of K as best known so far. Otherwise, the algorithm terminates.

The other parameters are specific for the different kinds of K-means algorithm used and will be discussed in the result chapter 5.

### 4.1.4 Collecting results

When all of the tasks are complete, we need to collect the results from each individual task and merge them into a common result. Thanks to the overlaps, the clusters produced by individual tasks will never be incomplete (those we threw away), but instead, there will be duplicate identifications of the same cluster in spatially neighboring tasks.

Based on the clustering technique used, these clusters will be exactly the same, or will be close to each other. As we can compare cluster centers only based on their floating point coordinates, we cannot rely on exact equality. Instead, we need to define a threshold distance at which we will declare two clusters to be duplicates and need to be merged.

Because we already know the accuracy of our astrometry, we can even improve the results of clustering algorithms inside the individual tasks. For example K-means tends to divide clusters in particularly dense areas into multiple smaller ones. These have their centers very close to each other (typically less then 500 mas).

The merging technique will serve two purposes then:

1. Remove duplicit results when collecting results on the edges of individual tasks.

2. Merge duplicit results of clustering inside the tasks.

[12]The problem will be with the *elbowFactor*, as it is tuned for the *task size* 15. For lower *task sizes* (i.e. bigger amount of observations), it can converge more slowly and for big K the algorithm will run very long

Thus, for each clusterID, we try to find other cluster centers closer than a given threshold (in our case cca 500mas) and from these choose the closest one and merge with it. The coordinates of a resulting cluster are re-computed by weighted coordinates of both of the merged clusters (based on number of members in each cluster). For merging the actual points we need to have a set container for the resulting cluster members, which causes additional, but inevitable memory overhead. More about the memory topic in chapter 4.1.5.

This technique of merging can be described as a simple 1 step streaming K-means algorithm. It will still not ensure that the corner cases such as the results of underlying clustering being in one line will be solved. But as the points entering this phase should be already accurate results of the underlying clustering algorithm, these cases will be extremely rare.

### 4.1.5 Memory optimization

The critical memory consumption point in our program is the distribution of our data into individual tasks. As only the majority (but not all) of our data is spatially localized (sorted), we have to go through the whole dataset to be sure that each task is complete (it contains all points needed to process the task without further communication).

One observation can be represented by two doubles (coordinates) and two Integers (database ID), which is 24B in memory. For our whole dataset, this stands for $4 * 10^8 * 24B = 8GB$ . This is still pretty usable on shared memory based architecture. We can still reduce the memory by using some kind of streaming for this process, but we postponed this optimization for later versions.

Then we start building the pointers to these coordinates which represent processing tasks at first, then clustering groups.

At the end, we need to collect the results. This process is the most memory consuming, as we have to ensure the clusters to be without duplicates. This is ensured by using a set container based on a tree structure - effectively using much more memory per one pointer, than a simple vector.

With simple memory usage, we can see the memory consumption on a small data set of one hundred thousand observations without optimization on Fig. 4.2 and with optimization on Fig. 4.3. These graphs were taken with Valgrind tool Massif [29] and displayed with Massif Visualizer [30].

The most efficient savings come from shrinking the data vector capacity when all coordinates are loaded and from discarding coordinate pointers to tasks that are already processed. Another enhancement that could be made is using integer indices to the data vector, instead of using direct pointers to it's contents (8B for a pointer vs 4B for an array index - we don't have more than integer size observations). This micro-optimization will be implemented if needed, but is not part of our current solution.

At the end of the program run, we can see a fast grow in memory. This is consumed by using a set to store the clustering results for each cluster. This is actually needed to ensure we will not have duplicate entries in one cluster after merging it with another one. At the end of the program, we have only this result map and the original data structures in memory, no more memory can be saved.

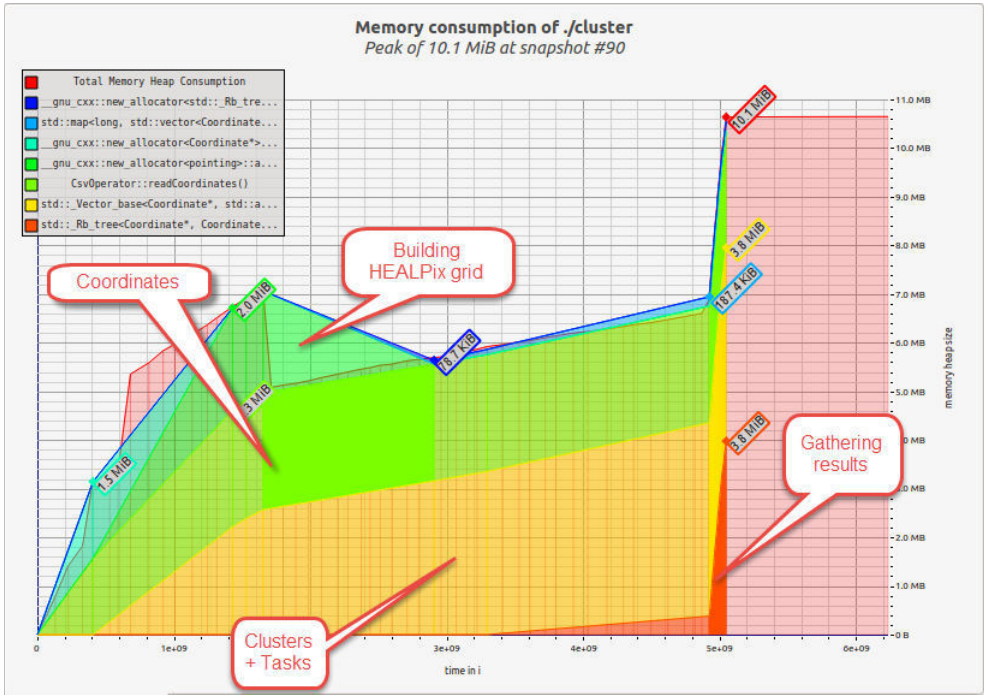


Figure 4.2: Basic memory consumption

## 4.2 Integration

The integration with PostgreSQL data is very simple, based on CSV export of information required for the actual clustering, and CSV import for the results of clustering. The tables used for export and import can be seen in chapter 2.2.

The result of our work can be then published by the mechanism already implemented in my Bachelor's thesis [7]. The resulting light curve can be seen on Fig. 4.4, displayed in SPLAT-VO [31]. The brightness of our star here is around 15 magnitude and the period where the star was observed is from 2456220 Julian Date (19.10.2012) to 2456320 Julian Date (27.1.2013).

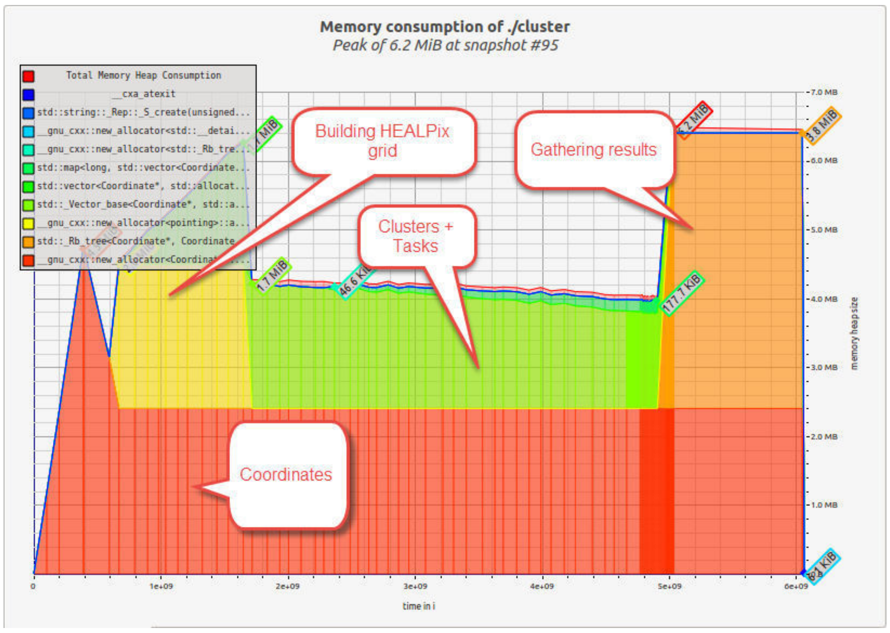
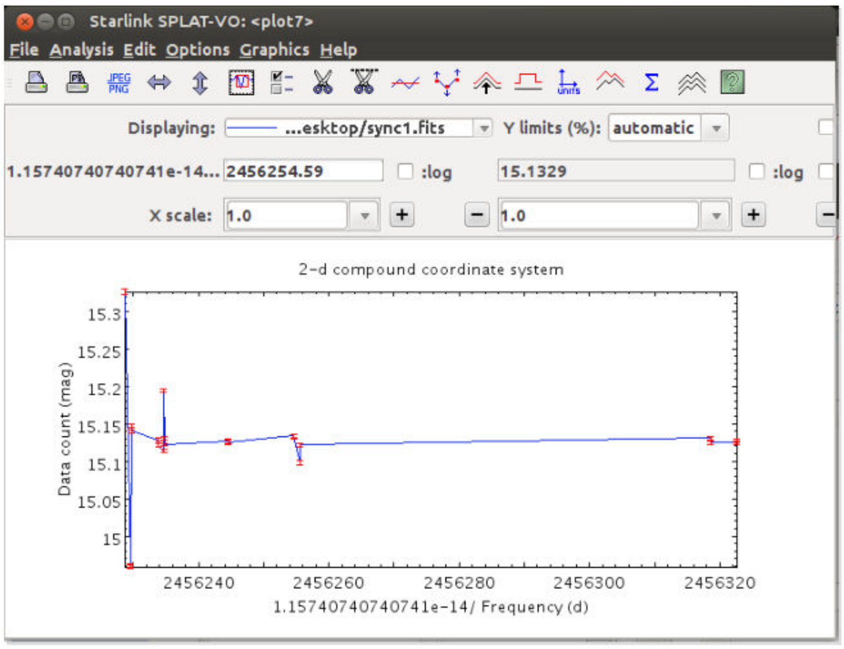


Figure 4.3: Memory savings applied



1.15740740740741e-14... 2456254.59 15.1329

X scale: 1.0 1.0

Figure 4.4: Resulting light curve displayed in SPLAT-VO

Chapter 5

# Results

We will be measuring our results in both performance and quality. For performance testing we are using a machine with 12 cores, 32 GB of RAM. For quality testing, we will be using various comparison of our results with online catalogs as well as analysing the results amongs themselves. We decided to do the comparisons for data on a logarithmic scale, using testing data set composed of 1 million, 10 million and 100 million observations.

The metric of quality will be the fitness of chosen cluster centers (i.e. that the cluster centers were assigned correctly). This can be expressed in various ways and we decided to use the one native to K-means algorithm - the distance of points to their cluster centers (this metric is linearly dependent on distortion used for K-means algorithm).

## 5.1 Time complexity

Here we will discuss the time complexity of the our algorithm. The key parameters on which the complexity depends are:

1. Strategy used
2. Data size
3. Parallel task size
4. Number of threads used
5. Other parameters specific for the strategy used

The strategies measured will be the incremental one described in section 4.1.3.1 and the hybrid version of K-means [26] as it provides the most quality results, as can be seen on Fig. 5.13.

Data sizes used will be as already mentioned 1 million, 10 million and 100 million observations.

The parallel *task size* is explained here 3.2.2.
First we start with the simple incremental clustering strategy.

### 5.1.1 Incremental strategy

The incremental clustering strategy is described in chapter 4.1.3.1. We will analyze the time usage of this simple algorithm for different data sizes.

#### 5.1.1.1 Task size based on data size

On the Fig. 5.1 we can see the total time of our clustering algorithm for 1 million, 10 million and 100 million observations. Here we can see that for the lower *task sizes* (i.e. greater number of observations per one task), are taking more and more time. This behavior is explained on Fig. 5.2.

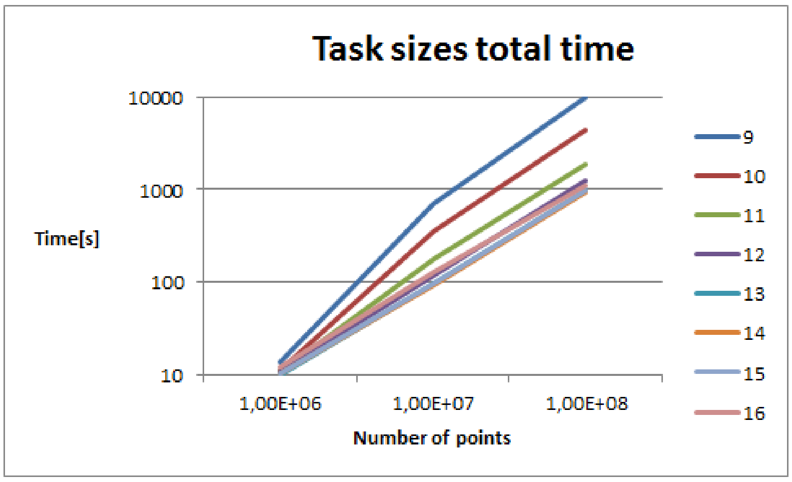


Figure 5.1: Total time based on data size

#### 5.1.1.2 Program phases based on task size

On the Fig. 5.2 we can see running time of parts of the algorithm for 100 million observations based on the *task size*. We can see here, that all of the algorithm parts run the same time, but increasing number of points in one task (decreasing the *task size* parameter) will cause the time of collecting the results. This is because we need to go through more and more results when we are merging the results from individual tasks from the clustering phase.

#### 5.1.1.3 Threads based on data size

On the Fig. 5.3 we can see here the running time for different numbers of threads for 1 million, 10 million and 100 million observations. The algorithm

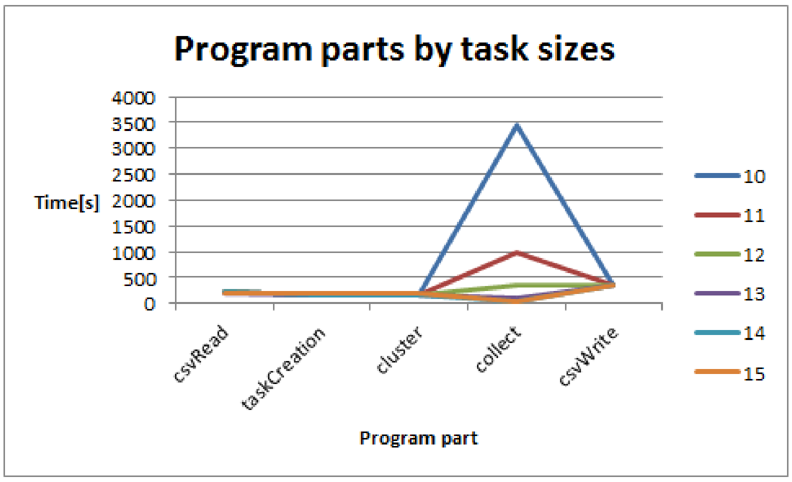


Figure 5.2: Program parts time

is linear and increasing number of threads will not accelerate it, because the clustering phase of the algorithm takes in average cca 80% of the time needed to even read the data from CSV, running on one thread. This relation is compared on Fig. 5.6.

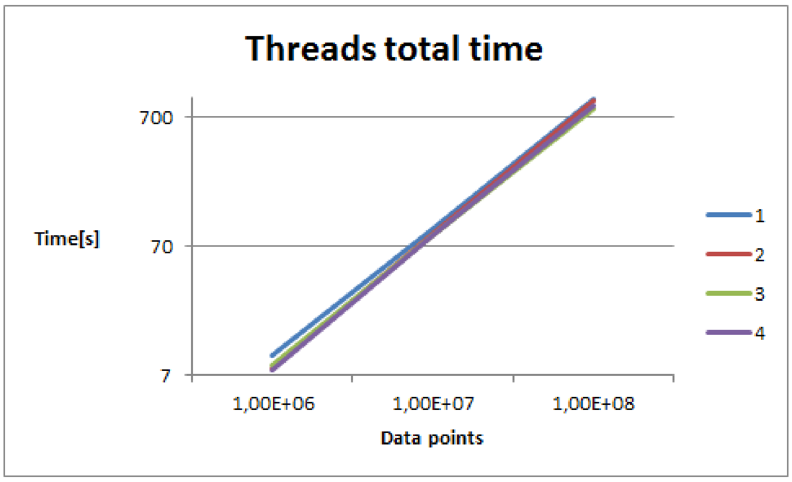


Figure 5.3: Thread time based on data size

#### 5.1.1.4 Threads based on task size

On Fig. 5.4 we can see running time of different numbers of threads for incremental clustering strategy, 100 million observations based on different parallel task sizes. Again we can see here that the number of threads is not accelerating the computation much, parallelized phase of clustering is too simple and short. For smaller *task size* (bigger amount of objects per one task) the algorithm runs longer.

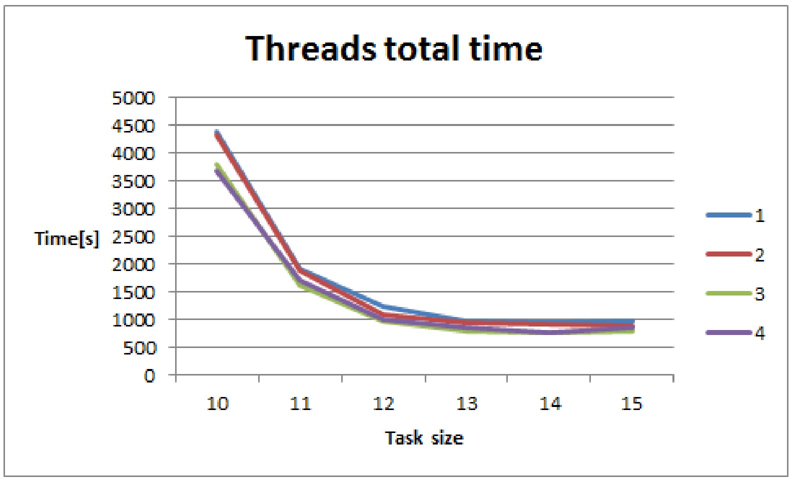


Figure 5.4: Thread time based on task size

#### 5.1.1.5 Parallel efficiency

On Fig. 5.5 we can see the real time for the clustering phase of our algorithm for 100 million rows. We can see that the parallelization is actually efficient but benefit of parallelization is not worth it as usually the clustering phase takes cca 20 % of the computational time (it's complexity is linear with the number of points).

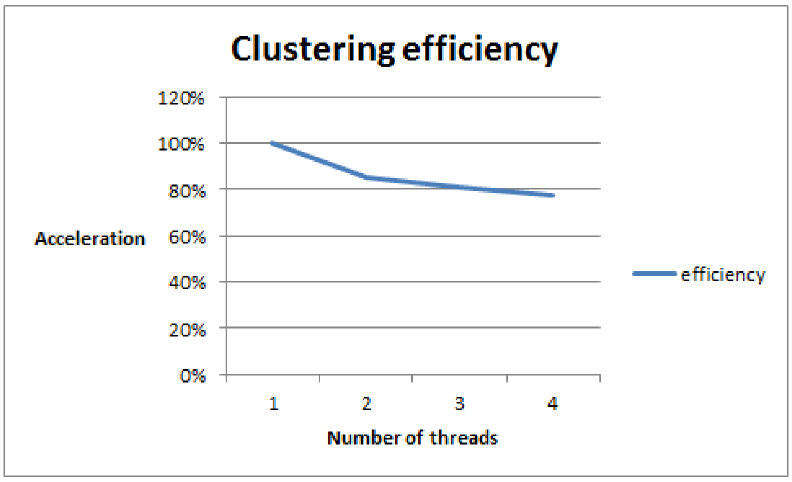


Figure 5.5: Parallel efficiency for clustering time of linear algorithm.

### 5.1.2 K-means hybrid strategy

Because of the higher complexity, we will measure K-means hybrid algorithm with 1 million objects. This number is still considerable when using with this high complexity strategy.

#### 5.1.2.1 Clustering relative time

On the following graphs we can see the comparison of the clustering phase time and the total time. For incremental algorithm, this is a very low ratio, so the results of parallelisation are poor. For a K-means algorithm, however, the percentage is very high and parallelisation works excellent. The comparison can be seen on Fig. 5.6.

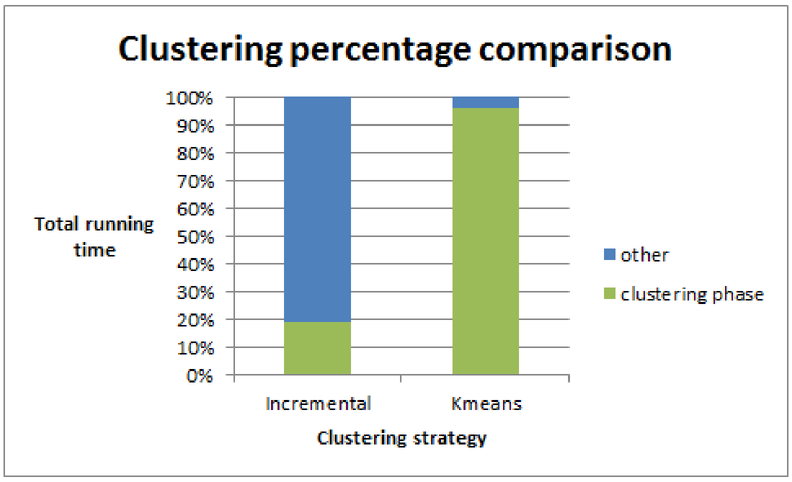


Figure 5.6: Clustering phase time compared to total running time

#### 5.1.2.2 K-means parallel efficiency

For a smaller data set of 1 million observations, it runs a K-means algorithm for times seen on Fig. 5.7. The parallel efficiency for this graph can be seen on Fig. 5.8. We can see that the parallelization is really worth it, as the parallel effectiveness is closing to one even for running time under one minute (i.e. the parallel acceleration is close to linear). The results will be even better for bigger data sets.

## 5.2 Quality

In this chapter we will discuss the results of the clustering algorithms tested. We will discuss the quality of the individual results, as well as their overall error rate and a comparison to on-line catalog.

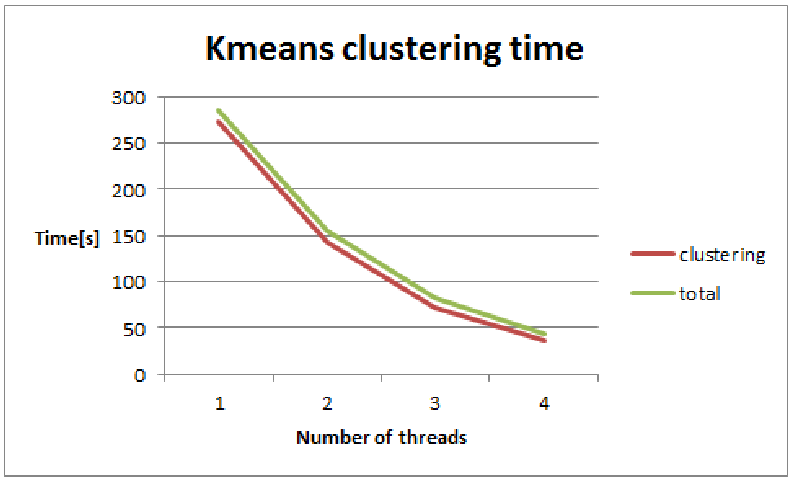


Figure 5.7: Real time of K-means hybrid algorithm

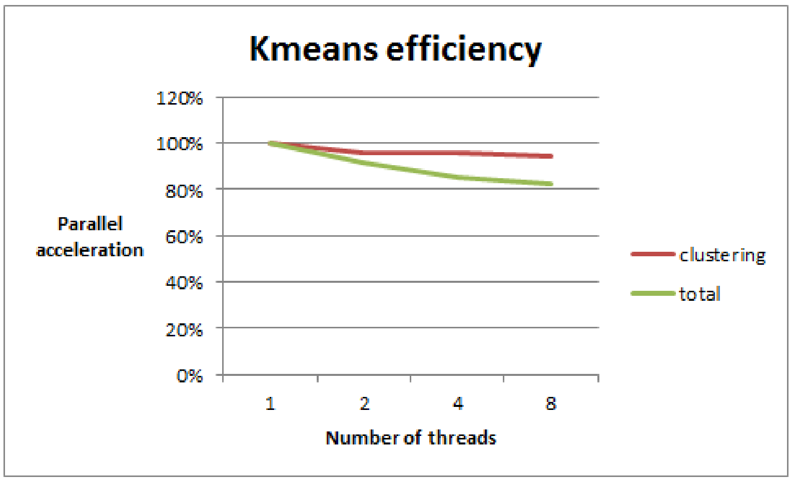


Figure 5.8: Parallel efficiency fo K-means algorithm

For most of the result analysis we used the latest version of Aladin [28] with the new functionality of displaying the HEALPix grid. For the statistical analysis such as histograms, we used TOPCAT [32].

### 5.2.1 HEALPix partitioning

On Fig. 5.9 the big pixel used for an individual parallel task is marked by the red arrow. The individual observations are marked as red circles, the cluster centers in the current task as yellow squares. The smaller quadrilaterals with yellow borders can be used for the overlapping region around our red marked task, as they are clearly bigger than the cluster size.

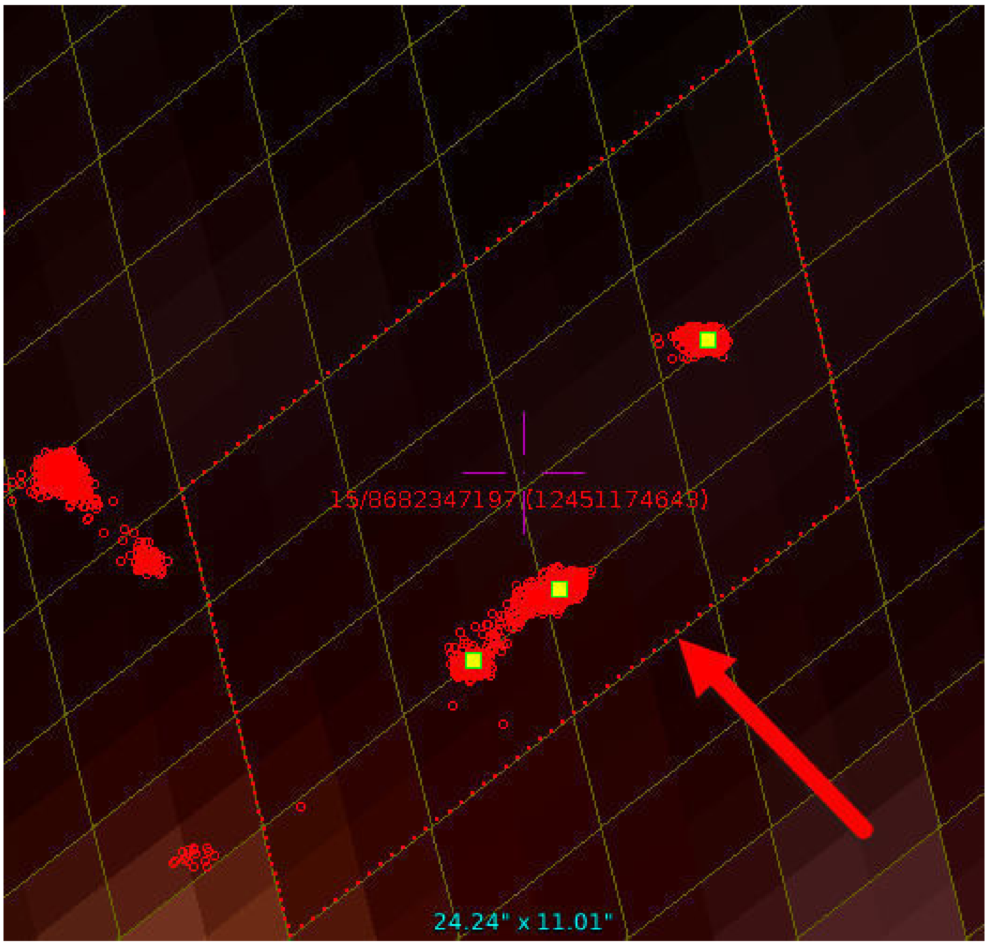


Figure 5.9: Aladin view of the HEALPix grid.

On Fig. 5.10 we can see the big pixel used for one task (borders marked by red dots) and the small quadrilaterals seen are used for overlaps. A line of these small pixels along the red line are forming the overlapping region. For the big pixel situated on the bottom of the image we can see the green marked observations are forming a cluster in it's overlapping region and need to be discarded, as they could interact with the cluster in the big pixel above, without having all of the points needed in the same task.

Another more detailed view can be seen on Fig. 5.10

#### 5.2.1.1 Expectation maximisation

Expectation maximization has poor results with our data and we show it on the following images. On Fig. 5.11 we can see the initial centers. On fig. 5.12 we can see the convergence of these centers during the iterations. No matter

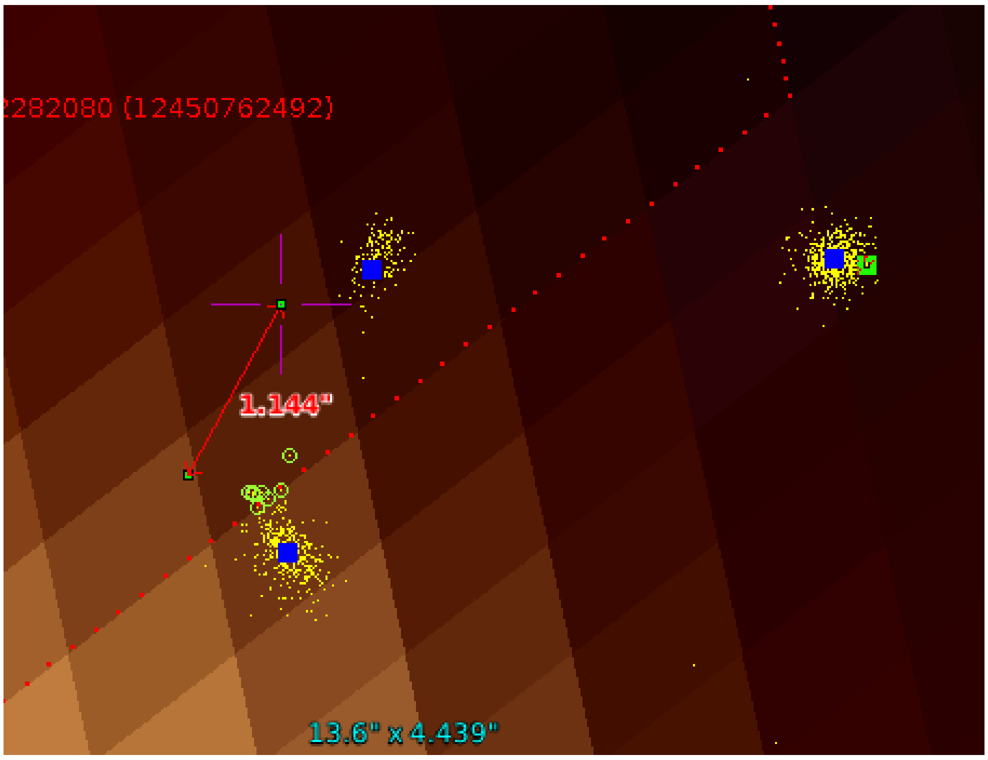


Figure 5.10: Clusters in overlapping region.

what parameters where used, the result always merged all of the data into one big cluster, the differences were only in the speed of convergence to this local minimum.

The reasons can be inappropriacy of the Expectation maximization algorithm for our data, misinterpretation of the arguments, or in the actual implementation we used from a 3rd party library, but we couldn't overcome them and EM algorithm was discarded as inappropriate for clustering our data.

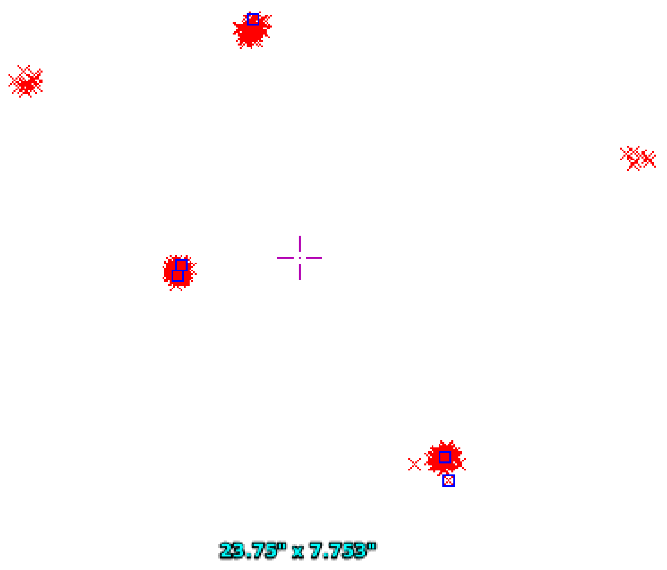


Figure 5.11: Initial centers chosen by random

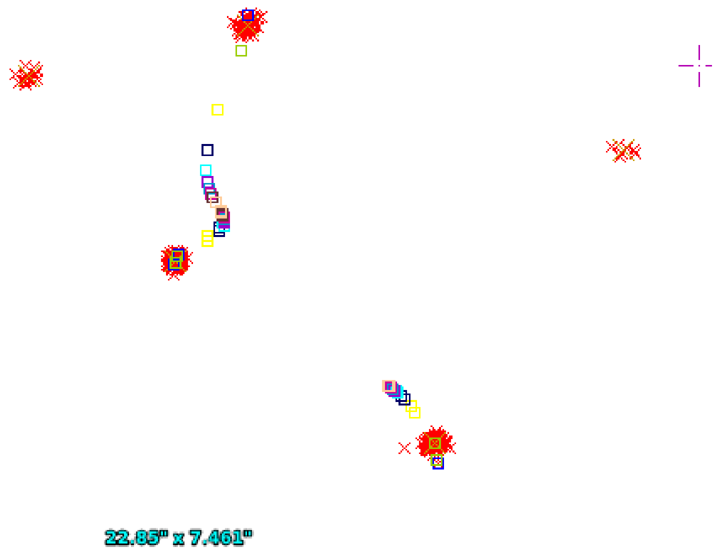


Figure 5.12: EM convergence

### 5.2.2 K-means

Several types of a K-means algorithm are displayed on Fig. 5.13, showing their particular flaws on examples where they misinterpret the clusters. The legend can be seen on the right and the arrows are pointing out the cluster centers this particular version of K-means produced after clustering the data displayed as red dots.

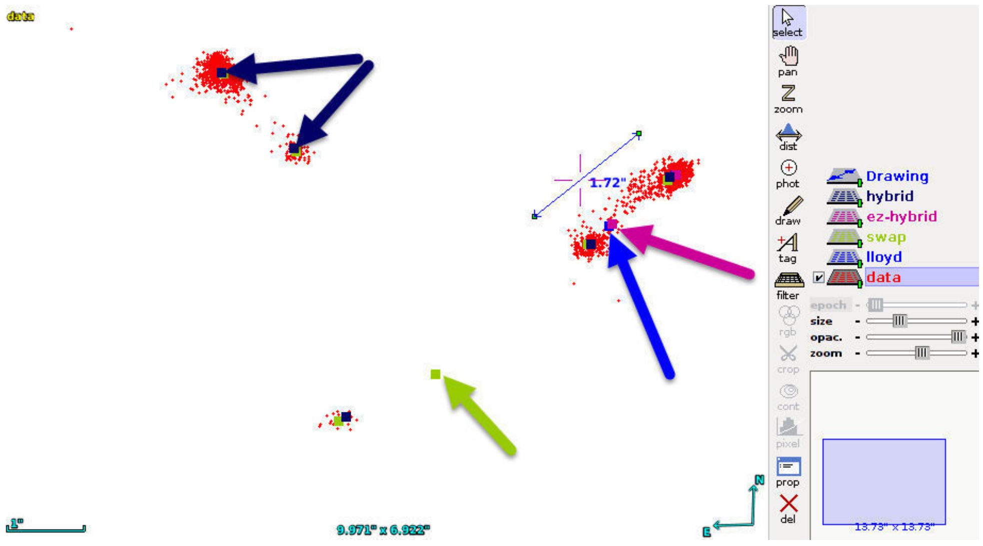


Figure 5.13: K-means variants comparison

#### 5.2.2.1 Elbow factor importance

On Fig. 5.14 we can see what happens if we specify the elbow factor too high. The willingness of the algorithm to accept higher number of centers is low and it ends before it can separate the data correctly.

#### 5.2.2.2 Merging radius too high

On Fig. 5.15 we can see that defining the cluster join radius too high will have similar effect. It joins multiple clusters together even if they were not duplicates.

#### 5.2.2.3 Merging radius too low

On the other hand, on Fig. 5.16 we can see that specifying the duplicate join radius too low (e.g. 100 mas) will cause the duplicate results on edges of our tasks remain. They will not be marked as duplicates and we result with cluster centers closer to each other than 1 arcsec and with observations split among two duplicate clusters.

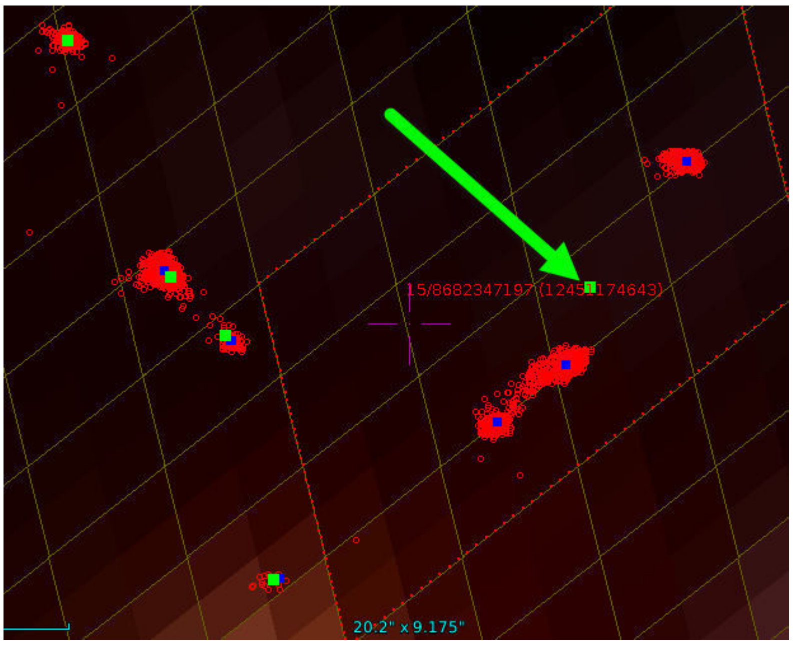


Figure 5.14: Elbow factor too high.

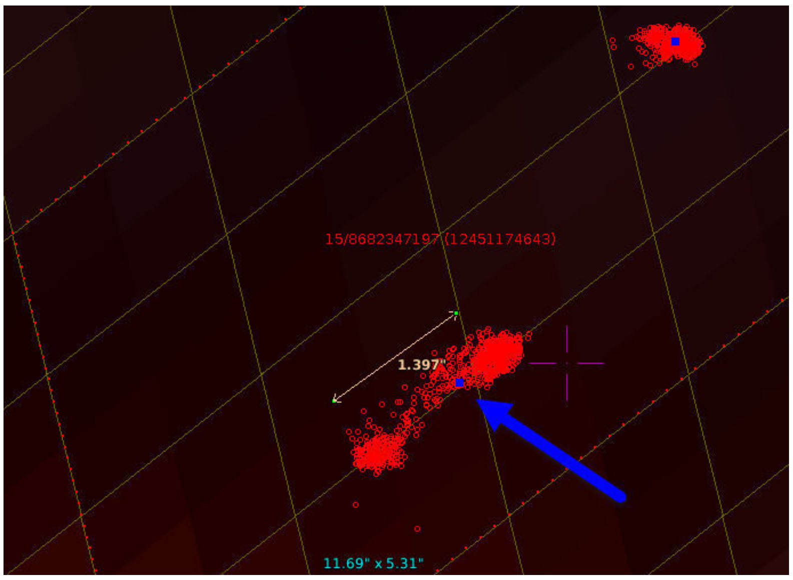


Figure 5.15: Cluster join radius too high.

### 5.2.3 Catalog comparison

Here we will try to cross-match the created catalog with points from which it was created and see the average distance between the cluster center and it's

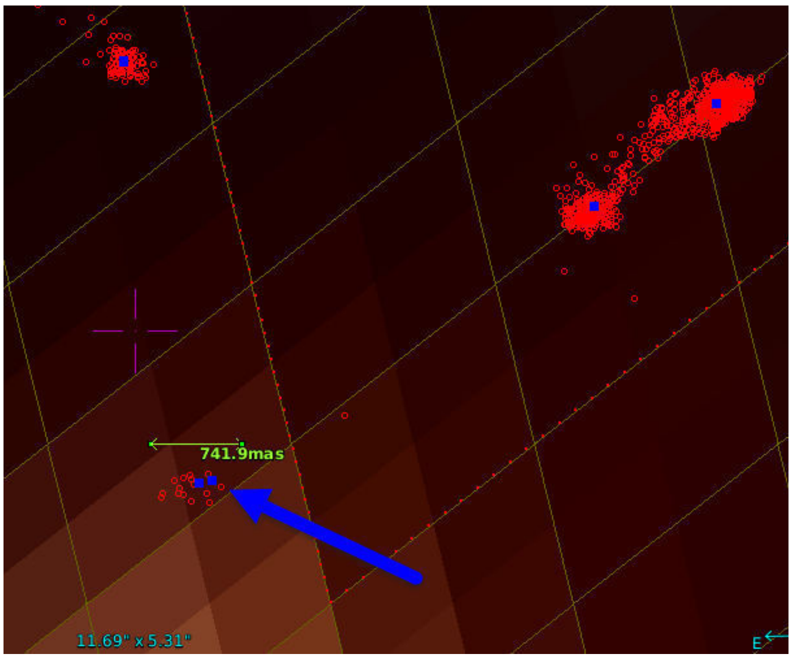


Figure 5.16: Cluster join radius too low.

members.

The graph on Fig. 5.17 has a pattern of chi-squared distribution. This fact comes from the way we are computing distances. For one dimension, the term can be simplified as $(x_1 - x_2)^2$. It is not important, whether $x_1$ is greater than $x_2$, this information is lost with powering the subtraction. So the most points on a histogram won't indeed be around zero, but around mean accuracy of the underlying astrometry process, which we stated to be around 0.25 arcsec.

The catalog identifiers are even more precise than the original astrometry, as they take average coordinates for each cluster, effectively reducing the error. The fact, that we get this histogram with high quality on-line catalog means that we are identifying our cluster centers very precisely, as the mean distance does not shift to higher numbers, but remains below the mean value of the astrometry accuracy.

On Fig. 5.17 we can see the results of our incremental algorithm for one hundred million rows cross-matched to 2MASS on-line catalog. It is a histogram of distances between our catalog identifiers and the ones in the on-line catalog. The red is for using big pixels for parallelization (*task size 10*), the blue is for finer ones (*task size 15*). We can see the results are almost the same - work parallelization is not costing us quality in the case of incremental algorithm.

The results of the k-means algorithm are differing very slightly from our own algorithm and if, they are worse. We can see that on Fig. 5.18 where catalog generation for 1 million observations is compared. The blue histogram is for distances from K-means centers to SDSS catalog objects and the red one for distances between the incremental strategy centers to SDSS catalog objects.

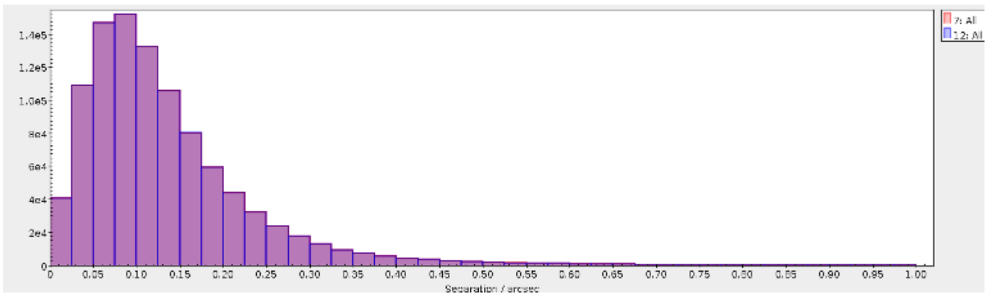


Figure 5.17: Catalog cross-match for incremental strategy

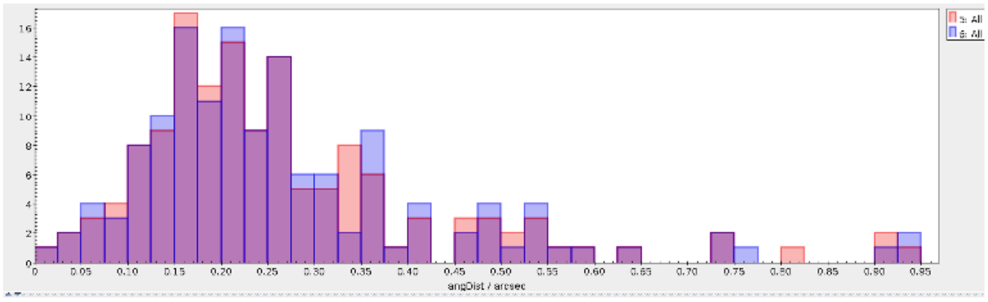


Figure 5.18: Catalog cross-match for incremental and K-means strategy

## 5.3 Result summary

The result quality of different strategies explained above is highly dependent on the actual data that is fed into the program. For the tested data our own incremental strategy provides more accurate results. For more sparse data, however, we believe that the K-means algorithm will produce better results and here the parallel acceleration of the whole solution will kick in.

Nevertheless, Both of the methods provide high quality results and work exactly as expected in the design phase of our project.

# Conclusion

The goal of this thesis has been met. We finally managed to create a fully operational solution, which can in small time create a catalog of light curves of all our observations. The different approach we chose because of the nature of our data made it very hard, but we succeeded. Our approach can be now reused in any other similar sky survey where the light curve information was never mined from it's data.

In the end, the results of our own clustering algorithm with a linear complexity are even more promising than the ones from sofisticated K-means algorithms, which are running in polynomial times. But the processing of the data of our size would not even be possible by these more complex algorithms, if we didn't divide the work so efficiently. That means we would not have a comparison for the strategies and could not state that our linear complexity clustering algorithm provides very useful results, even if compared to these more complex algorithms.

There is another huge benefit of our thesis. We effectively created a pattern that makes clustering or any other processing of spherical data very efficient even for high complexity algorithms as we are able to slice the data space into chunks, that can be processed separately, without degrading the result quality.

# Bibliography

[1] Škoda, P.; Hroch, F.; Nádvorník, J.; et al. Employing the Technology of Virtual Observatory as the Fundamental Framework for the CCD Photometry Survey. In *Astronomical Data Analysis Software and Systems XXIII*, *Astronomical Society of the Pacific Conference Series*, volume 485, edited by N. Manset; P. Forshay, May 2014, p. 305.

[2] Danish 1.54-metre telescope. April 2015, [Cited 2015-05-03]. Available from: `http://www.eso.org/public/teles-instr/lasilla/danish154/`

[3] Ros, E. High-Precision Differential Astrometry. April 2015, [Cited 2015-05-03]. Available from: `http://www.aoc.nrao.edu/events/VLBA10th/oral/11-wednesday/ros-ak.ppt`

[4] Mgr. Filip Hroch, P. Munipack. April 2015, [Cited 2015-05-03]. Available from: `http://munipack.physics.muni.cz`

[5] Zacharias, N.; Finch, C. T.; Girard, T. M.; et al. The Fourth US Naval Observatory CCD Astrograph Catalog (UCAC4). *The Astronomical Journal*, volume 145, no. 2, 2013: p. 44. Available from: `http://stacks.iop.org/1538-3881/145/i=2/a=44`

[6] IVOA Interoperability Workshop – Spring 2015. April 2015, [Cited 2015-05-03]. Available from: `http://www.sexten-cfa.eu/en/conferences/details/54-ivoa-interoperability-workshop--spring-2015.html`

[7] Nádvorník, J. *Ondřejov Southern Sky CCD Photometry Survey: Catalog Server*. Master's thesis, Czech technical university in Prague, 2013, [Cited 2015-05-03]. Available from: `https://dip.felk.cvut.cz/browse/details.php?f=F8&d=K102&y=2013&a=nadvoji1&t=bach`

[8] Doug Tody, J. M. F. B. T. B. I. B. A. M. P. O. J. S. P. S. R. T. F. V. t. D. A. L. w. g., Markus Dolensky. IVOA Recommendation: Simple Spectral

Access Protocol Version 1.1. 2012. Available from: http://arxiv.org/abs/1203.5725

[9] GAVO DC Software Distribution. April 2015, [Cited 2015-05-03]. Available from: http://soft.g-vo.org/

[10] Roeser, S.; Demleitner, M.; Schilbach, E. The PPMXL Catalog of Positions and Proper Motions on the ICRS. Combining USNO-B1.0 and the Two Micron All Sky Survey (2MASS). *The Astronomical Journal*, volume 139, no. 6, 2010: p. 2440. Available from: http://stacks.iop.org/1538-3881/139/i=6/a=2440

[11] Koposov, S.; Bartunov, O. Q3C, Quad Tree Cube – The new Sky-indexing Concept for Huge Astronomical Catalogues and its Realization for Main Astronomical Queries (Cone Search and Xmatch) in Open Source Database PostgreSQL. In *Astronomical Data Analysis Software and Systems XV*, *Astronomical Society of the Pacific Conference Series*, volume 351, edited by C. Gabriel; C. Arviset; D. Ponz; S. Enrique, July 2006, p. 735.

[12] MonetDB. April 2015, [Cited 2015-05-03]. Available from: https://www.monetdb.org/

[13] sciDB. April 2015, [Cited 2015-05-03]. Available from: http://www.paradigm4.com/

[14] International Virtual Observatory Alliance. April 2015, [Cited 2015-05-03]. Available from: http://www.ivoa.net/

[15] LSST Project. April 2015, [Cited 2015-05-03]. Available from: http://www.lsst.org/lsst/

[16] Szalay, A.; Gray, J.; Fekete, G.; et al. Indexing the Sphere with the Hierarchical Triangular Mesh. Technical report MSR-TR-2005-123, Microsoft Research, September 2005. Available from: http://research.microsoft.com/apps/pubs/default.aspx?id=64531

[17] Ivanova, M.; Nes, N.; Goncalves, R.; et al. MonetDB/SQL Meets SkyServer: The Challenges of a Scientific Database. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, SSDBM '07, Washington, DC, USA: IEEE Computer Society, 2007, ISBN 0-7695-2868-6, pp. 13–, doi:10.1109/SSDBM.2007.19. Available from: http://dx.doi.org/10.1109/SSDBM.2007.19

[18] Gray, J.; Szalay, A. S.; Thakar, A. R.; et al. There Goes the Neighborhood: Relational Algebra for Spatial Data Search. *CoRR*, volume cs.DB/0408031, 2004. Available from: http://dblp.uni-trier.de/db/journals/corr/corr0408.html#cs-DB-0408031

[19] MonetDB R integration. April 2015, [Cited 2015-05-03]. Available from: `https://www.youtube.com/watch?v=iPJJmRxxkZ0`

[20] Górski, K. M.; Hivon, E.; Banday, A. J.; et al. HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere. volume 622, Apr. 2005: pp. 759–771, doi:10.1086/427976, `astro-ph/0409513`.

[21] Apache Spark. April 2015, [Cited 2015-05-03]. Available from: `https://spark.apache.org/`

[22] nVidia CUDA Toolkit. April 2015, [Cited 2015-05-03]. Available from: `http://www.nvidia.com/object/cuda_home_new.html`

[23] OpenMP. April 2015, [Cited 2015-05-03]. Available from: `http://openmp.org/wp/`

[24] The R Project for Statistical Computing. April 2015, [Cited 2015-05-03]. Available from: `http://www.r-project.org/`

[25] Eddelbuettel, D.; Francois, R. Rcpp: Seamless R and C++ Integration. *Journal of Statistical Software*, volume 40, no. 8, 4 2011: pp. 1–18, ISSN 1548-7660. Available from: `http://www.jstatsoft.org/v40/i08`

[26] Kanungo, T.; Mount, D. M.; Netanyahu, N. S.; et al. An efficient k-means clustering algorithm: Analysis and implementation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, volume 24, no. 7, 2002: pp. 881–892.

[27] Sanderson, C. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. In *NICTA*, Australia, oct 2010.

[28] Bonnarel, F.; Fernique, P.; Bienaymé, O.; et al. The ALADIN interactive sky atlas. A reference tool for identification of astronomical sources. volume 143, Apr. 2000: pp. 33–40, doi:10.1051/aas:2000331.

[29] Massif: a heap profiler tool for Valgrind. April 2015, [Cited 2015-05-03]. Available from: `http://valgrind.org/docs/manual/ms-manual.html`

[30] Massif Visualizer. April 2015, [Cited 2015-05-03]. Available from: `https://projects.kde.org/projects/extragear/sdk/massif-visualizer`

[31] Starlink SPLAT-VO. April 2015, [Cited 2015-05-03]. Available from: `http://star-www.dur.ac.uk/~pdraper/splat/splat-vo/`

[32] Taylor, M. B. TOPCAT STIL: Starlink Table/VOTable Processing Software. In *Astronomical Data Analysis Software and Systems XIV*, *Astronomical Society of the Pacific Conference Series*, volume 347, edited by P. Shopbell; M. Britton; R. Ebert, Dec. 2005, p. 29.

Appendix A

# Acronyms

**BAT** Binary Association Table

**DaCHS** Data Center Helper Suite

**EM** Expectation maximization

**GAVO** German Astrophysical Virtual Observatory

**GPU** Graphical Processing Unit

**HEALPix** Hierarchical Equal Area isoLatitude Pixelization of a sphere

**HTM** Hierarchical Triangular Mesh

**IVOA** International Virtual Observatory Alliance

**LSST** Large Synoptic Survey Telescope

**MPI** Message Passing Interface

**OSPS** Ondřejov Southern Sky Photometry Survey

**Q3C** Quad Tree Cube

**SMC** Small Magellanic Cloud

**PL/SQL** Procedural Language - Structured Query Language

**PPMXL** Catalog of positions and proper motions on the ICRS

**UCAC4** The Fourth US Naval Observatory CCD Astrograph Catalog

**UDF** User Defined Function

**VO** Virtual Observatory

Appendix B

# Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
src ...................................... the directory of source codes
  src ........................................ implementation sources
  thesis ............... the directory of LaTeX source codes of the thesis
text ........................................... the thesis text directory
  thesis.pdf .......................... the thesis text in PDF format
```

Appendix C

# Source codes

## C.1 buildChunksFromCoordinates

```
int ChunkOperator::buildChunksFromCoordinates(int64 base1Nside, int64 base2Nside) {

    // the algorithm needs a vector of pointings to work, so we need to do some conversion
    vector<pointing> observations;
    for (size_t i = 0; i < obsCoords->size(); i++) {
        observations.push_back(pointing());
        observations[i].theta = (90 - (*obsCoords)[i].dec) * M_PI / 180; // colatitude in radian, some function of observations_degrees[i]
        observations[i].phi = (*obsCoords)[i].ra * M_PI / 180; // longitude in radian, some function of observations_degrees[i]
    }
    size_t noObservations = obsCoords->size();
    HEALPix_Base2 base1(base1Nside, NEST, SET_NSIDE);
    HEALPix_Base2 base2(base2Nside, NEST, SET_NSIDE);

    for (size_t i = 0; i < noObservations; i++) {
        // first see into which job the observation falls
        int64 idx_lores = base1.ang2pix(observations[i]);
        (*obsInCell)[idx_lores].push_back(&(*obsCoords)[i]);
        coords_by_pix_it got = obsInOverlap->find(idx_lores);
        if (got == obsInOverlap->end()){
            (*obsInOverlap)[idx_lores] = vector<Coordinate *> ();
        }

        // now check whether the surroundings of the observation touch neighbouring jobs
        int64 idx_hires = base2.ang2pix(observations[i]);

        fix_arr<int64, 8> neighbors;
        base2.neighbors(idx_hires, neighbors);
        for (size_t j = 0; j < 8; ++j) {
            if (neighbors[j] >= 0) {
                int64 nbidx_lores = base1.ang2pix(base2.pix2ang(neighbors[j]));

                if (nbidx_lores != idx_lores) { // touches a neighbour cell
                    if ((*obsInOverlap)[nbidx_lores].empty() ||
                            (*obsInOverlap)[nbidx_lores].back() != &(*obsCoords)[i]) {
                        (*obsInOverlap)[nbidx_lores].push_back(&(*obsCoords)[i]);
                    }
                }
            }
        }
    }
    return 0;
}
```