Algorithms for Large-Scale Astronomical Problems

Bin Fu

August 2013
CMU-CS-13-122

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This research was sponsored by the Air Force Research Laboratory under grant number FA87500720137 and the Moore Foundation under grant number 2160. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: big data, eScience, astronomy applications, distributed computing, data mining

For people having better lives,

Abstract

Modern astronomical datasets are getting larger and larger, which already include billions of celestial objects and take up terabytes of disk space. Meanwhile, many astronomical applications do not scale well to such large amount of data, which raises the following question: How can we use modern computer science techniques to help astronomers better analyze large datasets?

To answer this question, we applied various computer science techniques to provide fast, scalable solutions to the following astronomical problems:

- We developed **algorithms** to better work with big data. We found out that for some astronomical problems, the information that users require each time only covers a small proportion of the input dataset. Thus we carefully organized data layout on disk to quickly answer user queries, and the developed technique uses only one desktop computer to handle datasets with billions of data entries.
- We made use of **database techniques** to store and retrieve data. We designed table schemas and query processing functions to maximize their performance on large datasets. Some database features like indexing and sorting further reduce the processing time of user queries.
- We processed large data using modern **distributed computing** frameworks. We considered widely-used frameworks in the astronomy world, like Message Passing Interface (MPI), as well as emerging frameworks such as MapReduce. The developed implementations scale well to tens of billions of objects on hundreds of compute cores.
- During our research, we noticed that modern computer **hardware** is helpful to solve some sub-problems we encountered. One example is the use of Solid-State Drives (SSDs), whose random access time is faster than regular hard disk drives. The use of Graphics Processing Units (GPUs) is another example, which, under right circumstances, is able to achieve a higher level of parallelism than ordinary CPU clusters.
- Some astronomical problems are **machine learning** and statistics problems. For example, the problem of identifying quasars from other similar astronomical objects can be formalized as a classification problem. In this thesis, we applied supervised learning techniques to the quasar detection problem. Additionally, in the context of big data, we also evaluated existing active learning algorithms which aim to reduce the total number of human labels.

All the developed techniques are designed to work with datasets that contain billions of astronomical objects. We have tested them extensively on large datasets and report the running times. We believe the interdisciplinarity between computer science and astronomy has great potential, especially toward the big data trend.

Acknowledgements

I express my greatest gratitude to all the people who gave me the possibility to complete this thesis. I could not have achieved this without every piece of your help.

First, I want to thank my advisors, Professor Jaime Carbonell and Dr. Eugene Fink, who gave me the opportunity to conduct computer science research. When I first came to Carnegie Mellon University six years ago, I was a very raw undergraduate student. They guided me in every perspective of life in graduate school, and made me not only a better computer science researcher, but also a better person overall.

Second, thanks to Professor Garth Gibson, with whom I was honored to work. He supported my research, and provided me useful suggestions in various occasions. The most important thing I learnt from him is to think hard and be aggressive, which is exactly what I am lack of. I owe him a big favor.

There are also a couple of Carnegie Mellon faculty members that I learnt a lot from. I published two papers in collaboration with Dr. Julio López, who provided valuable vista and tips from computer system point of view. He also gave me a lot of encouragements along the way. Professor Christos Faloutsos delivered a significant amount of suggestions during our discussions, and he also illustrated me an ideal way to teach an undergraduate course when I was a teaching assistant in his class.

Since my thesis focuses on astronomical problems, I am lucky to work with faculty members and students from Bruce and Astrid McWilliams Center for Cosmology at Carnegie Mellon University, Department of Physics and Astronomy at University of Pittsburgh, and Pittsburgh Supercomputing Center. Professor Michael Wood-Vasey, Professor Rupert Croft, Professor Tiziana Di Matteo, Professor Shirley Ho, and Dr. Joel Welling taught me basic and advanced physics and astronomy knowledge, and gave me extensive feedback on my work. They played an important role for me to understand astronomy background that is essential for this thesis, and I thank them for their consideration and patience for me as a physics novice.

I also consulted both graduate and undergraduate students from Carnegie Mellon University Physics department about many astronomy questions, including Eric Chandler, Colin DeGraf, Tommy Dessup, Nishikanta Khandai, Mariana Vargas Magana, Ross O'Connell, Xiaoying Xu, and Feng Yu. They are very nice and provided a lot of useful information.

During these years I have worked with the following undergraduate students: Tarush Aggarwal, Tanachat (Gor) Nilanon, Yongwen (Peter) Liang, and Sangjae Yoo. They worked hard and contributed a significant part of my research and papers.

I feel that I was always playing catch-up in graduate school, so I express my apologies and thanks to those who needed to take care of my unnecessary troubles. CSD administrators, especially Deborah Cavlovich, SCS Help Desk, payroll office, and etc.

I also want to thank my friends, who shared most of my happiness and sorrows here in Pittsburgh: Bin Fan, Tinglong Dai, Wenjie Fu, Sicun Gao, Fan Guo, Yunchuan Kong, Ni Lao, Lei Li, Nan Li, Jialiu Lin, Yuefeng Lin, Xi Liu, Ning Lu, Min Luo, Kai Ren, Chao Shen, Long Qin, Huimin Tan, Yuandong Tian, Dingxin Qin, Ilari Shafer, Mehrbod Sharifi, Ruizhi Wang, Shuyang Wang, Chuang Wu, Yi Wu, Guangyu Xia, Lin Xiao, Guang Xiang, Guzi Xu, Changqi Yang, Xidong Ye, Junming Yin, Xin Zhang, Xiong Zhang, Le Zhao, Hua Zhong, Yuan Zhou, Haiyi Zhu, and anybody else who I forget to mention here (I have suffered Permanent Head Damage anyway). It is great to have them accompanied in my life these years.

Finally, and most importantly, there are three people I cannot express my gratitude more. My father Deqian Fu and my mother Yimin Kong always supported me and saw me growing up. My fiancée Liu Liu is the nicest person in the world. I dedicate this thesis and my deepest love to them.

Contents

Chapter 1 Introduction

We have entered the "big data" era. Massive information has been created every day at an unprecedented rate, making it an interesting and important issue to store, transmit, analyze, and summarize them.

For example, to index the current web, Google's new indexing system "Caffeine" now keeps track of 30 trillion web pages[1], and the system takes up about 100 petabytes (100 million gigabytes) of storage [2]. The similar trend also arises in environmental research [Reichman et al., 2011], genomics [Ward et al., 2013], biology [Swan, 2013], and etc. Take the medical field as an example: It has been estimated that if we want to record the hourly blood pressure readings for all Americans, it will require 1460 terabytes of storage[3], and more space will be needed to store other types of biomedical data.

What about astronomy? One may think that it is a discipline that does not need to worry about the big data issue. But the advent of modern digital telescopes dramatically increases the size of digital sky surveys: We are now talking about millions or billions of observable astronomical objects. The most recent Sloan Digital Sky Survey (SDSS), for instance, has included over 500 million stars and galaxies [Abazajian *et al.*, 2009], at a speed of 200 gigabytes per night, with the total amount of 140 terabytes in size. For another example, the Guide Star Catalog II contains almost one billion objects [Lasker *et al.*, 2008].

Furthermore, the field is quickly moving toward large time domain surveys, where the surveys are designed to map the whole sky once each few nights. A couple of ongoing projects, including the Panoramic Survey Telescope and Rapid Response System (Pan-STARRS) [PanStarrs] [Kaiser et al., 2002], and the Large Synoptic Survey Telescope (LSST) [Ivezic et al., 2008], will store many time slices of data, and thus become significantly larger than existing surveys. LSST, for example, is estimated to request an initial computing power of 100 teraflops and 15 petabytes of storage.

In the meantime, cosmologists, who conduct simulations about the evolving universe, are also producing much more data than before. Since there are about 170 billion galaxies in the observable universe and to the order of 6×10^{22} (sixty sextillion) stars [Gott *et al.*, 2005], as well as other types of particles such as the hypothesized dark matters, the potential scale of such simulations is enormous. With the increasing power of computing resources, recent publications on state-of-the-art N-body cosmological simulations ([Di Matteo et al., 2008] [Heitmann et al., 2008a] [DeGraf et al., 2012a]) have produced datasets with tens of billions of cosmological particles and occupy tens of terabytes of storage, even after aggressive temporal sub-sampling.

[1] http://www.google.com/insidesearch/howsearchworks/thestory/
[2] http://googleblog.blogspot.com/2010/06/our-new-search-index-caffeine.html
[3] http://onhealthtech.blogspot.com/2011/10/rise-of-big-data.html

Many classical astrophysical[4] applications, however, do not scale up well to such data volumes. For example, to solve the *astronomical correlation function* problem [Peebles, 1980], a straightforward solution takes quadratic time to the number of objects in the dataset, which becomes extremely slow for datasets with more than tens of millions of objects. For a second example, all memory-based techniques are at least cumbersome, if not unable, to process datasets that are larger than the aggregate memory of the computing resources. Actually, when an astronomy dataset is large enough (e.g., 65 billions of objects [DeGraf et al., 2012a]), it is already an interesting topic of how to store and transmit the data themselves.

The use of supercomputers is a natural and popular solution to handle these kinds of large astronomy data. Since large-scale N-body simulations are now conducted on supercomputers with a hundred thousand cores and a few hundred terabytes of memory[5], subsequent analysis can be conducted similarly on the same supercomputer [Chhugani et al., 2012]. However, the access of those supercomputers is not always convenient or free. Astronomers need to e.g. write proposals and share their data to receive time slots on supercomputers, and those time slots are not easily extensible. More flexible tools are thus needed to analyze large data.

So in this thesis, I explore the intersection between modern astronomy and computer science. I want to answer the following question:

> *Assume that we have acquired a large-scale astronomical dataset. How can we use state-of-the-art computer science techniques to help astrophysicists better analyze it?*

The answer depends on the characteristics of the problem we attack. Specifically, we have utilized various computer science techniques to analyze large-scale datasets:

Algorithm. For some astrophysics problems, we do not need to throw in more computing power right away while facing a large dataset. A better algorithm can also do the trick. In Chapter 4 and Chapter 5, we present two pieces of work that both can deal with billions astronomical data entries. For both problems, the information a user requires each time only accesses a small proportion of data out of the whole dataset. As a result, with careful data organization, user queries can be quickly processed by a single desktop machine.

In Chapter 3, we developed an algorithm to work with big data in a different way. When calculating the astronomical correlation functions, we noticed that the computational complexity of existing algorithms is expensive. So instead we explored the possibility of using a sampling-based approximate algorithm to estimate correlation functions. We provided ways to calculate

[4] In this thesis, we use the words "astronomy" and "astrophysics" interchangeably. Strictly speaking, astrophysics is a branch of astronomy, but in this thesis we will not discuss other subfields of astronomy (such as geology).

[5] For example, the Kraken supercomputer with 112,896 cores, 147TB memory, 3.3 PB disk, and 1.17 peak petaflops http://www.nics.tennessee.edu/computing-resources/kraken

the sampling error and showed the empirical running time. To what is better, our solution is also easy to parallelize.

Database. For the problem introduced in Chapter 5, we made use of a relational database to provide a simple and straightforward solution. We carefully designed the table schema, and used indexing and sorting to further reduce query time.

Distributed computing. Not every problem can be simplified computation-wise. Many problems have inevitable high computational complexity, and in order to reduce their processing time, distributed computing techniques have to be applied. In this thesis, we considered two major distributed computing frameworks and compared their differences.

We started with the framework that is already popular to the astronomers, Message Passing Interface (MPI, [MPI, 1993]), a leading message-passing framework in high performance computing community. MPI has been widely used in the astronomy world, and since it aggregates the memory of all the compute nodes together, it greatly increases the power of memory-based algorithms. MPI is specifically suitable for compute-intensive tasks (e.g. N-body simulation), but toward data-intensive tasks, it requires more efforts for programmers to take care of, including data partition, failure handling and so on. It is inconvenient for programmers to handle all these related features themselves.

To overcome those limitations, a new wave of distributed computing frameworks have emerged in recent years, led by Google's MapReduce [Dean and Ghemawat, 2004] and Microsoft's Dryad [Isard et al., 2007] systems. Those frameworks better support distributed computing on large datasets. In this thesis, we mainly used Hadoop[6], the open source implementation of MapReduce, to implement and solve two astronomy problems (Chapter 2 and Chapter 3). We provide a more detailed comparison between Hadoop, MPI, and others in Section 3.4.4.

Hardware. We noticed that two pieces of emerging computer hardware are suitable for sub-problems we encountered in Chapter 3. One example is the use of Solid-State Drives (SSDs), which is becoming an alternative to normal hard disk drives. SSDs have fast random access time (as fast as 0.1ms, comparing to 2.9~12ms of normal hard disk drives), and thus they are efficient to be applied to randomly retrieve a small amount of information from a large dataset.

Furthermore, we made use of General Purpose Graphics Processing Units (GPUs) to accelerate the calculation of all pairwise distances in an astronomical dataset. GPUs are not as general parallel computing resources as CPU clusters, since each GPU core has access to less memory, and it is fast only if the executing program contains little branches. However, it turned out to be suitable for our application. Finally our implementation using SSD and GPU together is competitive comparing to our CPU-based technique.

[6] http://hadoop.apache.org

With the advent of other new hardware (non-volatile memory, shingled disk, etc.), it is worth keeping an eye on their potential use on astronomy problems.

Machine learning. Astrophysics is highly correlated to machine learning and statistics. In fact, there is an individual field, *Astrostatistics*, which solves astrophysics problems using statistical tools. In this thesis, we focus on a specific classification problem: identifying *quasars* from other types of objects. In Chapter 6, we applied supervised learning techniques to solve the problem.

Moreover, with the datasets getting larger, we cannot afford to acquire the labels (galaxies, stars, quasars etc.) of all celestial objects. The quasar classification problem under this scenario can be formulated to an *active learning* problem. In this setting, a user only needs to label a limited number of objects, and an active learning algorithm interacts with the user and decides which objects to label. Normally, an active learner is more efficient than ordinary supervised learners in terms of that it usually requires fewer labels for an active learner to train a classifier with comparable accuracy.

In the following chapters, we give more details on our works to utilize the aforementioned computer science techniques to help astrophysicists better analyze large datasets. From Chapter 2 to Chapter 6, in each chapter we introduce one problem we tackled[7]:

- We developed a distributed solution to the Friends-of-Friends problem [Huchra and Geller, 1982], which is a standard astronomical application for analyzing clusters of galaxies. The distributed procedure can process tens of billions of objects, which makes it sufficiently powerful for modern astronomical datasets and cosmological simulations ([Fu et al., 2010], Chapter 2).
- The computation of correlation functions [Peebles, 1980] is a standard cosmological application for analyzing the distribution of matter in the universe. We studied existing approaches to this problem and proposed a distributed approximation procedure based on a combination of these approaches, which scales to datasets with multi-billion objects ([Fu et al., 2012a], Chapter 3).
- When astronomers analyze telescope images, they match the newly observed objects to an existing catalog. We present a fast matching procedure on a catalog with billions of

[7] I conducted the implementations and experiments for all five pieces of works, except the black hole database (Chapter 5) in which Dr. Julio López completed the data pre-processing works. The algorithms of the distributed Friends-of-Friends technique (Chapter 2) and catalog indexing (Chapter 4) are mostly proposed by Eugene Fink, where I participated in discussions and provided suggestions for improvements. The algorithm of the black hole database (Chapter 5) came from my discussion with Julio López. The algorithms of the correlation function techniques (Chapter 3) and quasar selection (Chapter 6) was developed by me.

objects, which processes millions of newly observed objects per second using only a desktop computer ([Fu et al., 2012b], Chapter 4).

- Cosmologists conduct simulations to study the evolution of black holes by looking at the event history where two or more black holes merge to form a larger one. Existing analytical method no longer scales to the size of black hole datasets produced by the latest cosmological simulations. We introduce algorithms and strategies to store, in a relational database, a forest of black hole merger trees ([López *et al.*, 2011], Chapter 5).
- We report our initial works on the identification of quasars: a type of galaxy that is valuable to astrophysics researchers (Chapter 6). We applied normal supervised learning techniques to classify quasars from other kinds of astronomical objects. Furthermore, since the acquisition of labels is expensive, we also used active learning techniques to reduce the overall labeling cost.

Finally, I summarize in Chapter 7 and envision the related future work. I believe the interdisciplinary of astronomy and computer science has broad opportunity for collaboration and bright prospect.

Chapter 2 DiscFinder: Identification of Galaxy Clusters

DiscFinder [Fu et al., 2010] is a scalable, distributed, and data-intensive group finder for analyzing observation and simulation astrophysics datasets. Group finding is a form of clustering used in astrophysics for identifying large-scale structures such as clusters and superclusters of galaxies. DiscFinder runs on commodity compute clusters and scales to large datasets with billions of particles. It is designed to operate on datasets that are much larger than the aggregate memory available in the computers where it executes. As a proof-of-concept, we have implemented DiscFinder as an application on top of the Hadoop framework. DiscFinder has been used to cluster the largest open-science cosmology simulation datasets containing as many as 14.7 billion particles. We evaluate its performance and scaling properties and describe the performed optimization.

2.1 Introduction

Today, the generation of new knowledge in data-driven sciences, such as astrophysics, seismology and bio-informatics, is enabled by the processing of massive simulation-generated and sensor collected datasets. The advance of many fields of science is increasingly dependent on the analysis of these necessarily much larger datasets. For example, high-resolution cosmological simulations and large sky surveys are essential in astrophysics for answering questions about the nature of dark energy and dark matter (DE&DM) and the formation of large-scale structures in the universe.

The analysis of these datasets becomes extremely challenging as their size grows. They become too large to fit in the aggregate memory of the computers available to process them. Enabling the scaling of science analytics to these larger datasets requires new data-intensive approaches. Frameworks, such as Hadoop and Dryad, are commonly used for data-intensive applications on Internet-scale datasets. These applications include text analysis, natural language processing, indexing the web graph, mining social networks and other machine learning applications. The natural question is then: Can we leverage these data-intensive frameworks for science analytics? In this work, we want to understand the requirements for developing new algorithms to enable science analytics using these systems. In the process, we should understand the advantages and limitations of these frameworks and how they should be enhanced or extended to better support analytics for science.

As part of our collaboration with domain scientists, we have developed DiscFinder: a new data-intensive distributed approach for finding clusters in large particle datasets. Group finding is a technique commonly used in astrophysics for studying the distribution of mass in the universe and its relation to DE&DM. DiscFinder is complementary to other existing group finding methods and is particularly useful for processing very large datasets on commodity compute clusters even in the case where the available aggregate memory cannot hold the entire dataset.

Although, the mechanisms described here are specific to group finders for astrophysics, the principles are generally applicable to clustering problems in other fields of science.

DiscFinder is scalable and flexible. DiscFinder scales up to process datasets with tens of billions of particles, the largest open-science simulation datasets. The DiscFinder design is compact and conceptually simple. The approach used in DiscFinder leverages sequential group finder implementations, which are employed for clustering relatively small subsets of the input particles. The main idea is to group and partition the input particle dataset into regions of space, then execute a sequential group finder for each partition, and finally merge the results together to join the clusters that span across partitions. The sequential finders are independently executed on relatively small subsets of the input to keep their running time low. The approach is implemented as a series of jobs on top of the Hadoop framework. DiscFinder relies on Hadoop for splitting the input data, managing and coordinating the execution of tasks and handling task failures. Finally, the DiscFinder strategy is flexible in the sense that it allows multiple implementations of the sequential group finder to be used. DiscFinder distributes the execution of the sequential finder to scale to large datasets. This strategy reduces the overall implementation complexity and benefits from the efforts invested in the development of the sequential finders.

We are interested in determining the feasibility of the DiscFinder design and understanding its performance characteristics. In our evaluation, we first characterize the DiscFinder scalability with respect to the data size and find that it is possible to cluster large datasets with this approach.

The main benefits of DiscFinder and future analysis applications implemented using similar approaches are their simplicity and potentially shorter development time. However, the simplicity of the implementation comes at a performance cost. We characterize the DiscFinder running time and apply a set of modifications to the implementation to improve its performance. There is clearly room for improvement both at the application and framework levels. We discuss various potential optimizations that can provide additional performance benefits. As the performance of these frameworks improves, they will be more widely used in data analytics for science.

The rest of this chapter is structured as follows: We describe the motivating application in Section 2.2, and summarizes previous related work in Section 2.3; the DiscFinder design is explained in Section 2.4; the implementation details are presented in Section 2.5; We presents the performance evaluation of the approach in Section 2.6.

2.2 Motivating Application

Analysis of Astronomical Datasets. Domain scientists are now commonly faced with the challenge of analyzing massive amounts of data to conduct their research. In particular, ever increasing large observation and simulations datasets abound in astrophysics. The advent of digital sky surveys, beginning with the Sloan Digital Sky Survey (SDSS), increased dramatically the scope and size of astronomical observational datasets [Abazajian *et al*., 2009]. The latest SDSS data release was over 30 TB in size. The field is moving toward large time domain astronomy surveys, such as Pan-STARRS [PanStarrs] [Kaiser et al., 2002] and LSST [Ivezic et al., 2008], which will store many time slices of data. Pan-STARRS is producing in the order of 1TB/day of imagery. Both Pan-STARRS and LSST are projected to produce multi-TB datasets over the lifetime of the surveys. Similarly, state-of-the-art N-body cosmological simulation, such as the Bigben BHCosmo and Kraken DM simulations [Di Matteo et al., 2008] produce multi-billion particle datasets with sizes in the orders of tens of terabytes, even after aggressive temporal sub-sampling. This down-sampling is needed to deal with the bandwidth and capacity limits of the storage system available in the supercomputers where these simulations execute.

The analysis of these datasets is essential for tackling key problems in astrophysics, such as the understanding the nature of dark energy (DE) and dark matter (DM), and how DE&DM controls the formation of large-scale structures in the universe [Colberg and Di Matteo, 2008], such as clusters and superclusters of galaxies. To better understand the role of DE&DM in the evolution of the universe, theoretical and observational astrophysicists analyze the aggregate properties of large-scale structures, such as the distribution of their size, volume and mass [White, 2002]. Finding the groups of particles that make up the large-scale structures is the first step toward carrying out this process. Once the groups have been identified, then their properties can be calculated and the groups can be decomposed into sub-halos for further analysis [Springel et al., 2008] [Couchman *et al*., 1996] [Dikaiakos and Stadel, 1996] [Springel, 2005] [Heitmann *et al*., 2008b].

Group Finding. In astrophysics, group finding refers to a family of physics-based spatial clustering problems applied both to observation and simulation datasets. Their input is a set of celestial objects such as stars, galaxies, gas, dark matter, etc. We will refer to those as particles, points or objects. A group finder separates the particles into groups that make up larger structures such as galaxies and clusters of galaxies. In very loose terms, we will refer to those structures as groups or clusters interchangeably. In slightly more formal terms, a group finder takes an input set of particles $P = \{p_1, p_2,\ldots, p_n\}$ and produces a set of m disjoint groups $G = \{G_1,\ldots, G_m\}$ such that each group G_i comprises the subset of points from P (i.e., $\forall G_i \in G$: $G_i \subset P$ and $\forall G_i \in G$, $\forall G_j \in G$, $i \neq j$: $G_i \cap G_j = \emptyset$). The criteria for determining the membership of a particle to a group may use physics-based computations that take into account particle properties such as position, mass and velocity. There are many variants of group finding approaches, both from the algorithmic point of view as well as the criteria used for the selection.

Friends-of-Friends (FOF). FOF is a simple algorithm proposed by Huchra and Geller to study the properties of groups of galaxies in observation surveys [Huchra and Geller, 1982] [Davis et al., 1985]. Its group membership criteria are solely based on the Euclidean inter-particle distance. FOF is widely used and works in practice for many analysis scenarios. There are a large number of more sophisticated group finding approaches that are based on FOF.

FOF is driven by two parameters: a *linking length* (τ) and a *minimum group size* (*minGz*). The input is a set of particles P, in particular each particle is a tuple of the form $<pid_i, (x_i, y_i, z_i)>$, where pid_i is the particle identifier and x_i, y_i, z_i are the particle's position coordinates in 3D space. The only group membership criteria is the following: two particles p_i and p_j belong to the same group G_l (i.e., they are *friends*), if the distance between them (d_{ij}) is less than τ. This procedure is illustrated in two dimensions in Figure 2.1. Any other particle p_k also belongs to the group G_l, if the distance between p_k and any of the particles in G_l is less than τ. After we apply this criterion, all the friends of p_i become friends of p_j and vice versa, which reflects the name of the approach. The output set G comprises the groups that contain a number of points equal to or larger than the *minGz* parameter. The output is represented as a list of tuples of the form <*pid*, *groupId*> where *groupId* is the group identifier.

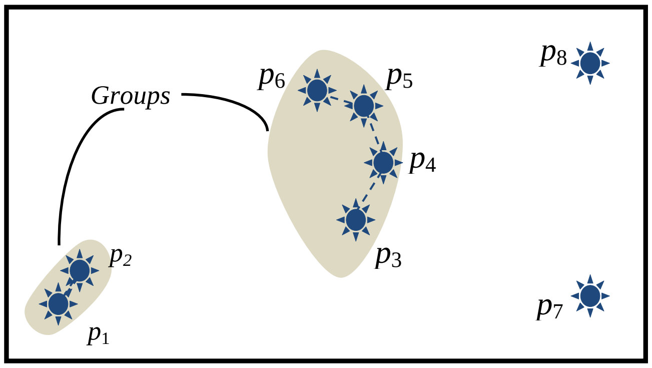


Figure 2.1: Friends of Friends (FOF) clustering. This dataset contains 8 particles $p_1,\ldots,p_8$. Shaded regions denote groups of particles. Notice that particles p_3 and p_6 belong to the same group although they are not close enough to each other. The reason is that they are indirectly linked through their friends, particles p_4 and p_5. In this example, *minGz* = 2, and thus particles p_7 and p_8 do not belong to any group.

There are available sequential implementations of the FOF algorithm, such as the one from the "N-body shop" at the University of Washington [UW-FOF]. These implementations rely on building a spatial indexing structure, such as a *kd*-tree, to speed up lookups to nearby particles [Lee and Wong, 1977]. However, sequential solutions are relatively slow to process billions of galaxies. There are also various parallel group finders for distributed memory machines with low-latency networks that are often used in simulations (See Section 2.3).

Existing group finding implementations, whether sequential or parallel, are in-core and thus require large enough available memory to fit the dataset and internal data structures. This means that finding groups and analyzing large simulation datasets require supercomputers of comparable capacity as the one used to generate them. For the very large datasets these resources are not readily available. In the cases where there is enough aggregate memory to execute the group finding process, the end-to-end running time is dominated by the I/O required to load the input dataset and produce the results. We have developed an alternative data-intensive group finding approach named DiscFinder, which complements existing approaches. DiscFinder is useful for finding groups in datasets need to be loaded from external storage, such as it is often the case in the analysis of astronomy surveys and post-simulation analysis of synthetic datasets. DiscFinder makes it possible to find groups in datasets that are much larger than the memory of the available compute resources.

2.3 Related Work

Group finders used in astrophysics refers to a class of clustering approaches. The group membership criteria used in the basic FOF approach may not be appropriate for certain applications. Variations of the base algorithm are designed to handle uncertainties in the input data, and to take into account other particle properties besides their positions [Gottloeber, 1997] [UW-Skid]. For example, Extended FOF (EXT-FOF) is a finder used in photometric datasets [Botzler *et al*., 2004]; probability FOF (pFOF) is used to identify galaxy groups in catalogs in which the red shift errors have large dispersions [Liu *et al*., 2008]. Hierarchical FOF [Gottloeber, 1997], SOF[8], SKID [UW-Skid], DENMAX [Gelb and Bertschinger, 1994], IsoDEN [Pfitzner *et al*., 1997], HOP [Eisenstein and Hut, 1998] and Sub-halos [Springel et al., 2008], among others, are sophisticated group finders with selection criteria that take into account the density of the neighborhood surrounding a particle and whether or not that particle is gravitationally bound to the larger structure. The approximate implementation from the University of Washington "N-body shop" (aFOF [UW-aFOF]) aims to overcome the severe slowdown experienced by the basic FOF implementation when the linking parameter τ is relatively large.

Parallel group finders, such as pHOP [Liu *et al*., 2003], HaloWorld [Pfitzner and Salmon, 1996], Amiga Halo Finder (AHF) [Gill *et al*., 2004] and Ntropy [Gardner et al., 2006], are implemented atop the Message Passing Interface (MPI) [MPI, 1993] and are designed to execute on parallel distributed memory machines with a fast low-latency interconnect between the processing nodes. These implementations are suitable for finding groups during the execution of the simulation. Ntropy is implemented as a framework atop MPI that provides a distributed *kd*-tree abstraction. One of its goals is to make it easy to implement analysis algorithms, such as FOF, on massively parallel distributed memory platforms. The user writes a function for reading the data that the Ntropy framework calls at run time to load the data into memory.

All the aforementioned approaches require the complete dataset to fit in memory to execute. Their reported performance excludes the time needed to load the data and assume that the data is already in memory, which makes sense for the case where these approaches are used in numerical simulations. Recently, Kwon *et al*. developed an approach atop Dryad

[8] http://www-hpcc.astro.washington.edu/tools/so.html

[Kwon et al., 2009] [Kwon *et al*., 2010] that shares similarities with the work presented here. They have shown results for clustering datasets with up to 1 billion particles using 8 nodes.

2.4 DiscFinder

DiscFinder is a distributed approach for group finding in datasets that may be larger than the total aggregate memory of computers performing the computation. DiscFinder enables the use of stand-alone sequential group finders in a distributed setting, and thus leveraging the efforts that has been put in the development of those sequential implementations. At a high level, the basic idea behind DiscFinder is to take a large unordered input particle set, then organize it and split it into multiple spatial regions and find groups in each region using the sequential group finder. The resulting groups from each region are merged to obtain the global set of groups. Figure 2.2 depicts the stages involved in the DiscFinder pipeline. As in many other computational sciences applications that deal with a spatial phenomenon, DiscFinder uses spatial partitioning to split and distribute the computation across processing units. Unlike other applications, there is no explicit communication in the algorithms used at the application level. Instead, the needed data is moved by an underlying framework or through a distributed file system.

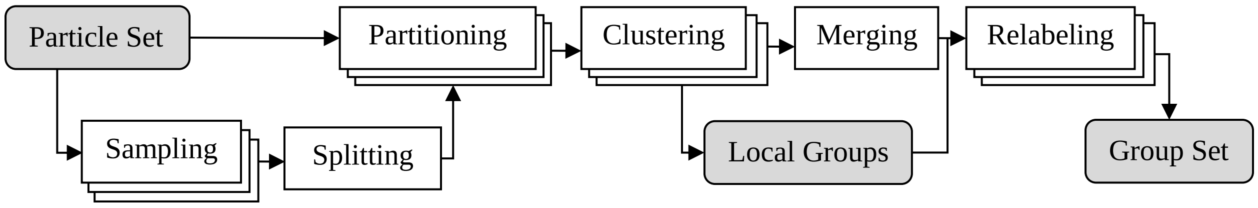


Figure 2.2: DiscFinder Pipeline. The DiscFinder computation flow graph comprises the following stages: 1. Sampling (distributed), 2. Splitting (sequential), 3. Partitioning (distributed), 4. Clustering (distributed), 5. Merging (sequential), and 6 Relabeling (distributed).

Particle Set. The input particle dataset contains tuples of the form <*parId*, *pos*, *oAttr*>, where *parId* is the particle identifier, *pos* is the position of the particle in 3D space, and *oAttr* are other particle attributes such as mass, temperature, velocity, and so on. To cluster particles using the FOF criteria, we are only interested in *parId* and *pos*. Other attributes can be safely ignored and not read at all.

Sampling. The objective of this stage is to sample the dataset to build a suitable partition of the data so the processing can be evenly split into independent compute nodes. Since the particles positions in space is the primary criteria for determining the groups, the generated partitions correspond to regions of space. The simplest approach is to divide the space into equal-size disjoint regions, such as 3D cubes. However, we have a-priori knowledge that points in real-world astronomy datasets, whether generated by simulation or collected by observation, rarely follow a uniform distribution. This makes it undesirable to simply split the space into equal-size cubes, as this partitioning scheme generates partitions with different number of points per partition, and thus becomes hard to balance the load. A tree-based data structure, such as a *kd*-tree, can be used to split the domain space into cuboid regions with equal number of points. This process is memory intensive and the time complexity of building such *kd*-tree using a median-finding algorithm is O(*n* log *n*), where *n* is the number of particles. The resulting cuboid regions are axis-aligned rectangular hexahedra, which we simply refer to as *boxes*.

To deal with the size of the input dataset, instead of building a *kd*-tree for the whole dataset, we build a much smaller *kd*-tree with randomly selected sample points. The sampling stage is performed in a fully data-parallel manner. The input dataset is divided into many disjoint pieces such as file offset ranges or subsets of files for datasets made up of many files. Then a worker task can sample each of these subsets by choosing a small percentage of the records, e.g., 0.01%, and output the position of the sampled records. There is a sample set per worker task.

Splitting. This is a sequential step that creates the split boxes used for partitioning the particles. It takes the sample sets produced by the sampling tasks in the previous step and builds a *kd*-tree that can be used to partition the input particle set into boxes with roughly the same number of particles per box. The granularity of the boxes, and thus the depth of the tree, is chosen so that in the cluster stage, for each box there is enough memory left to execute the sequential group finder code on the particles in the box. This yields many partitions that are usually larger than the number of available worker tasks for the pipeline. The number of worker tasks is proportional to the number of available compute nodes, e.g., 4 to 8 worker tasks per compute node. If the number of partitions is less than the number of worker tasks that means the pipeline can be easily executed with fewer computers.

Shell-based Partitioning. This stage takes the input particle set and sorts it according to the partitions produced by the splitting phase. Splitting the data in disjoint spatial partitions requires communication across neighboring partitions to determine whether a group crosses a partition boundary. For example, the domain shown in Figure 2.3 is divided into 4 rectangular partitions. Although particles 3 and 4 are in separate partitions, they are close enough to belong to the same group. DiscFinder is designed so each partition can be processed independently and asynchronously. Partitions can be processed at different points in time. DiscFinder is targeted to execute atop data processing frameworks, such as MapReduce, Hadoop, or Dryad, that do not provide explicit communication primitives for the applications.

DiscFinder uses a shell-based partitioning scheme to avoid explicit communication at the cost of a small increase in the memory requirement for each partition. All the partitions are extended by a small factor $sl = \tau/2$ on each side, where τ is the linking length parameter for the group finder. The end result is that a partition has a shell around it that contains points shared with adjacent partitions. Shell-based partitioning enables the independent and asynchronous processing of each partition by decoupling the local computation of groups inside a partition from the resolution of groups that span across partitions.

To illustrate the approach, consider a 2D domain made of 4 partitions as shown in Figure 2.3. The non-overlapping partitions are delimited by the heavy continuous line. Each partition is extended along each boundary by an additional length of $\tau/2$ (shaded area). The particles numbered from 1 to 7 in the figure form a group. Without the shell, two disjoint groups would be created. The group in the top-left partition would contain three of the points (1, 2, 3) and the group in the top right partition would comprise the other four points in the group (4, 5, 6, 7). Joining these two non-overlapping groups requires communication across the tasks that process these partitions. Once the partitions are extended with the shell, the top-left partition finds a group G_1 = 1,…,5; The top-right partition contains a group with 5 points G_2 = 3,…,7. The members of these two groups overlap, which facilitates the unification of the groups in a separate merging stage down the pipeline.

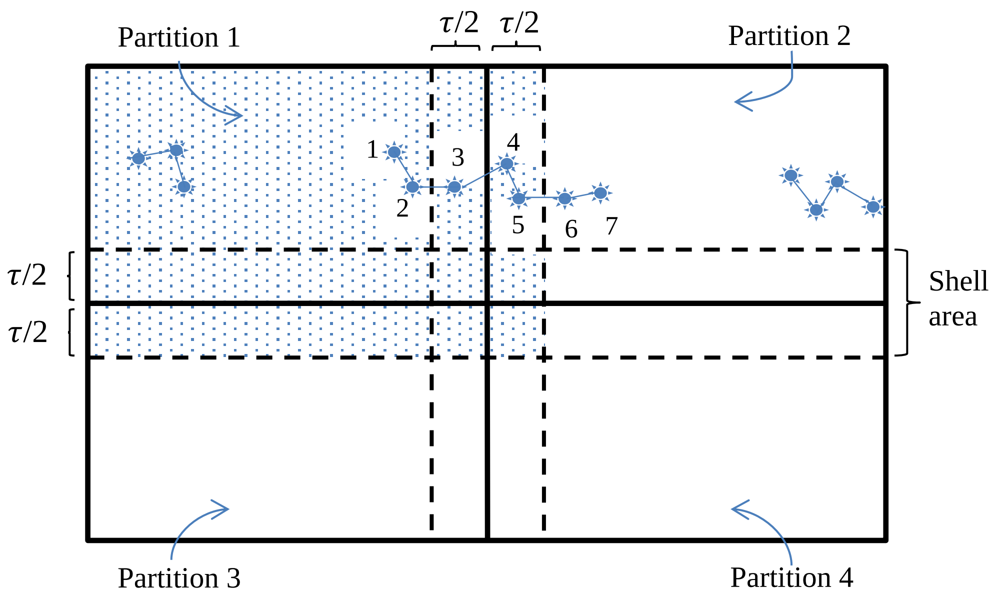


Figure 2.3: Shell-based Partitioning. Four partitions are shown here in two dimensions. Each partition has an additional shell of length $\tau/2$ along the boundary. The shaded area at the top left of the figure represents the actual size of the partition 1 once it has been extended to include its shell.

The choice of the value for *sl* ensures that when two points in separate partitions are at a distance $d(p_i, p_j) < \tau$, then at least one of them is included in the shell of one of the partitions, which is enough to later find the groups that span multiple partitions. Note that shell-based partitioning increases the volume of each partition and potentially the number of particles to be processed per partition. This approach works under the assumption that the size of the shell, or equivalently the chosen linking length parameter τ, is relatively smaller than the size of the partition, which usually holds in practice.

The partitioning stage is distributed across many worker tasks. Each task reads a chunk of the input particles and classifies them into buckets that correspond to the spatial partitions (including their shells). Each bucket contains a subset of the particles belonging to a spatial partition.

Distributed Clustering. In this stage, all the particles in each bucket (i.e., partition) are collected so a worker task has all the particles in a shell-extended partition. The partitions are distributed to worker tasks. Locally, each task executes a sequential group finder, such as the FOF [UW-FOF] or aFOF implementations [UW-aFOF] from University of Washington. For each point in a partition, the local group finder generates a <*pointId*, *pGroupId*> tuple, where the group identifier *pGroupId* is local to each partition. The generated tuples are separated into two sets: *shell set* and *interior set*. The shell set contains points inside the shell region. This set is passed to the merging stage. The interior set contains points that fall inside the original disjoint partition, but not in the extended shell of any partition. This set is stored and used again later in the relabeling stage. At the end of the distributed clustering stage there are R shell sets and R interior sets, where R is the number of partitions generated by the splitting phase.

Merging. As shown in Figure 2.3, a point inside a partition's shell is processed in two or more partitions. Such a point may belong to different local groups across partitions. Figure 2.4 shows the resulting groups for two partitions (1 & 2) of Figure 2.3. Points 3, 4, 5 are assigned to group G_1 in partition 1 and to group G_2 in partition 2. The purpose of the merging stage is to consolidate groups that span multiple partitions by using R shell sets generated in the previous stage. The amount of data passed to this stage is relatively smaller than the size of the particle datasets (since τ is relatively small), thus reducing both compute and I/O requirements for this stage.

The merging stage employs a Union-Find algorithm to merge subgroups that have common points into unique global groups [Galler and Fischer, 1964] [Galil and Italiano, 1991]. Union-Find uses a *disjoint-set* data structure to maintain several non-overlapping sets, while each set containing one or several elements. This data structure supports the following two operations: *Union* and *Find*. Union(A, B) merges two sets A and B and replaces them with their union. Find(e) determines to which set an element e belongs. For the purpose of the group merging stage, a group corresponds to a set in the disjoint-set data structure, and a particle corresponds to an element. Initially, each particle belongs to its own group. The procedure iterates over all the particles and when it discovers that a particle belongs to two different groups, and then it merges those two groups using the Union operation. The output of the merging procedure is a list of *relabeling rules* that describe the equivalent groups, i.e., the groups that span across partitions. These rules are tuples of the form <*oldGroupId*, *newGroupId*>. A set of relabeling rules is produced for each partition.

This data structure and the corresponding Union-Find algorithm can be implemented using hash tables. The complexity of the Union-Find algorithm is nearly linear in the number of input elements [Tarjan, 1975]. Since it is only applied to a small subset of the particles, the ones inside the shells, its running time is relatively short compared to the rest of the pipeline. Since this algorithm is sequential, its scalability is limited by the memory available in a single processing node. We have used it to execute pipelines with close to 15 billion particles. Until now, its memory requirements have not been an issue.

Relabeling. This is the last stage in the pipeline. Its purpose is to apply the relabeling rules from the merging stage to each of the partitions generated in the clustering stage. This is a data-parallel process. The work is distributed to many tasks, where each task applies the rules to a partition. This is done in a single pass. The partitions are expected to be evenly balanced, so the running time of this step is proportional to N/m, where N is the total number of particles and m is the number of available processing units.

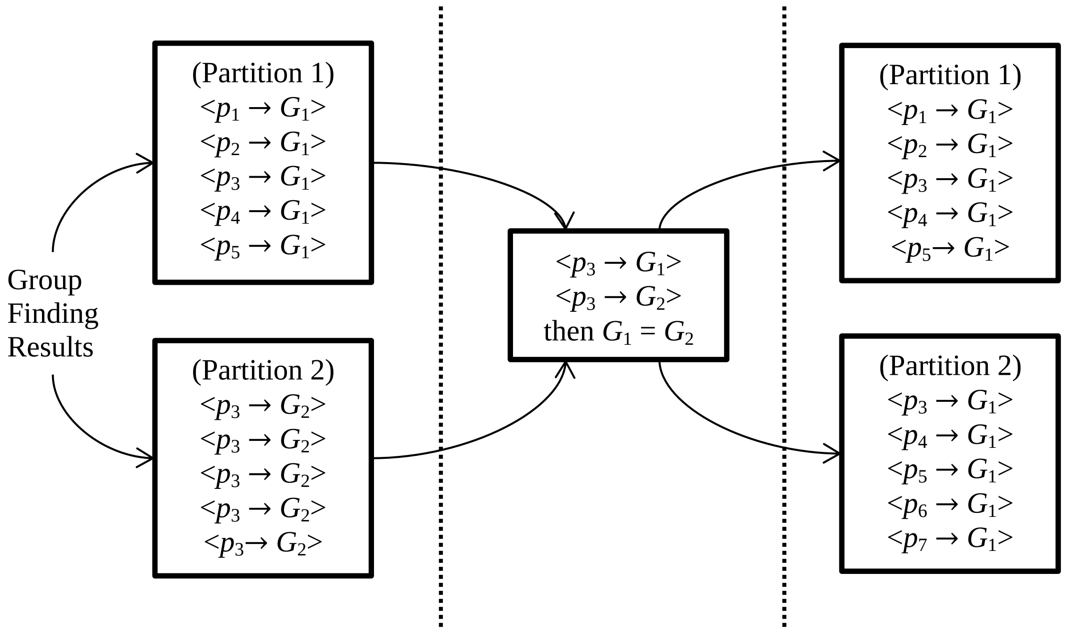

Group Finding Results
(Partition 1)
<$p_1 \rightarrow G_1$> <$p_2 \rightarrow G_1$> <$p_3 \rightarrow G_1$> <$p_4 \rightarrow G_1$> <$p_5 \rightarrow G_1$>
(Partition 2)
<$p_3 \rightarrow G_2$> <$p_3 \rightarrow G_2$> <$p_3 \rightarrow G_2$> <$p_3 \rightarrow G_2$> <$p_3 \rightarrow G_2$>
<$p_3 \rightarrow G_1$> <$p_3 \rightarrow G_2$> then $G_1 = G_2$
(Partition 1)
<$p_1 \rightarrow G_1$> <$p_2 \rightarrow G_1$> <$p_3 \rightarrow G_1$> <$p_4 \rightarrow G_1$> <$p_5 \rightarrow G_1$>
(Partition 2)
<$p_3 \rightarrow G_1$> <$p_4 \rightarrow G_1$> <$p_5 \rightarrow G_1$> <$p_6 \rightarrow G_1$> <$p_7 \rightarrow G_1$>

Figure 2.4: Merging Stage. The shell sets for partitions 1 and 2 in Figure 2.3 contain groups that share common points. The Merging Stage consolidates the sub-groups that span multiple partitions.

2.5 Pipeline Implementation

The DiscFinder algorithms are designed to be implemented using a MapReduce [Dean and Ghemawat, 2004] style of computation. We implemented the DiscFinder pipeline atop Hadoop[9] – an open-source, free implementation of the MapReduce framework. Alternatively, the DiscFinder pipeline could be implemented as a collection of programs that use a programming interface such as MPI [MPI, 1993] and/or OpenMP [Chandra *et al.*, 2000] and are glued together with scripts that execute with the help of a resource manager such as PBS [Bayucan *et al.*, 1999]. Our objective is to explore how the MapReduce programming model, and frameworks such as Hadoop, can be used to tackle large-scale data-intensive analytics found in many science domains.

Hadoop. Hadoop is a distributed framework that has become very popular among the Internet services companies. It is widely used at companies such as Yahoo! and Facebook for their

[9] http://hadoop.apache.org

analytics applications for web graph and social network data. Applications for the framework use a MapReduce programming model. Hadoop has a few features that make it attractive for large-scale data-intensive processing: its support for processing datasets larger than the available memory of the compute resources; its ability to scale the processing to handle very large datasets; its resilience to individual component failures; its relatively simple programming model; and, its community supported open-source implementation.

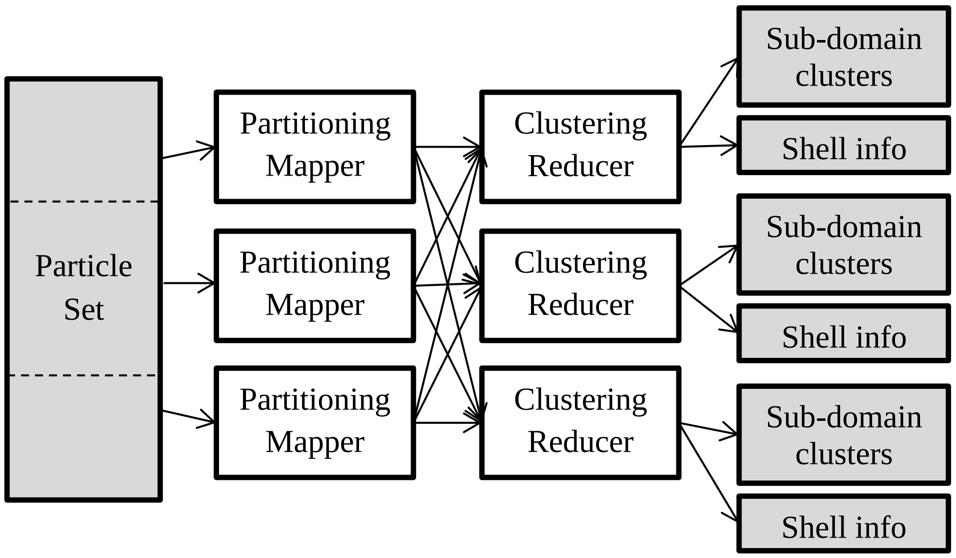


Figure 2.5: DiscFinder partitioning and clustering stages. This is the central MapReduce job in the DiscFinder pipeline.

As a first approximation, Hadoop provides a mechanism for applying a user-defined procedure (map function) over a large dataset, then grouping the results by a key produced by the map function, and then applying another user-defined procedure (reduce function) to the resulting groups. The execution of these functions is distributed over a compute cluster. The framework handles the partitioning of the input data, the data communication, the execution of the tasks and the recovery from errors such as hardware failures and software crashes. For example, Figure 2.5 illustrates a MapReduce job that belongs to the DiscFinder pipeline. The input particle dataset is split across three map tasks that perform the partitioning stage by assigning a key (partition box id) to each input record (particle). The framework collects all the intermediate records that have the same key (box id) such that a reducer process receives all the records with the same box id. Separate distributed processes execute the reduce function that, in this case, perform the clustering pipeline stage. Each reducer produces a subset of the output dataset.

The map and reduce functions may execute in the same set of computers. The datasets are available to the tasks on a distributed file system (DFS). Hadoop works with a variety of DFSs including parallel file systems, such as PVFS2, and Hadoop's own HDFS. HDFS is modeled after the Google File System [Ghemawat *et al.*, 2003]. It is designed to aggregate the storage capacity and I/O bandwidth of the local disks in a compute cluster. Adding hosts to the file system results in capacity and bandwidth scaling. In a typical Hadoop deployment, the computation is carried out in the same computer used for HDFS. The Hadoop job scheduler makes an effort to place the computation in the hosts that store the data.

DiscFinder Pipeline. The DiscFinder pipeline (Figure 2.2) is implemented as a succession of Hadoop MapReduce jobs and sequential programs written in Java. The sampling and splitting stages are implemented as one MapReduce job, the partitioning and clustering phases make up another job. The merging stage is implemented as a stand-alone Java program. The relabeling phase is a distributed map-only job.

The procedures shown below are the high-level driver for the DiscFinder pipeline. It takes as input the astrophysics dataset P and the grouping parameters (here τ) and produces as output a list with mappings from particles to groups. Remember that the input set P contains a set of tuples where each tuple corresponds to a particle in the dataset. The tuple contains information about the particle such as its identifier and other attributes such as its position and velocity among others.

```
Procedure DiscFinder(P, τ): Driver for the DiscFinder pipeline.
Input: P is a set of N points, where each point p_i is of the form <parId, attributes>
Input: τ is the linking distance parameter for FOF
Output: Mapping of particles to groups.
// Run a map reduce job for the sampling phase
1 MapReduce( SamplingMap, SamplingReduce );
/* Use map reduce for point partitioning points and running the groupfinder distributedly */
2 MapReduce( PartitionMap, ClusteringReduce );
3 CollectShell() // Collect the shell data into a central location
4 UnionFind() // Merge groups across partitions
// Finally, use a map-only job to relabel the points
5 MapReduce( RelabelMap, NullReducer );
```

The first step (Line 1) corresponds to the dataset sampling in the processing pipeline. It is carried out using a distributed MapReduce job. The map and reduce functions for this step are shown in detail below in the procedures *SamplingMap* and *SamplingReduce*.

Sampling and Splitting. The sampling stage is implemented as a map function in the first job. Each map task randomly selects a subset of particles. A single reducer task in this job implements the splitting stage. The *CubeSplit* procedure (line 5 in the SamplingReduce procedure), takes the output of the sampling and, using a *kd*-tree, splits the domain into partitions that are used in the subsequent MapReduce job. The partition description is written to HDFS.

```
Procedure SamplingMap(Int Key, Particle Value)
Input: Key is a randomly generated integer value for each point.
Input: Value contains relevant payload values for the particle, such
as <parId, position>
Input: c is a pre-defined prime integer to indicate the sampling rate.
We set c = 16001 in our experiments.
Output: A subset of sampled particles
/* Key is used to sample the input points, since the attribute is
randomly generated */
1 if Key % c == 0 then
2         EmitIntermediate(1, Value.position)
3 end if
```

```
Procedure SamplingReduce(Int Key, Iterator Values)
Input: Key is emitted by the SamplingMap procedure.
Input: Values contains the positions of the sampled particles.
Output: Partitioning scheme for the input dataset
1 List<Particle> particleList = null
2 foreach v in Values do
3         particleList.add(v)
4 end foreach
/* Use particleList to build a kd-tree and use τ to generate the
boundary of each extended cube. */
5 Output(CubeSplit(particleList))
```

Partitioning and Clustering. The second MapReduce job executes the partitioning and clustering stages, which correspond to the bulk of the processing in the pipeline. The partitioning stage is executed by the map tasks (See Figure 2.5). Each task takes a portion of the particle dataset and executes the function shown in procedure *PartitioningMap*. This procedure is executed for each input particle. It receives as parameters the particle's identifier and position. First, it determines the partition in which this particle falls (Line 1) and emits a tuple with the partition (box.id) as the key, and the particle id and position as the values. Additional tuples are emitted for particles that fall in the shell of any other partition(s) (Lines 3–7).

The framework groups the tuples having the same box id and sends them to a reducer task and calls the *ClusteringReduce* procedure for each group of values that have the same key. This procedure executes the external group finder implementation (Line 1), specifically the UW FOF sequential implementation [UW-FOF]. It involves creating an input file in the local file system for the external program, launching the executable and reading the output produced by the program from the local file system. The group membership information for particles that lie in the shell is stored in a separate file (Lines 4–6).

```
Procedure PartitioningMap(pId, position)
Partition particles into overlapping boxes.
Input: pId → particle identifier.
Input: position → particle position (<x, y, z>).
Output: Tuple → <key = boxId, value = (pId, position)>
1 box ← getPartitionBox( position )
2 EmitIntermediate( box.id, pId, position )
3 if position in shell(box) then
      // Emit tuples for adjacent boxes
4     foreach oBox: getAdjBoxes( position, box ) do
5         EmitIntermediate( oBox.id, pId, position )
6     end foreach
7 end if
```

```
Procedure ClusteringReduce(boxId, particles)
Apply the sequential group finder to a partition (box).
Input: boxId → partition identifier.
Input: particles → list of tuples <pId, position>.
Output: Particle membership list <pId→gId>. where pId: particle id, gId: group id.
// Run local group finder
1 particleToGroupMap = groupFind( particles )
2 foreach particle in particleToGroupMap do
3     Emit( particle.id, particle.groupId )
      // Write in a separate file the group membership for particles in the shell
4     if particle in shell(box) then
5         Output( particle.id, particle.groupId )
6     end if
7 end foreach
```

Merging. The merging stage is implemented as a sequential Java program that reads the files with the shell information, which were generated by the ClusteringReduce procedure. The merging code executes the union-find algorithm and writes out the relabeling rules only for groups that span across partitions.

```
Procedure UnionFind(groupMapping)
Execute the union-find algorithm for particles on the shell.
Input: groupMapping is a set of <parId, groupId> pairs.
Output: A set of group-transition rules.
1 parToGroupMap ← EmptyMap;
2 groupToGroupMap ← EmptyMap;
3 foreach (parId, groupId): groupMapping do
4       if groupId not in groupToGroupMap then
            // Add self-mapping: groupId to groupId
5           groupToGroupMap.put( groupId, groupId )
6       end if
7       if parId not in parToGroupMap then
            // Add mapping: parId to groupId
8           parToGroupMap.put( parId, groupId )
9       else
10          groupId* = parToGroupMap.get( parId );
11          while groupId not equal groupToGroupMap.get( groupId ) do
12              groupId = groupToGroupMap.get( groupId )
13          end while
14          while groupId* not equal groupToGroupMap.get( groupId* ) do
15              groupId* = groupToGroupMap.get( groupId* )
16          end while
17          if groupId not equal groupId* then
                // Merge (union) groups groupId and groupId*
18              groupToGroupMap.put( groupId, groupId* )
19          end if
20      end if
21 end foreach
22 foreach (groupId, groupId') : groupToGroupMap do
23      if groupId not equal groupId' then
24          while groupId' not equal groupToGroupMap.get( groupId' ) do
25              groupId' = groupToGroupMap.get( groupId' )
26          end while
27          Output <groupId, groupId'> to group-transition rule file.
28      end if
29 end foreach
```

Relabeling. A map-only job (no reducer) implements the final relabeling stage. Each task reads the groups for a partition and the corresponding relabeling rules. For each tuple <*pid*, *gIdLocal*>, a corresponding tuple <*pid*, *gIdGlobal*> is generated using the relabeling rules to map from local group (*gIdLocal*) to the equivalent global group (*gIdGlobal*).

```
**Procedure** RelabelMap(Int *Key*, Int *Value*)
Relabel groups and particles for each partition. A mapper operates on a set of
particles in a cube partition. Each entry in this set is of the form <parId, groupId>
Input: Group-transition rules produced by the Union-Find procedure.
Input: Key is the particle ID.
Input: Value is the group ID.
Output: Relabeled particle set.
1 parId = Key, groupId = Value
2 if exists mapping from groupId to groupId’ then
          // Relabel by changing groupId to groupId'
3         Emit(parId, groupId’)
4 else
          // Emit original result
5         Emit(parId, groupId)
6 end if
```

2.6 Evaluation

The goal of our evaluation is to test whether DiscFinder is a feasible approach for clustering massive particle astrophysics datasets, and indirectly whether similar approaches can be used for other large-scale analytics in science. We want to measure and understand the overheads introduced by the DiscFinder algorithm and the Hadoop framework, and thus find ways to improve both the application and the framework. We conducted a set of scalability and performance characterization experiments as shown below.

Datasets. In our evaluation we used snapshots (time slices) from the three different cosmology simulation datasets shown in Figure 2.6. BHCosmo simulates the evolution of the universe in the presence of black holes [Di Matteo et al., 2008]. Coyote Universe is part of a larger series of cosmological simulations carried out at Los Alamos National Laboratory [Heitmann et al., 2008a]. DMKraken is a 14.7 billion dark-matter particle simulation carried out by our collaborators at the CMU McWilliams Center for Cosmology . These datasets are stored using the GADGET-2 format [Springel, 2005].

Name	Particle count	Snap size	Snap count	Total size
BHCosmo	20M	850MB	22	18.7GB
Coyote Universe	1.1B	32GB	20	640GB
DMKraken	14.7B	0.5TB	28	14TB

Figure 2.6: Cosmology simulation datasets

Experimental Setup. We carried out the experiments in a data-intensive compute facility that we built in late 2009, named the OpenCloud[10] cluster. Each compute node has eight 2.8GHz CPU cores in two quad-core processors, 16 GB of memory and four 1 TB SATA disks. The nodes are connected by a 10 GigE network using Arista switches and QLogic adapters at the hosts. The nominal bi-section bandwidth for the cluster is 60 Gbps. The cluster runs Linux (2.6.31 kernel), Hadoop (0.19.1), PVFS2 (2.8.1) and Java (1.6). The compute nodes serve both as HDFS storage servers and worker nodes for the MapReduce layer. A separate set of 13 nodes provide external RAID-protected storage using PVFS2. Hadoop is configured to run a maximum of 8 simultaneous tasks per node: 4 mappers and 4 reducers.

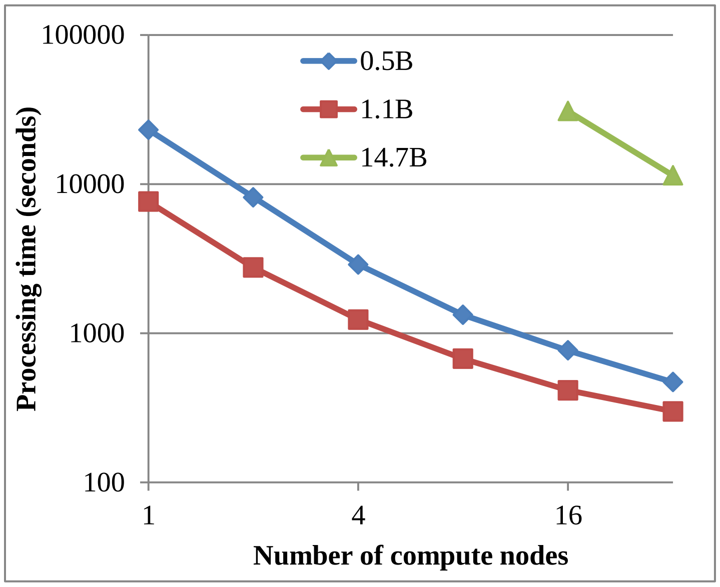
Datasets Pre-processing. The original input data is in the GADGET-2 format, which is a binary format used by astrophysicists. To fit it naturally to the Hadoop framework and speed up implementation, we converted all the input data to a simple text format. The whole conversion costs significantly (e.g. 11 hours for the 14.7 Billion point dataset), although each dataset only needs to be processed once.

2.6.1 Scalability Experiments

With these experiments we want to determine whether DiscFinder can be used to cluster datasets that are much larger than the aggregate memory of the available compute resources, and how its running time changes as we vary the available resources. Similar to the evaluations of compute-intensive applications, we performed weak and strong scaling experiments. However, the results are presented in terms of number of compute nodes, as opposed to number of CPUs. This is more representative of the evaluated application, where the memory capacity and I/O bandwidth are the bottlenecks. Due to the nature of how tasks are managed in Hadoop, there is no one-to-one mapping between tasks and number of CPU cores. The results reported below are the average of three runs with system caches flushed between runs. Some of the running times include partial task failures that were recovered by the framework and allowed the job to complete successfully.

[10] http://wiki.pdl.cmu.edu/opencloudwiki/bin/view/Main/ClusterOverview

In the strong scaling experiments, the total work is kept constant and the work per host changes inversely proportional to the number of compute nodes. In the weak scaling experiments, the work per host is kept constant and the total work grows proportional as compute resources are added. We varied the number of compute hosts from 1 to 32. The results are shown in Figures 2.7 and 2.8. The X axis in these figures is the cluster size (number of worker nodes) in log scale.



Nodes	0.5 B	1.1 B	14.7 B
1	23107	7625	n/a
2	8149	2754	n/a
4	2890	1234	n/a
8	1326	675	n/a
16	767	415	30816
32	470	299	11429

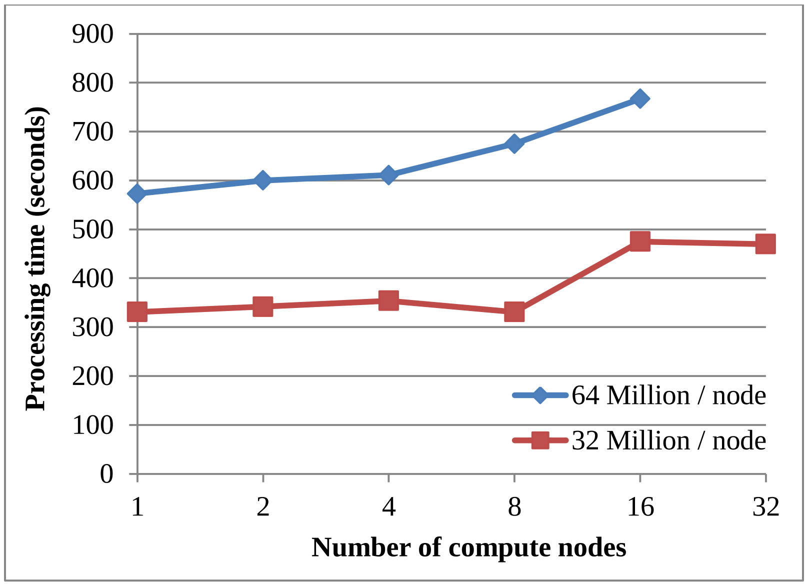
Figure 2.7: Strong scaling.

Strong Scalability. Figure 2.7 shows the strong scalability of the DiscFinder approach. The Y axis is the running time in log scale. The curves correspond to different dataset sizes of 14.7, 1.1, and 0.5 billion particles. Notice that the 14.7 billion dataset is larger than the memory available for the experiments (16 and 32 nodes). The same applies for various scenarios in the cases for 1.1 and 0.5 billion particles. Linear scalability corresponds to a straight line with a slope of –1, where the running time decreases proportionally to the number of nodes. The DiscFinder running time is not linear. With a small number of nodes, the running time actually is superlinear[11], which probably due to that not enough memory are provided to the Hadoop framework (so too much disk access, and possible memory thrashing slows down the computation job), and with a larger number of nodes each node performs too little work and the framework overhead dominates.

Weak Scalability. To perform the weak scaling experiments, we extracted sub-regions of the different datasets to match the appropriate sizes needed for the experiments. The Y axis in the weak scaling graph (Figure 2.8) is the elapsed running time in seconds (linear scale). The curves

[11] Similar phenomena were reported for other Hadoop jobs, including Terasort, CloudBurst (https://blogs.oracle.com/BestPerf/entry/20090920_x2270m2_hadoop), and PageRank (http://cs264.org/projects/web/Porter_Judson/brownell-porter/).

correspond to different problem sizes of 64 (top) and 32 (bottom) million particles per node. The point of 32 nodes in the X axis for the 32M/node curve corresponds to a problem size of 1 billion particles. Similarly, the 16 node, 64M/node curve also has a total problem size of 1 billion particles.



Nodes	64M / node	32M / node
1	573	331
2	600	342
4	611	354
8	675	331
16	767	415
32	n/a	470

Figure 2.8: Weak scaling values in seconds.

The best running time in our experiments lies around a sweet spot between 4 to 8 nodes for datasets smaller than 1 billion particles. We expect non-negligible overheads, introduced by the framework and the approach, due to the incurred I/O, data movement and non-linear operations such as the shuffle/sort in the job that performs the partitioning and clustering stages. A non-negligible amount of memory in each node is used for the HDFS processes. The shuffle/sort operation also uses large amounts of memory per reducer task when the data sizes are large. The running time in all cases exhibits a non-linear trend, especially for larger number of nodes where the framework overheads dominate the running time. Even with all the aforementioned overheads and task failures, it was possible to process and cluster particle datasets that were much larger than the available memory.

2.6.2 Performance Characterization

We are interested in gaining insights about the DiscFinder running time. We performed additional experiments to break down the total running time into the time spent in each stage of the pipeline. In addition, we split the running time of the second job (Partitioning and Clustering) into the phases corresponding to finer-grained operations in that job. For this set of experiments we used 32 nodes and focused on the largest dataset (14.7 billion particles). The phases for the running time breakdown are the following.

1. *Sampling*: This corresponds to the sampling stage of the pipeline.

2. *Splitting*: This stage builds a *kd*-tree to spatially partition the space.

3. *Loading input data*: This refers to reading the particle dataset from HDFS. This step is performed in the map phase of the main MapReduce job (PartitioningMap).

4. *Loading box index*: Time spent loading the index that contains the partition information. This is done in the PartitioningMap procedure.

5. *Box search*: This is the time required to search the spatial partition information in the map phase in the PartitioningMap procedure.

6. *Shuffling / Sorting*: This is the time required to move data from the mappers to the reducers in the main MapReduce job (from PartitioningMap to ClusteringReduce).

7. *FOF data generation*: This is the time required in the reducer to generate the input data for the external group finder program.

8. *FOF execution*: This is the time needed to run the external group finder.

9. *Merging*: Pipeline merging stage.

10. *Relabeling*: Pipeline relabeling stage.

We conducted extra experiments from the start of the pipeline to each of above phase. Using the time difference between these experiments we have acquired the running time of each phase. For instance, the running time of step4 is measured by the difference between experiment of phases 1–4 and experiment of phases 1–3.

The detailed breakdown of the running time for each of these steps is shown in Figure 2.9. Columns 2 and 3 show the absolute (in seconds) and relative time for the unmodified implementation of the pipeline. The relative time is expressed as a percentage of the total running time. The data shows that the Sampling/Splitting, Box search and Shuffle/sort steps account for about 80% of the running time. The time spent in the sampling/splitting step was unexpectedly high. The long time spent in the box search step was due to an inefficient implementation of the search. An incorrect setting of a Hadoop parameter caused the shuffle/sort step to take longer than necessary.

Performance Improvements. We performed a set of optimizations to improve the overall performance of the pipeline. The breakdown of the running time for the improved version is shown in columns 4 and 5 of Figure 2.9. Column 4 has the absolute time for each step in seconds and column 5 contains the relative time with respect to the total running time of the improved version. Column 6 shows the speedup for that step relative to the baseline. The overall speedup is 3X. However, do not read too much into this result. What it really means is that it is easy to introduce performance bugs that may lead to inefficient implementations. Below are the anecdotes of our debugging experience.

Step	Base		Improved		Speedup
	Second	Relative time %	Second	Relative time %	
Sampling	1555	13.4%	859	22.5%	1.8
Splitting	62	0.5%	62	1.6%	1.0
Load particles	229	2.0%	232	6.1%	1.0
Load box idx	3	0.0%	12	0.3%	0.3
Box search	3422	29.5%	122	3.2%	28.0
Shuffle/sort	4363	37.6%	1000	26.2%	4.4
FOF data gen.	762	6.6%	320	8.4%	2.4
FOF exec	576	5.0%	584	15.3%	1.0
Merging	151	1.3%	137	3.6%	1.1
Relabeling	486	4.2%	482	12.7%	1.0
Total	11609	100.0%	3810	100.0%	3.0

Figure 2.9: Breakdown of the DiscFinder elapsed time. The experiments were running on a snapshot of the DMKraken dataset (14.7 billion particles) on 32 worker nodes.

- Improving the Sampling/splitting Phase. In this step only a very small number of particles need to be sampled, but our implementation as a natural MapReduce program still had to read the full dataset, which incurred extra overhead. The current solution consists of performing the sampling outside Hadoop in a separate stage. The sampling time is still relatively high. We are working on alternate solutions to further decrease the sampling time using some of the sampling facilities available in later versions of Hadoop (0.20.x).
- Speeding up Box Lookups. The performance of the box lookup was affected by the initial implementation choice. We had used a simple linear search mechanism for this structure. However, this box lookup is performed at least once for every particle. On aggregate, the lookup time becomes significant. Replacing the lookup mechanism with a routine that uses a O(log *n*) algorithm, where *n* is the number of partitions, provided major benefits.
- Adjusting Number of Reducers. In this set of experiments the particles are split into 1024 spatial partitions. At first, it was only natural for the domain application developer to set the total number of reduce tasks equal to the total number of partitions. In this way, each reduce task would process a single partition. This is the same approach used in high performance computing applications, where the number of partitions matches both the number of processes and processors. During the execution of the original DiscFinder implementation, we noticed in our cluster monitoring tool that the network bandwidth consumption had a cyclic behavior with periods of high utilization followed by idle periods. This effect was caused by multiple waves of reduce tasks fetching data produced

by the map tasks. By setting the number of reducers for the job to (numberOfNodes – 1) × reducersPerNode = 124, all the reduce tasks were able to fetch and shuffle the data in a single wave, even in the presence of a single node failure. In this scenario, a reduce task processed multiple domain partitions. This adjustment required no code changes, as this is the normal mode of operation for the framework.

Although the running time significantly decreased after these modifications, there is clearly room for additional improvement, both at the algorithmic level in the application and in terms of efficiency in the framework.

2.6.3 Future optimizations

- Fine tuning the size of the memory buffers used for the shuffle/sort step in Hadoop jobs.
- Re-designing the pipeline so the external FOF can run on the map phase as opposed to the reduce phase to make more efficient use of the memory, which would allow for higher parallelism and more efficient use of the memory.
- Reading binary data directly: Loading the data using a binary Hadoop reader to avoid preprocessing step.
- Better sampling and box lookup implementation.
- Shuffle: Text vs. binary shuffle transfers. Using a binary representation for transferring data between map and reduce during the shuffle phase.
- Shuffle: Compressing the intermediate tuples, to reduce the network I/O.
- Producing compressed output in the last stage (relabeling).

2.7 Potential utilization of our proposed technique

Currently the most popular large-scale Friends-of-Friends solvers are based on MPI, e.g. Ntropy [Gardner et al., 2006]. Those solutions work best for large-scale cosmological simulations, because the simulations are normally conducted in supercomputers running MPI[12], and it is convenient to directly use the simulation output in an MPI cluster, rather than moving data to a Hadoop cluster first.

Furthermore, some advantages of Hadoop over MPI do not stand out in this situation as of now. For example, Hadoop handles failures automatically, and users do not need to worry if an individual node goes down. MPI itself does not provide this functionality. However, currently when large-scale cosmological simulations are being conducted, failures do not occur often. Our collaborators indicated that in their simulations using 98,304 cores, failure happened "a few

[12] MapReduce/Hadoop is inefficient at running cosmological simulations. MapReduce is designed to run batch processing jobs, not iterative jobs or jobs with too much communications between compute nodes. Generally, the traditional high performance computing (supercomputer, MPI) is designed for compute-intensive jobs, while Hadoop/MapReduce is good for data-intensive jobs.

times" during a 7-day span, but not a serious concern. They also wrote checkpoints every 6~7 hours (not only for failure prevention, but also at the end of each allocated time slot).

Although our Hadoop implementation has not yet been utilized by astronomers[13], they do think that it has several attractive features. For one, although the supercomputers are very powerful, they are not free to access: For example, in order to use the 98,304 cores of Kraken supercomputer (almost all of its cores), our collaborators can only take full control of it for 24 hours per week. On top of that, they need to write proposals and make their data public. These limitations add difficulties for astronomers to use supercomputers.

If a user cannot access to the many cores of supercomputers, then it is harder for him to use MPI-based solutions to analyze a large dataset, since most MPI-based solutions require all the data to be loaded into memory. In contrary, our Hadoop solution does not have that requirement and it only needs all the data in each partition to fit into the memory of a reducer. This is a very big plus.

Furthermore, Hadoop provides automatic data partition to users. This is helpful especially when a user wants to re-partition data for a different cluster setting (e.g. use different number of nodes, and/or different set of nodes). The Hadoop framework automatically provides all above operations. Actually, for each partition of input data, Hadoop even tries to allocate the nearest Map task to process it, making better exploitation of data locality. An MPI-based alternative requires users to write additional codes for these functionalities, which is a lot of works and error-prone.

Finally, although the benefit of Hadoop's fault tolerance is not standing out now, one of our astronomer collaborators said the feature is still attractive because comparing to MPI, he feels safer to use Hadoop to execute long-running tasks. To summarize, all these Hadoop's features dealing with large data make our implementation a very flexible tool. Our collaborators indicated that they would try it in some scenarios, or use it to complement their current tools.

2.8 Conclusion

The analysis of state-of-the-art and future astrophysics datasets is difficult due to their size. Group finding and other analysis techniques need to be scaled to operate on these massive datasets. DiscFinder is a novel data-intensive group finder that scales to datasets with tens of billions of particles. DiscFinder has enabled the analysis of the largest state-of-the-art cosmology datasets. Nevertheless, its first implementation has relatively high overheads introduced by application algorithms and framework implementation. We described different approaches to improve its performance. As the analysis of truly very large datasets requires a data-intensive

[13] Although our technique has been used as an I/O intensive scientific workload in the evaluation of computer systems [Fan *et al*., 2011] [Tantisiriroj *et al*., 2011].

approach, there are opportunities and needs to improve the performance and extend the programming models to better support analytics applications for science.

Chapter 3 Exact and Approximate Computation of a Histogram of Pairwise Distances between Astronomical Objects

3.1 Background

In this chapter, we turn to the following astrophysics problem: building a histogram of pairwise distances between celestial objects, the astronomical *correlation function* problem [Peebles, 1980]. The histogram essentially measures the distribution of astronomical objects at given distances, and it has been widely used by astrophysicists.

For instance, one application of correlation functions is to help cosmologists better understand the universe. Now cosmologists have come to the consensus that the universe is not only expanding, but expanding at an accelerated rate. How to better understand and measure the acceleration of our expanding universe has become one of the most important questions in modern cosmology. Relatively, the recession velocities of distant astronomical objects are easier to determine via the Doppler effect, but the exact distance of distant objects is extremely difficult to measure very accurately.

Recently cosmologists studied the Baryon Acoustic Oscillations (BAO), a complex astrophysical effect. What BAO tells us is a precise distance formed at the early stage of the universe – currently measured at about 490 million light years – that separates the hypothesized dark matter and other baryonic matter. Since both the baryons and dark matter continued to attract matter and eventually form galaxies, cosmologists expect a greater number of galaxies that are separated by that specific distance. Although people cannot observe this phenomenon directly, one can measure it by looking at the separations of large numbers of galaxies, i.e. using tools like correlation functions.

Moreover, since the physics of BAO is simple enough, there are few uncertainties in the measurement, making it the most accurate cosmological distance indicator (a *standard ruler*) known as of now[14]. As a result, the study of BAO has been very popular recently. For example, the BAO signal has been detected in the latest Sloan Digital Sky Survey [Anderson et al., 2012], as shown in Figure 3.1.

[14] More details about the current applications of BAO can be found in the project description of SDSS-IV: http://www.sdss3.org/future/sdss4.pdf.

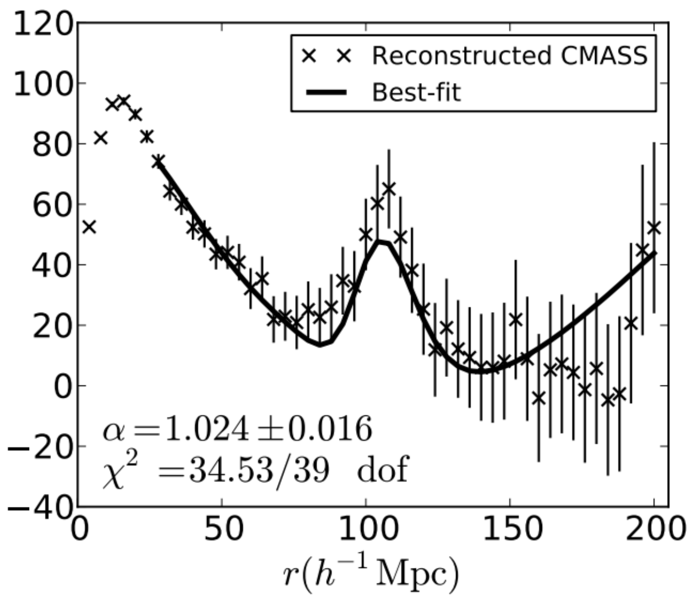


Figure 3.1: BAO signal detected in the Sloan Digital Sky Survey. There is a peak around 100~110 Mpc/h[15] (equivalent to about 490 million light years), which indicates that more galaxies are separated by that distance.

The current datasets that the BAO is calculated on is getting larger. For example, BigBoss[16] has collected two millions red luminous galaxies, on which cosmologists calculate the correlation function. The Euclid project[17] is expected to observe 20 million red luminous galaxies. Moreover, in order to determine whether indeed a *greater* number of galaxies are separated by some distance in the dataset, we also need to calculate the correlation function over a random point set[18] which, to get statistical significance, should be at least 50 times larger than the original data. As a result, in the near future cosmologists will need to calculate correlation functions over billions of astronomical objects for the BAO application. And correlation functions are also used in many other scenarios.

In this chapter, we provide a survey on different ways to calculate the correlation function. Since a naive solution takes quadratic time, a more efficient solution is necessary on large datasets. Researchers have proposed sequential approaches to better compute correlation functions. In particular, Gray and Moore used *kd*-trees [Gray and Moore, 2000], and Belussi and Faloutsos developed an approximation algorithm based on fractal dimensions

[15] "Mpc" is short for Megaparsecs. 1 parsec is equal to 3.26 light years, and one Megaparsec is 10^6 parsecs. "h" represents the Hubble constant.
[16] http://bigboss.lbl.gov/
[17] http://sci.esa.int/euclid
[18] Since the coverage maps of most data surveys are in irregular shape, we usually cannot get an analytical solution theoretically, but have to generate random data in the irregular space (Monte Carlo) and calculate the correlation function on it.

[Belussi and Faloutsos, 1995]; however, our experiments have shown that the existing sequential techniques are either very slow or give inaccurate approximations.

Scientists also presented distributed computing solutions [Dolence and Brunner, 2008] [Ponce et al., 2011] [Chhugani et al., 2012]. Although some can quickly calculate correlation functions on datasets with up to billion objects [Chhugani et al., 2012], they requires powerful yet not easily accessible supercomputers with tens of thousands of computing cores. In this chapter, we propose a sampling method and combine it with the *kd*-tree technique, which results in an efficient and accurate approximation of the correlation function [Fu et al., 2012a]. The proposed technique can also be easily distributed, resulting in a powerful tool that only uses tens or hundreds cores and applicable to datasets with multi-billion objects.

3.2 Problem

We assume that each astronomical object is a point in three-dimension space with known coordinates. We are given a set of N astronomical objects, denoted $p_1, p_2, \ldots, p_N$, and a strictly increasing series of $M + 1$ distances, denoted $d_0, d_1, \ldots, d_M$, defining the bins of a histogram.

For each index i from 1 to M, we need to determine the number of object pairs such that the distance between each pair is between d_{i-1} and d_i:

$$cf(i) \text{ is the number of pairs } (p_u, p_v), \text{ where } u < v,$$

$$\text{such that } d_{i-1} \leq dist(p_u, p_v) < d_i$$

where $dist(p_u, p_v)$ is the Euclidean distance between p_u and p_v.

We assume that the given sequence of distances is a geometric progression. That is, we are given the distance d_0 and a constant $C > 1$, and we need to compute the correlation function for the distances $d_0, C \cdot d_0, C^2 \cdot d_0, \ldots, C^M \cdot d_0$.

We conducted experiments on a dataset of 4.5 million objects, with coordinate values between 0.0 and 40.0 (the unit in this dataset is Mpc/h [Di Matteo et al., 2008]). The dataset is provided by our collaborators from McWilliams Center for Cosmology at Carnegie Mellon University. We compute the correlation function with the following parameters unless indicated otherwise: $M = 257$, $d_0 = 0.001$, and $C = 1.044$.

We have implemented all the sequential algorithms in Java 1.6 and tested them on a 2.66GHz Intel Core 2 Duo desktop with 2GB memory.

3.3 Existing Solutions

3.3.1 Naive algorithm

We first consider the straighforward algorithm shown in Figure 3.2, which iterates over all pairs of objects, thus taking $O(N^2)$ time. In Figure 3.3, we compare the actual running time and the theoretical complexity of the algorithm.

```
**Output**: Counters $cf_1, cf_2, \ldots, cf_M$

for $i = 1$ to $M$ do $cf_i = 0$
for $u = 1$ to $N - 1$ do
   for $v = u + 1$ to $N$ do
      find $i$ such that $d_{i-1} \le dist(p_u, p_v) < d_i$; $cf_i$ ++
```

Figure 3.2: Naive algorithm.

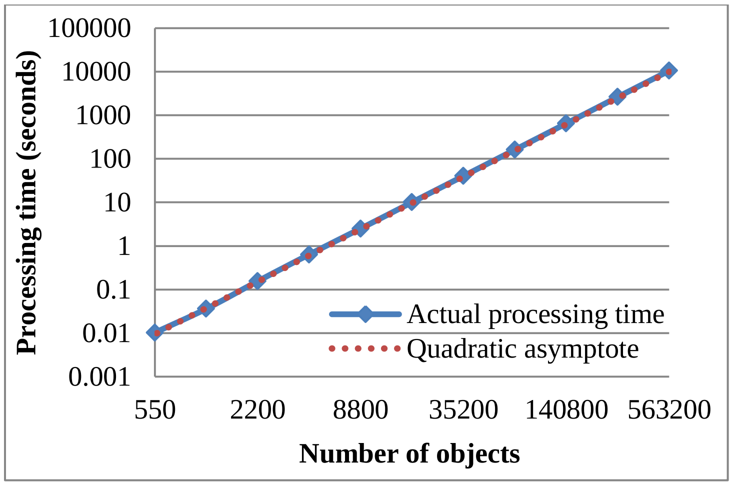


Figure 3.3: Running time of the naive algorithm, which is quadratic to the number of objects. Note that both axes are on logarithmic scales.

There are two approaches to determine in which histogram bin each object-pair falls: a single logarithm operation (since the distance sequence is a geometric progression) and binary search. While the theoretical time complexity of the single logarithm operation is constant, its computation takes multiple CPU circles, and in practice it is slower than binary search for a short sequence of distances. In Figure 3.4, we compare the running time of these two approaches. The results show that the logarithm computation is more efficient only when the length of the distance sequence, M, is larger than 75.

In Figure 3.5, we show the exactly computed correlation function for the dataset of 4.5 million objects. The elasped time of the naive algorithm is 176 hours (7.3 days).

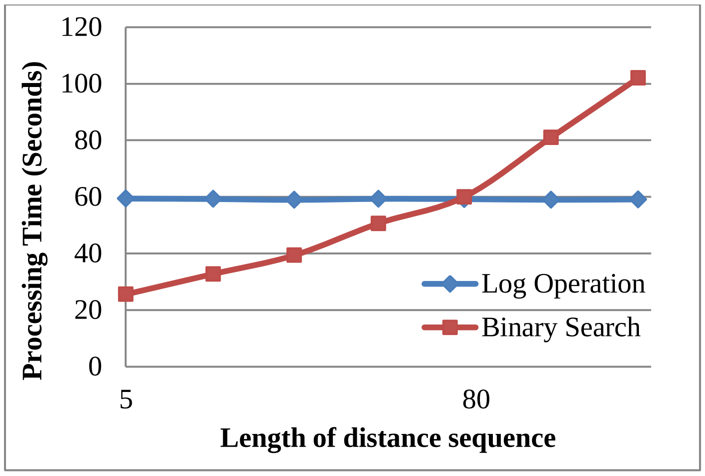


Figure 3.4: Running time of the two approaches in the naive algorithm: logarithm operation and binary search. Note that the horizontal axis is on logarithmic scale. The use of the logarithm operation is faster when the length of the distance sequence is over 75, i.e. $M > 75$.

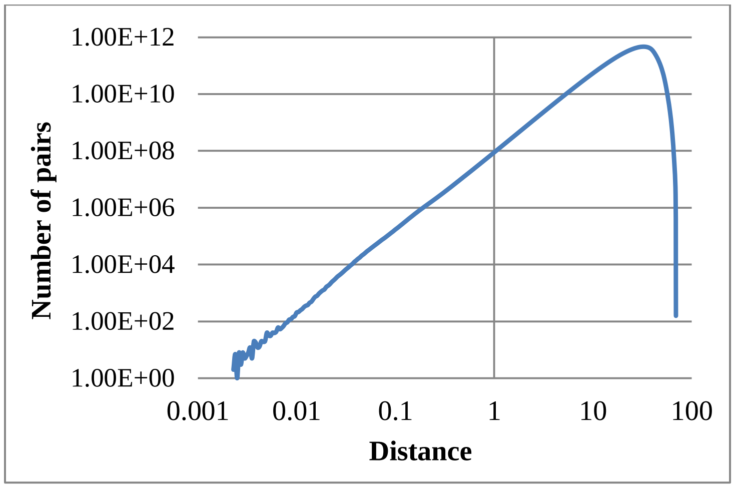


Figure 3.5: The exact correlation function for the set of 4.5 million objects. Note that both axes are on logarithmic scale.

3.3.2 *Kd*-tree algorithm

Gray and Moore used *kd*-tree to accelerate the computation of correlation functions [Gray and Moore, 2000], which is presented in Figure 3.6. Unlike the naive algorithm, their procedure processes one range of distances, d_{i-1} to d_i, at a time. The *kd*-tree structure supports tree pruning, which speeds up the computation. The time complexity of processing a single range of distances is $O(N^{5/3})$. The overall processing time grows as we increase the length M of the distance sequence, as shown in Figure 3.7.

To improve the efficiency, we have developed a new version of the *kd*-tree algorithm, called *multiple-range* algorithm, designed for processing multiple ranges in one pass. The pseudo-code of the multiple-range *kd*-tree algorithm is shown in Figure 3.8. We have conducted experiments on both the original single-range algorithm and the new multiple-range algorithm. The main results are as follows:

a. As shown in Figure 3.9, the single-range algorithm follows the $O(N^{5/3})$ asymptote. The multiple-range algorithm is in practice faster than the single-range *kd*-tree algorithm, but its asymptotic complexity is $O(N^2)$.
b. Why the time complexity of the multiple-range *kd*-tree algorithm is $\mathrm{O}(N^2)$? The reason is that C in our experiments is relatively small, specifically, $C = 1.044$, which means that the ranges used in constructing the histogram of distances are very fine-grained, and the tree pruning in this situation does not lead to the reduction of the asymptotic time complexity.
c. Since the distance sequence is relatively long ($M = 257$), the *kd*-tree algorithms in our experiment actually ran slower than the naive algorithm.
d. The processing time of the single-range *kd*-tree algorithm differs significantly for different distances, as shown in Figure 3.10. The results suggest that the pruning is most effective for small and very large distances.

To summarize, we have confirmed that both the single-range and the multiple-range *kd*-tree algorithms have superlinear time complexity. Figure 3.9 further shows that they can be slower than the naive algorithm. They are therefore impractically slow for massive dataset with billions of objects. The *kd*-tree technique is effective for small and very large distances, but not in-between.

```
Output: Counters $cf_1, cf_2, \ldots, cf_M$

for $i = 1$ to $M$ do $cf_i = 0$
construct a kd-tree from the root node. Each node in the kd-tree has a left child and
a right child. The bounding box of each node is calculated and stored. The number
of objects (num) in each node is also calculated.
for $i = 1$ to $M$ do
    $cf_i$ = SINGLE-DISTANCE(root, root, $d_{i-1}$, $d_i$);

Procedure SINGLE-DISTANCE(kd-node $n_1$, kd-node $n_2$, double small, double large)
if ($n_1.num < 1$ || $n_2.num < 1$) then return 0
if ($n_1.num == 1$ && $n_2.num == 1$) then
    //assume $n_1$ includes object $p_1$ and $n_2$ includes object $p_2$
    if ($small \le dist(p_1, p_2) < large$) then return 1
    else return 0
calculate the maximum distance (max) and the minimum distance (min) between
the bounding boxes of $n_1$ and $n_2$.
if ($min \ge large$ || $max < small$) then return 0
if ($min \ge small$ && $max < large$) then
    return $n_1.num \cdot n_2.num$
if ($n_1.num > n_2.num$) then
    return SINGLE-DISTANCE($n_1.left$, $n_2$, small, large) +
            SINGLE-DISTANCE($n_1.right$, $n_2$, small, large)
else
    return SINGLE-DISTANCE($n_1$, $n_2.left$, small, large) +
            SINGLE-DISTANCE($n_1$, $n_2.right$, small, large)
End Procedure
```

Figure 3.6: Single-range *kd*-tree algorithm.

3.3.3 Fractal approximation

The main advantage of the naive and *kd*-tree algorithms is that they provide exact results. On the downside, they are slow for massive datasets. The processing of 4.5 million objects takes several days, and the time grows superlinearly with the number of objects, which means that processing a set with billions of objects would be very slow even on a powerful supercomputer. We now consider the alternative of developing fast approximate algorithms for computing correlation functions.

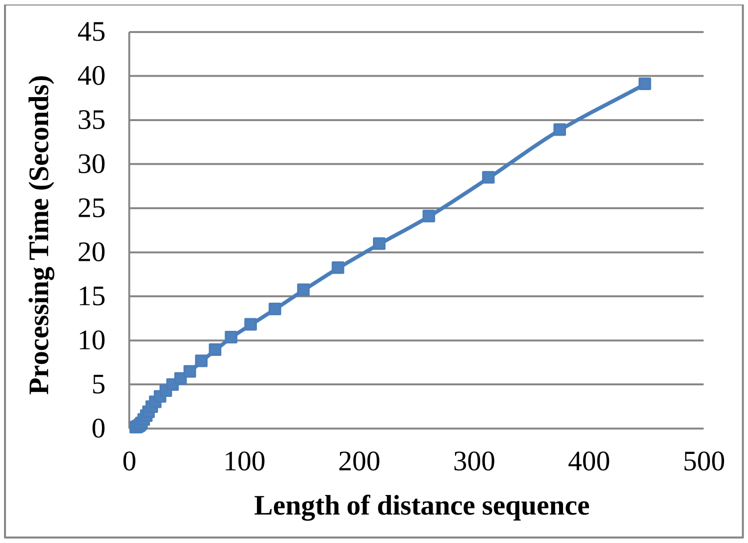


Figure 3.7: Dependency of the running time on the length of distance sequence for the single-range *kd*-tree algorithm. In this set of experiments we only change *C*. We have run this experiment with a dataset of 75 thousand objects.

Belussi and Faloutsos used an approximate technique for computing fractal dimensions of point sets [Belussi and Faloutsos, 1995], which can be readily adapted to approximate correlation functions in two main steps, as shown in Figure 3.11. First, we divide the space into a grid of cubic cells, iterate over all objects, and assign each object to the corresponding cell. Second, we count the number of objects in each cell.

The underlying assumption behind the fractal approximation is that we assume the object density among nearby regions is similar, which leads to two approximation steps. First, when calculating how many objects are within distance r of an object p_i, we do not consider a ball centered at p_i with radius r, but instead use a cube centered at p_i with side length $r \cdot (4\pi/3)^{1/3}$. Since the volumes of the ball and the cube are the same, we expect that the numbers of objects within them is roughly the same. Second, rather than going over each object and counting the number of objects within their corresponding cubes, we use a set of global cubes (the aforementioned *cells*) to represent all the cubes around objects, which greatly reduce the time complexity.

If we represent the grid using a hash table, the time complexity of the fractal algorithm is linear on the number of objects. We show the empirical running time in Figure 3.12. If we process each of the M ranges separately, the overall time complexity is $O(M \cdot N)$. In Figure 3.13, we show the dependency of the running time on the number of ranges. We may further reduce the processing time to $O(N)$ by sorting the cells in Z-order [Morton, 1966].

```
Output: Counters $cf_1, cf_2, \ldots, cf_M$

for $i = 1$ to $M$ do $cf_i = 0$
construct a kd-tree from the root node. Each node in the kd-tree has a left child and a right child. The bounding box of each node is calculated and stored. The number of objects (num) in each node is also calculated.
MULTIPLE-DISTANCES(root, root);

Procedure MULTIPLE-DISTANCES(kd-node $n_1$, kd-node $n_2$)
if ($n_1.num < 1 \,||\, n_2.num < 1$) then return
if ($n_1.num == 1$ && $n_2.num == 1$) then
    //assume $n_1$ includes object $p_1$ and $n_2$ includes object $p_2$
    find $i$ such that $d_{i-1} \le dist(p_1, p_2) < d_i$; $cf_i$ ++
    return
calculate the maximum distance (max) and the minimum distance (min) between the bounding boxes of $n_1$ and $n_2$.
if ($min \ge d_M \,||\, max < d_0$) then return
if ($[min, max] \in [d_{i-1}, d_i)$ for an $i$) then
    $cf_i$ += $n_1.num \cdot n_2.num$;
    return
if ($n_1.num > n_2.num$) then
    MULTIPLE-DISTANCES($n_1.left$, $n_2$)
    MULTIPLE-DISTANCES($n_1.right$, $n_2$)
else
    MULTIPLE-DISTANCES($n_1$, $n_2.left$)
    MULTIPLE-DISTANCES($n_1$, $n_2.right$)
End Procedure
```

Figure 3.8: Multiple-range *kd*-tree algorithm.

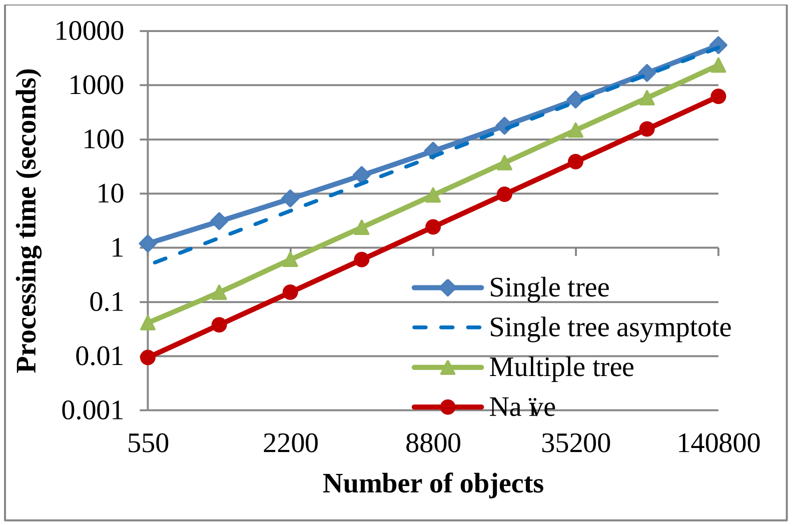


Figure 3.9: Running time of the single-range *kd*-tree algorithm, multiple-range *kd*-tree algorithmm, and the naive algorithm (Figure 3.3). The single-range algorithm follows an $O(N^{5/3})$ asymptote. The multiple-range algorithm has quadratic complexity but it is faster than the single-tree algorithm in practice.

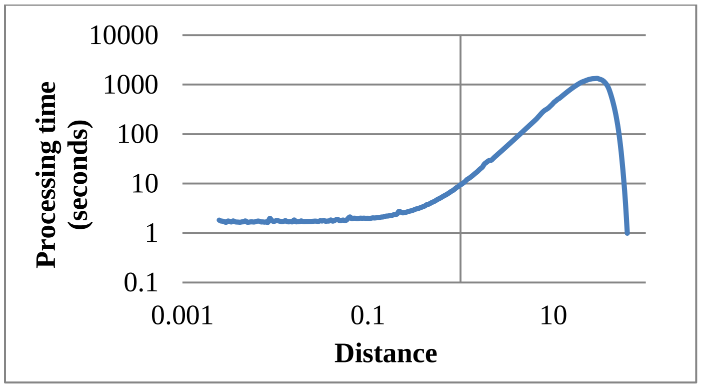


Figure 3.10: The dependency of the running time on the distances for the single-range *kd*-tree algorithm. Note that the vertical axis is on logarithmic scale. This experiment is conducted on a dataset with 450 thousands objects. The results show that the processing of small and very large distances is much faster than the processing of medium distances.

```
Output: Counters $cf_1, cf_2, \ldots, cf_M$

create an array $s_i$, $i = 0$ to $M$
$s_0 = 0$
for $i = 1$ to $M$ do
    divide the space into a grid of cubic cells with side side $d_i$
    we use $f_k$ to denote the number of objects falling in the $k^{th}$ cell
    $s_i = \lg(\sum_k (f_k)^2)$
    $D = (s_i - s_{i-1}) / (d_i - d_{i-1})$
    $cf_i = N (N - 1) (\pi / 6)^{D/3} (2d_i)^D / 2$
for $i = M$ to 2 do
    $cf_i = cf_i - cf_{i-1}$
```

Figure 3.11: Fractal algorithm.

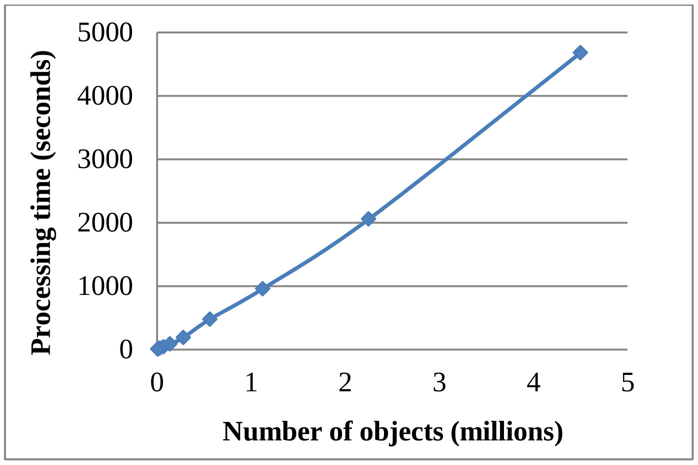


Figure 3.12: Running time of the fractal approximation, which is linear on the number of objects.

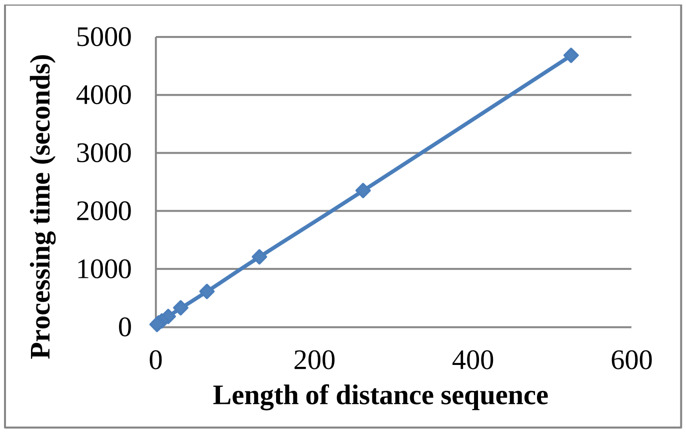


Figure 3.13: Dependency of the running time on the length M of the distance sequence.

While this procedure is fast, the resulting approximation is inaccurate. Belussi and Faloutsos reported that the relative error of the fractal procedure for estimating $\sum_{k=0}^{i} cf_k$ is in the 10–15% range [Belussi and Faloutsos, 1995]. Since in our application the correlation function histograms use cf_k, the resulting error is even greater, achieving as high as 40%.

3.4 Proposed Technique

3.4.1 Sampling algorithm

We next consider the application of sampling to approximate the computation of correlation functions. Specifically, we randomly select S objects from the original set and apply the naive algorithm to this smaller sample. To convert the resulting counters on the sample to the counters on the original set, we multiple each of them by $(N/S)^2$ (Figure 3.14).

To get a more accurate estimate, we repeat the described procedure T times, and then compute the mean $\mu(cf_i)$ and the standard deviation $\sigma(cf_i)$ for each estimation cf_i. According to the central limit theorem, when T is at least 30, the means $\mu(cf_i)$ follow normal distribution, which allows determining their confidence intervals. For instance, the true counter on the original dataset cf_i has a 95% probability to be located within $2\sigma(cf_i)$ away from our estimation $\mu(cf_i)$:

$$P(-2\sigma(cf_i) < |cf_i - \mu(cf_i)| < 2\sigma(cf_i)) = 0.95$$

, which enables us to calculate the maximum relative error of our estimations, each has a 95% probability to be correct:

$$Maximum\ relative\ error = \max\left(\frac{|cf_i - \mu(cf_i)|}{\mu(cf_i)}\right) = \frac{2\sigma(cf_i)}{\mu(cf_i)}$$

In Figure 3.15, we show the maximum relative error of the sampling algorithm. We have found that even a small sample provides a relatively accurate approximation for many distance ranges. For example, if S = 10,000, the sampling algorithm can produce estimates with less than 1% error for distances from 2 to 50.

The overall running time of the sampling algorithm is the sum of (1) time to retrieve samples from the original dataset and (2) the time to conduct the subsequent computations on the samples. We use a straightforward method to select samples from the original dataset (for another implementation using SSDs, see Section 3.4.3), which makes a liner pass through the whole dataset, thus taking $O(N)$ time. For the dataset of 4.5 million objects, this sample selection takes 2.3 seconds. The time of applying the naive correlation function algorithm to the selected samples is $O(T \cdot S^2)$.

```
Output: Mean and standard deviation of $cf_1, cf_2, \ldots, cf_M$

$T = 30$   // $T$ is the number of samples
create two dimensional array $a_{uv}$, $u = 1$ to $M$, $v = 1$ to $T$
for $u = 1$ to $M$ do
    for $v = 1$ to $T$ do
        $a_{uv} = 0$
for $t = 1$ to $T$ do
    randomly select $S$ objects $r_1, r_2, \ldots, r_S$ from $p_1, p_2, \ldots, p_N$
    for $u = 1$ to $S - 1$ do
        for $v = u + 1$ to $S$ do
            find $i$ such that $d_{i-1} \leq dist(r_u, r_v) < d_i$; $a_{it}$ ++
    for $i = 1$ to $M$ do $a_{it} = a_{it} \cdot (N/S)^2$
for $i = 1$ to $M$ do
    $\mu(cf_i) = (a_{i1} + a_{i2} + \ldots + a_{iT}) / T$
    $\sigma(cf_i)^2 = ((a_{i1} - \mu(cf_i))^2 + (a_{i2} - \mu(cf_i))^2 + \ldots + (a_{iT} - \mu(cf_i))^2) / (T(T-1))$
```

Figure 3.14: Sampling algorithm.

Another advantage of the sampling technique is that it has enormous potential to handle a massive dataset: Assume that we have two datasets with different sizes, where their objects follow the same spatial distribution. Then although it still takes more time to select samples from the larger dataset (O(N)), their remaining computations are the same. As a result, the sampling technique is able to process a much larger dataset than other techniques.

On the downside, the sampling error is higher for small and very large distances. Generally, the error of estimating cf_i (Figure 3.15) is in inverse proportion to the value of cf_i (Figure 3.5).

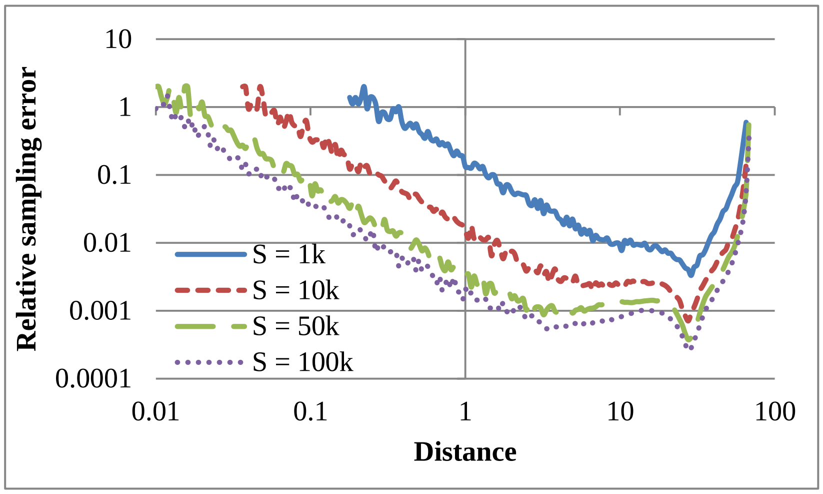


Figure 3.15: Relative error of the sampling technique. We plot the maxiumum errors with 95% confidence interval. The four curves represent experiments with samples of size 1,000, 10,000, 50,000 and 100,000.

3.4.2 Hybrid algorithm

We have shown that the sampling method is fast but inaccurate for small and very large distances (Figure 3.15). On the other hand, the *kd*-tree technique is fast for small and very large distances (Figure 3.10). We can thus obtain better results by combining these two techniques. Specifically, we apply the *kd*-tree algorithm to small and very large distances, and the sampling technique to the distances in the middle.

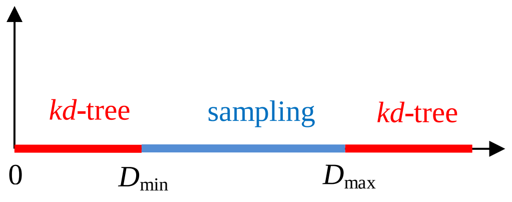


Figure 3.16: Illustration of the hybrid technique.

Input: Required maximum relative error *epsilon*
Output: Mean and standard deviation of $cf_1, cf_2, \ldots, cf_M$

- Find turning points $D_{\min}$ and $D_{\max}$ according to *epsilon*.
- For distances inside $[D_{\min}, D_{\max})$, select the number of sampled objects S according to *epsilon*, and use the sampling algorithm in Section 3.4.1.
- For distances inside $[d_0, D_{\min})$ and $[D_{\max}, d_M)$, use the *kd*-tree algorithm described in Section 3.3.2.

Figure 3.17: Hybrid algorithm.

The related parameter tuning involves setting the "turning points" $D_{\min}$ and $D_{\max}$, and determining the appropriate sample size S. We have selected these parameters based on empirical results, with the purpose of achieving the given accuracy in minimal running time.

In Figure 3.18, we compare the running time of the hybrid algorithm given different allowed approximation errors (*epsilon*).

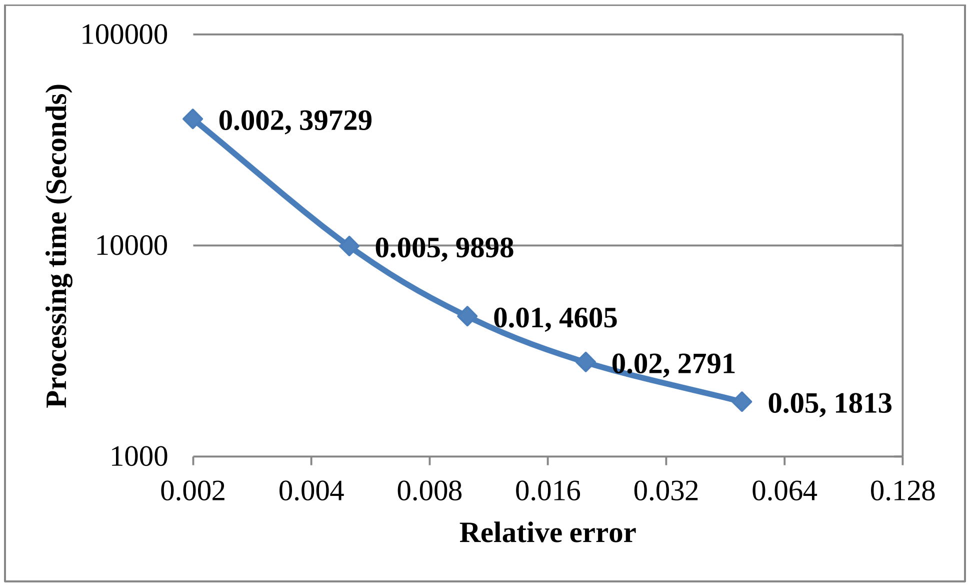


Figure 3.18: Running time of the hybrid algorithm with different allowed errors. The five data points correspond to the errors of 0.2%, 0.5%, 1%, 2% and 5%.

In Figure 3.19, we compare the running time of the hybrid algorithm with other techniques on the set of 4.5 million objects. The proposed hybrid algorithm is at least one order of magnitude faster than the exact computations, even if we limit the approximation error to 0.2%. When the running time of the hybrid algorithm is similar to that of the fractal approximation, its error is at least one order of magnitude smaller.

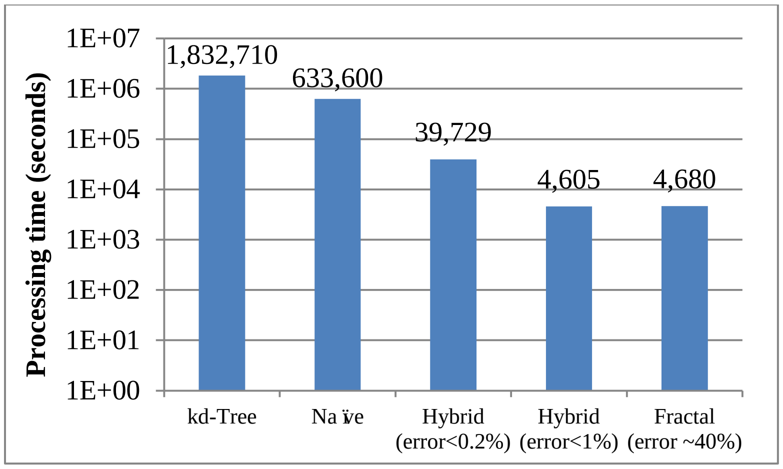


Figure 3.19. Running time of the described techniques on the 4.5 million objects dataset.

3.4.3 Distributed hybrid algorithm

Although our hybrid technique can process a larger dataset than other existing sequential techniques, it is still slow to produce results with very high precision, or deal with even larger datasets. In this section we introduce our efforts to parallelize the hybrid procedure, which further reduces its overall running time.

Our hybrid method consists of the sampling algorithm and the *kd*-tree algorithm. The sampling algorithm is easy to parallelize, but the *kd*-tree algorithm cannot be trivially distributed. So we extended the sampling idea to the *kd*-tree algorithm, by not applying the *kd*-tree computation to the set of all objects, but to several samples from the original dataset, and then we compute the means and standard deviations as similar to the sampling algorithm (Figure 3.14). The samples used in the *kd*-tree computation are larger than the samples used in the naive computation, which ensures sufficient accuracy for small and very large distances.

We next introduce our implemention of the distributed hybrid algorithm using Hadoop. We evaluate its performance on the a compute intensive cluster in Carnegie Mellon University. After that, given the scenario that users do not access a Hadoop cluster themselves, we also discuss the deployment of our implementation on an Amazon cloud computing cluster.

Hadoop implementation

We use Hadoop to implement the distributed hybrid technique. Hadoop provides a convenient distributed computing framework to users, so they can mainly focus on the functionality of their code, rather than worrying about other issues of distributed computing, especially ones related to the processing of large datasets.

Our hybrid algorithm is highly parallelizable and fits well to the Hadoop framework. During the Map phase, we select random samples. During the Reduce phase, we process the selected samples in parallel. The distributed implementation is similar to that of the DiscFinder pipeline in Section 2.5 (procedures PartitioningMap and ClusteringReduce).

We first tested our implementation on the OpenCloud computer cluster, which is a 64-node cluster in Carnegie Mellon University. Each compute node has eight 2.8GHz CPU cores in two quad-core processors, 16 GB of memory and four 1TB SATA disks. The nodes are connected by 10 GigE network.

Our first goal is to see whether our distributed algorithm can be applied to analyze a large dataset. Other than the dataset with 4.5 million objects that we processed in previous sections, we tested our algorithm on two larger datasets: the Coyote Universe [Heitmann et al., 2008a], which contains 1.1 billion objects, and the DMKraken provided by our collaborators at the McWilliams Center for Cosmology at Carnegie Mellon, which contains 5 billion objects. We normalized the coordinates of objects in each dataset to make their bounding boxes equal-sized.

In Table 3.1, we show the running time to process the three datasets using 32 reducers. We issued the query with same distance ranges and maximum allowed error[19]. The results show that our Hadoop implementation is able to process datasets with billions of objects. For larger datasets, most of the elapsed time is spent on the procedure of sample selection (the Map phrase).

Table 3.1: Time of computing correlation functions using with 32 reducers, for the allowed maximum error of 1%. We set $d_0 = 0.006$, $d_M = 65$, $C = 1.044$, and $M = 216$ in this experiment.

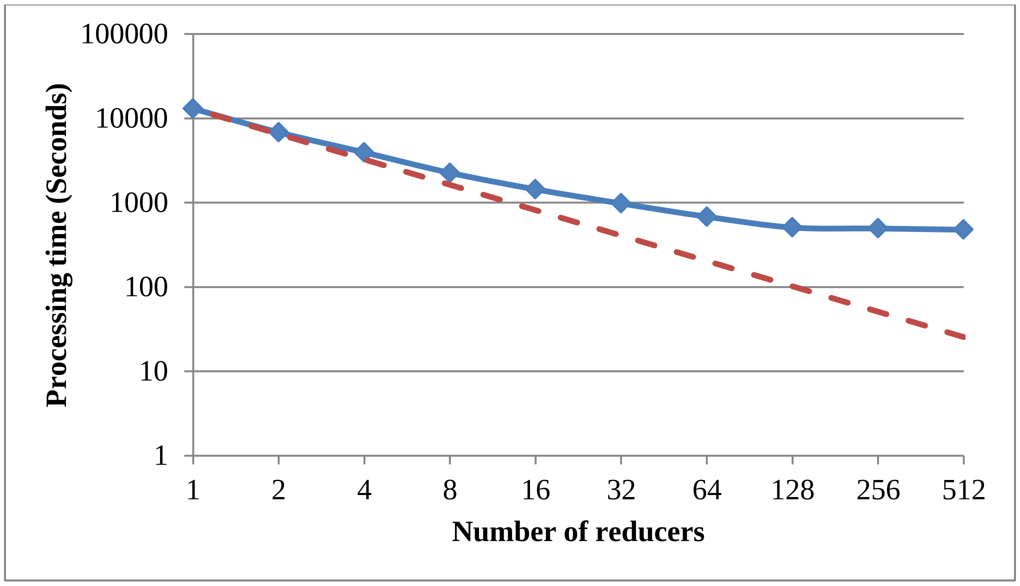
Number of objects in the overall dataset (before sampling)	**Running time (Seconds)**
4.5 million	888
1.1 billion	1529
5 billion	2436

The second goal of our experiments is to evaluate the scalability of the distributed hybrid algorithm, so we conducted a set of strong scalability experiments shown in Figure 3.20. We processed the DMKraken dataset with 5 billion objects using different number of reducers, and plotted the running time. For this set of experiments we issued 512 samples. First in Figure 3.20(a), we put 30,000 objects to each sample; then in Figure 3.20(b), we increased the number of sampled objects to 120,000.

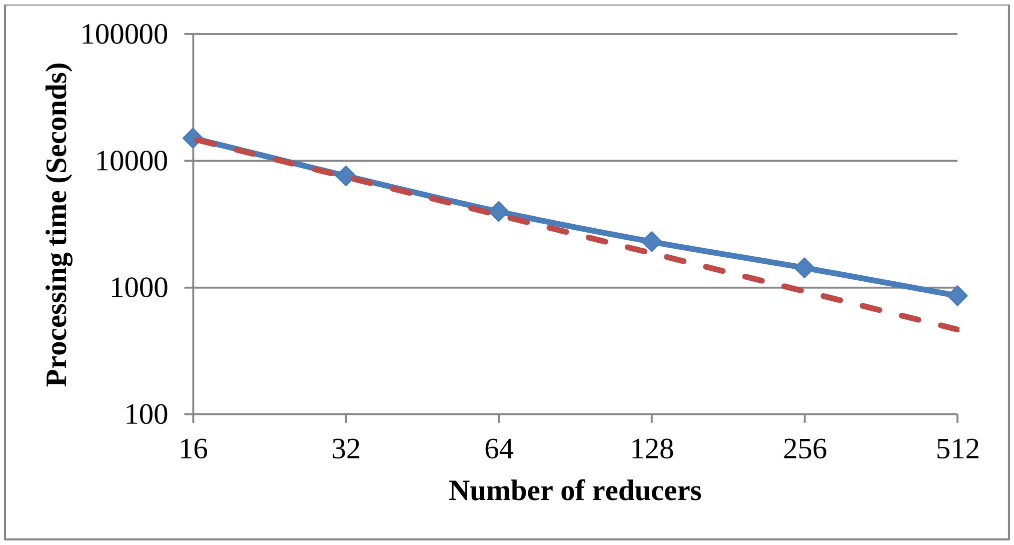
These experiments indicate that the scalability is affected by the number of sampled objects. With more objects in a sample, each reducer having to execute more calculation, and the

[19] We should note that our distributed algorithm has one natural weakness: it cannot effectively calculate the distance ranges where the number of qualifying pair is very small. That is because the calculation on different samples leads to very high variance, so the estimated errors would be very high (Figure 3.15). That's why the distance ranges we use here ($d_0 = 0.006$, $d_M = 65$) is a little bit narrower than previously specified ($d_0 = 0.001$, $d_M = 70$).

overhead of Hadoop framework and data communication become less obvious. The speedup in Figure 3.20(a) begins to decrease around 16 reducers, but with more calculation for each reducer, the speedup in Figure 3.20(b) is perfect up to 64 reducers.



(a) 512 samples, each with 30,000 sampled objects.



(b) 512 samples, each with 120,000 sampled objects.

Figure 3.20: Strong scalability experiments on our Hadoop implementation. Blue line represents actual running time and red dotted line plots the ideal case. For both sets of experiments we used 512 samples, and changed the number of reducers. With more sampled points and thus more computations to do, (b) exhibits better scalability than (a). Note that the X axis of (b) starts at 16 reducers, not 1, and the Y axis of (b) starts at 100 seconds, not 1 second.

Finally, we conducted another set of experiments to see how the processing time varies to achieve different precision. Similar to what we did in Figure 3.18 to the sequential hybrid method, we specify different maximum errors and see how long the distributed method takes to get to the required precision. In this set of experiments we stuck to the DMKraken dataset (5 billion objects) using 128 reducers. Figure 3.21 illustrates the processing time of the distributed method, and we also copied the data points in Figure 3.18 to the same figure. The distributed method (5 billion objects) under current setting is about one magnitude faster than the sequential method (4.5 million objects) to achieve the same precision.

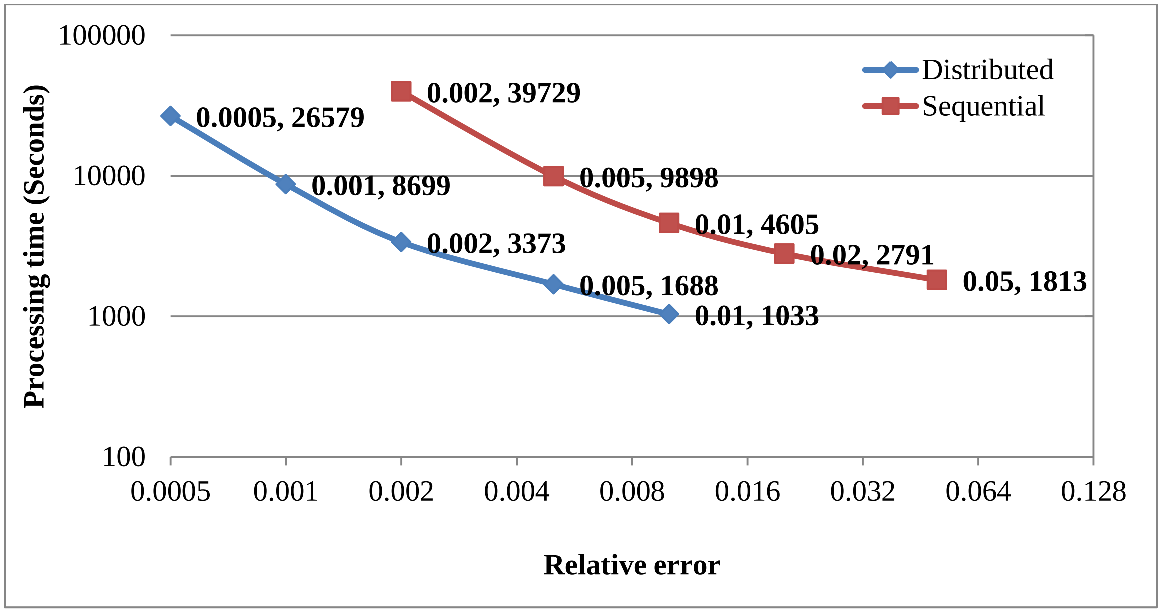


Figure 3.21: Running time of the distributed hybrid algorithm with different allowed errors. The five blue data points correspond to the maximum errors of 0.05%, 0.1%, 0.2%, 0.5% and 1%. All distributed experiments use 128 reducers and calculate distance ranges from $d_0 = 0.01$. Result from the sequential hybrid method (Figure 3.18) is also plotted for comparison.

Hadoop implementation on Amazon EC2

In the previous part we evaluated the performance of our distributed hybrid algorithm on the OpenCloud cluster. However, not every cosmologist has a ready-to-use Hadoop cluster. In this section, we answer the following question: How can we help cosmologists analyze large-scale data using Hadoop without a Hadoop cluster at hand?

Cloud computing is one possible answer for them. The idea behind cloud computing is that some Internet companies provide computers, networks, and other related computing resources, on which users can deploy and run their own applications. Users can pre-select from various operating systems and software, and they can also install new software themselves easily. Users can request the amount of resources as they wish, and it is flexible for them to add more resources or remove excessive resources in real time. Usually cloud computing resources are charged on a pay-per-use model[20].

Currently the most widely used cloud computing service is Amazon Elastic Compute Cloud (EC2), from which users can rent computer machines. Using EC2 it is convenient and flexible to setup a Hadoop cluster. Amazon Simple Storage Service (S3) – Amazon's online file storage service – can be used to store data for Hadoop jobs running on EC2.

We conducted the Hadoop experiments of our distributed hybrid technique on an Amazon EC2 cluster. Our goal is to evaluate current EC2 computer clusters comparing to the OpenCloud cluster from performance and economical point of view. We setup a Hadoop pre-installed Linux cluster on EC2, which consists of one master node and seven compute nodes. Each node has 1.7 GB memory and 5 EC2 compute units. Each EC2 compute unit provides the equivalent CPU capacity of a 1.0–1.2 GHz 2007 Opteron/Xeon processor, and costs $0.165 per hour. In other words, we spent $1.32 per hour to acquire an EC2 cluster with equivalently 35 compute cores in total.

We repeated the same experiments on our EC2 cluster. We used the same input dataset (1.1 billion objects) and same number of reducers (32). The performance of Hadoop jobs on EC2 was unstable, and it is roughly 60~70% slower than jobs on OpenCloud, which is reasonable since the EC2 cluster has slower CPU and network.

We also compare the pricing of the EC2 cluster to OpenCloud. The OpenCloud cluster has been in service for about three years (25,000 hours)[21]. Its building cost was around $400,000 for 64 nodes, so our usage worth about 2 dollars per hour at the time of the experiments[22]. Consequently, current cloud computing services are not essentially cheaper, although it provides a flexible solution, which is most helpful when users only want to test their code on a small cluster, and when they want to increase/decrease the computing resources frequently.

3.4.4 Other distributed implementations

Hadoop provides a neat solution of distributed computing to users, and they are especially powerful toward large datasets. Since MapReduce origins from the Internet industry, they are not yet very popular to the domain science community.

[20] To encourage the use of cloud services, most providers now have a free-tier for basic use. For example, Amazon EC2 provides 750 hours of Micro Instance (its low-end computer) usage for free per month. Amazon S3 offers 5 GB of standard storage for free.

[21] … as of the end of 2012.

[22] The Hadoop configuration on OpenCloud is 6 Map slots and 4 Reduce slots for each node. So using 32 reducers is equivalent to 8 compute nodes. If we simply spread the building cost of OpenCloud over the course of its lifespan (three years), then the cost of using 8 nodes is about 2$ per hour. Notice that here we ignore all other maintenance costs.

Similar in other science disciplines, the most widely used distributed computing frameworks for astrophysicists and cosmologists are still MPI [MPI, 1993]. Cosmologists use MPI to conduct large-scale simulations, so they usually possess a computer cluster with MPI installed, rather than Hadoop. Thus it is usually more convenient for them to use MPI to analyze large datasets [Dolence and Brunner, 2008] [Chhugani et al., 2012].

Additionally, recently some cosmologists also use General Purpose Graphic Processing Units, or GPUs[23], to conduct their simulations. GPU is distinct from CPU, and it provides a different angle for us to design parallel computing algorithms. Sometimes the speedup of GPU implementations is substantially higher than that of CPU.

So in this section, we compare the implementations of our hybrid algorithm using Hadoop, MPI, and GPU. We compare their characteristics and analyze their advantages and disadvantages. Although we only focus on one application here, we hope to provide a more general guidline to other similar lage data analysis scenarios in domain science.

For simplicity, in this section we mainly focus on the sampling technique on the naive calculation (Section 3.4.1 and Figure 3.22). We do not discuss the distributed computing on the *kd*-tree algorithm (researchers discussed the acceleration of *kd*-tree computation on supercomputers [Dolence and Brunner, 2008] [Chhugani et al., 2012]).

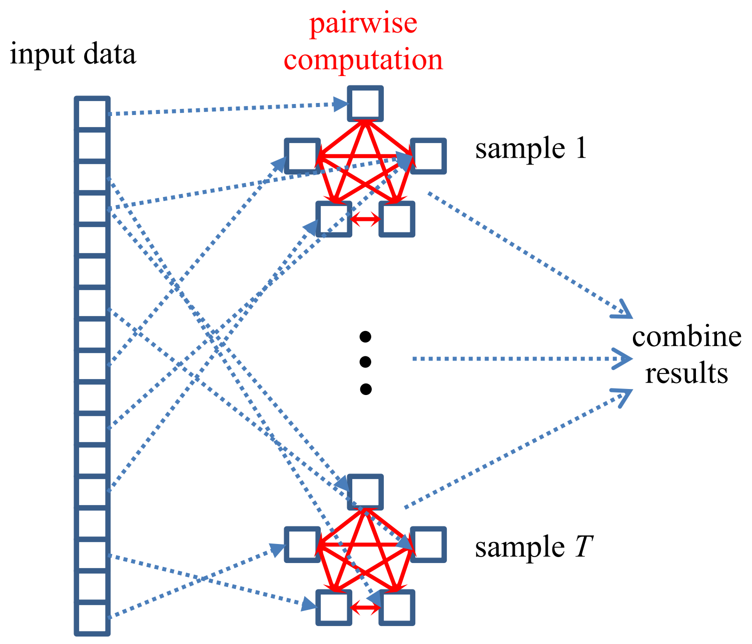


Figure 3.22: Illustration of the sampling-based naive calculation.

[23] The formal abbreviation should be "GPGPU". Since there is no ambiguity, in this section we use "GPU" for simplicity.

MPI implementation

Message Passing Interface (MPI) is a popular standard for distributed computing. It is the most widely used distributed computing technique to astronomers and other domain scientists. MPI framework allocates multiple compute machines which themselves communicate by sending/receiving messages. We implemented our sampling algorithm in MPI and show the pseudocode in Figure 3.23. We used the Portable Batch System[24] (PBS, [Bayucan *et al*., 1999]), resource management software for computer cluster, along with MPI to handle the job allocation on the cluster and other issues.

The MPI implementation is simple and efficient. MPI is considered to be the universal choice for high-throughput computation, and are deploying on the world's fastest supercomputers. However, it is not as good a framework as Hadoop to handle large data. We discuss its advantages and disadvantages in detail at the end of this section.

```
…
// MPI initializations
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Get_processor_name(name, &length);
…
randomly select $S$ objects $r_1, r_2, \ldots, r_S$ from $p_1, p_2, \ldots, p_N$
for $u = 1$ to $S - 1$ do
    for $v = u + 1$ to $S$ do
        find $i$ such that $d_{i-1} \le dist(r_u, r_v) < d_i$; $a_{it}$ ++
…
//start to pass results to master
if rank <> 0 then
    MPI_Send(…);
else
    for $i = 1$ to $numThread$ do
        MPI_Recv(…);
…
```

Figure 3.23: The MPI implementation of the distributed sampling algorithm

SSD and GPU implementation

In previous experiments, all the numbers we reported are the program execution time, and we did not consider the pre-processing time before running experiments. For example, in the Hadoop experiments, we did not count the time to transfer data to Hadoop File System; In the Amazon

[24] The MPI implementation is run on the Ferrari cluster of the Physics Department of Carnegie Mellon University (http://ferrari.phys.cmu.edu/ferrari/greet.html).

EC2 cloud computing experiments, we also did not count the time to put data into the Amazon Storage System; Similarly, in the MPI experiments, the time to partition data and transmit them to different compute node (if necessary) is not calculated as well. These inspire us to answer the following question: Assuming that we store a large dataset on a single disk, which implementation is the most convenient one if we consider the end-to-end cost?

In this regard we propose another implementation. The new solution consists of two procedures: First, we store the input data in the Solid-State Drives (SSDs) and use it to select samples from a large dataset; then we make use of the General Purpose Graphics Processing Units (GPUs) to conduct the all pairwise distances computation on selected samples. We show that the running time of this new solution is comparable to the previous Hadoop implementation, while the new approach requires almost zero data pre-processing works.

a. SSD

As we state in Section 3.4.1, the sample selection procedure simply traverses the whole dataset, so it has O(N) complexity where N is the number of objects in the dataset. Normally it is a fast procedure comparing to the all pairwise distances computation, but it is still costly if the dataset is very large or if the computation is less intensive (for example, if we only need a very rough estimate of the correlation function).

For instance, our 1.1 billion objects dataset occupies about 50 Gigabytes space on disk. Currently the data transfer rate of normal hard disk drives is about 100 Megabytes per second, which means that it takes around 500 seconds to go through the whole dataset and select all the samples.

But one does not need to traverse the whole dataset to select samples. Instead, we can pre-calculate the positions of all the sampled objects, and *seek* each of them from disk. To select T samples with S objects in each sample, $S \times T$ random seeks are needed. If the total number of seeks is relatively small comparing to the number of objects in the dataset, this seek-based sample selection may become faster than the traversal of the whole dataset.

Applying this seek-based sampling to our previous experiments, however, where $T = 30$ samples and $S = 120{,}000$ objects in each sample, results in a much slower process. Given the current random seek time of hard disk drives (12ms), this seek-based procedure takes 43,200 seconds, much worse than the traversal sampling (~500 seconds).

That is when we introduce one of the emerging storage systems – Solid-State Drives (SSDs). Different from the normal hard disk drives that use magnetic material to store data, solid-state drives use integrated circuit assembles, and it can achieve comparable read/write speed than that of hard disk drives. Moreover, the random access time of SSDs is significantly better, reaching as fast as 0.1ms comparing to 2.9~12ms of normal hard disk drives.

With the help of SSDs, the processing time of seek-based sampling is greatly improved: Using an SSD with 0.1ms random access time, retrieving samples on the previous setting takes about 360 seconds, better than the 500 seconds figure of the traversal solution. Although the improvement is not dramatic, it will be more obvious if the dataset is even bigger. For example, if the input dataset is 500GB and we still sample the same number of objects, then the seek-based sampling using an SSD still takes around 360 seconds, while the time to traverse the dataset will increase to 5,000 seconds.

This solution, though, would be the most efficient if we already store all the data in SSDs. Currently SSDs are still more expensive than hard disk drives[25]. However, the size of state-of-the-art SSDs can reach 2TB and it is becoming more and more popular. Many laptops now have already used SSDs as its primary storage media.

b. GPU

We have introduced a better technique to select samples, and now we focus on speeding up the following all pairwise distances computation on the samples, which has O(TS^2) complexity.

In this part we make use of General Purpose Graphics Processing Units (GPUs) to distribute the computation. Graphics processing units are first used in the computer graphics community, and recently the general purpose graphics processing units have been applied to conduct general computation tasks as well. Comparing to CPU, GPU especially excels at high-throughput computation. For example, the GPU module we have tested on, Tesla M2050 from NVIDIA, has a theoretical peak performance of 515 Gflops[26]. On many scenarios a GPU cluster can achieve substantially higher throughput than a CPU cluster.

The computing model of GPU is very different from CPU (Figure 3.24), and it is not equally suitable to process every distributed computing task. First, GPU modules do not have as much memory as CPU clusters. A Tesla M2050 has an aggregate 3GB memory for all its 448 cores to share. Secondly, GPU is most effective if each subtask follows the same instruction sequence. If the code sequence contains many branches, and different subtasks follow different control flows, then the parallelism of GPU modules will be seriously hindered.

[25] NAND flash SSDs cost approximately \$0.65 per GB. HDDs cost about \$0.05 per GB for 3.5 inch and \$0.10 per GB for 2.5 inch drives (data collected in early 2013).

[26] flops = FLoating-point Operations Per Second. A single-core 2.5GHz processor has a maximum of 10 billion flops = 10 Gflops.

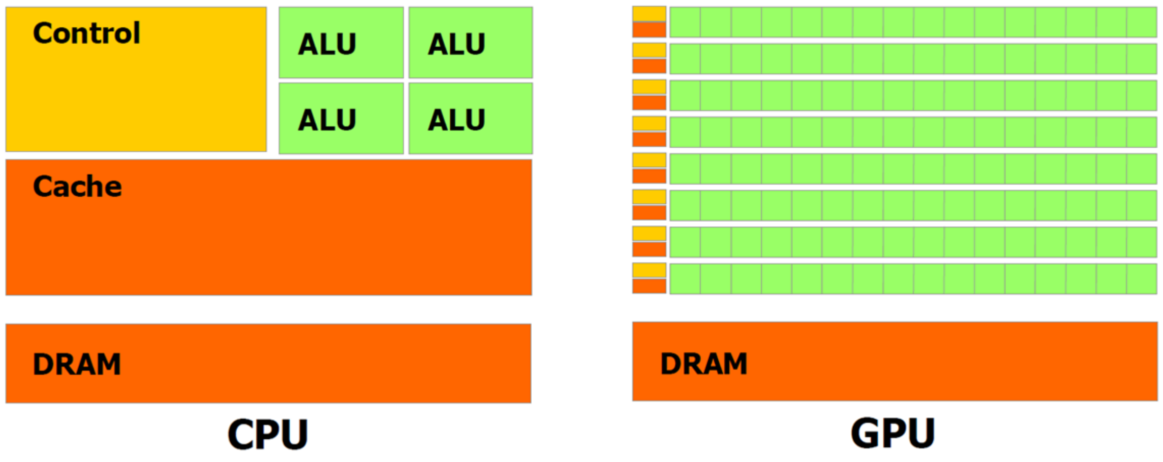


Figure 3.24: Illustration of the architecture differences between CPU and GPU. The figure is extracted from http://ixbtlabs.com/articles3/video/cuda-1-p1.html.

Our sampling-based all pairwise distances computation is suitable for GPU modules, since the samples do not require a large amount of extra memory, and the computation of each task shares the same control flow. Different from all our previous distributed computing solutions that distribute the computations by different samples, here we distribute the computation of a single sample to multiple GPU threads. We first load the objects in a sample into the global memory of GPU. Then, we let each GPU thread hold one object. Finally, each thread visits the global memory to fetch all other objects in the sample, and calculate the distances between the object it holds and other objects. This implementation is similar to a piece of previous work [Ponce et al., 2011].

We tested our GPU implementation on an Amazon EC2 machine with a GPU module. The GPU module is NVIDIA Tesla "Fermi" M2050, which contains 448 GPU cores (each at 1.15GHz), and its price is $2.1 per hour. Out of the two mainstream GPU programming models on market, we used NVIDIA's CUDA framework, which supports C/C++ extensions. The core function of our GPU implementation is illustrated in Figure 3.25.

Our GPU implementation achieved high level of parallelism. Comparing to the sequential code on GPU using only one GPU thread, the parallel code is 600X faster. Notice that although the GPU module contains only 448 cores, the actual speedup may exceed the total number of cores. However, since the GPU cores are slower than our CPU cores and due to other GPU costs like data loading, our parallel GPU implementation is about 100 times faster than our sequential CPU implementation.

To summarize, we used SSD and GPU together to develop another implementation. As Table 3.2 shows, the running time of the new implementation is comparable to the Hadoop implementation on OpenCloud using 128 cores. A more impressive property of this new technique is that it requires little pre-processing on the input dataset if it is already stored in an SSD.

```
void vecadd(double* data_x, double* data_y, double* data_z,
            int* result, int* final)
{
  // Some initializations are omitted.
  // Let this thread to hold one sampled points
  double x = data_x[index];
  double y = data_y[index];
  double z = data_z[index];

  // Thread visits global memory to get other sampled points
  for (int i = 0; i < OBJECT_NUM; i++) {
    double distSqr = (data_x[i] - x) * (data_x[i] - x) +
                     (data_y[i] - y) * (data_y[i] - y) +
                     (data_z[i] - z) * (data_z[i] - z);
    int bin_num = (int)(sqrt(distSqr) – small_epsilon);
    if (bin_num < INTERVAL_NUM)
      // local counters
      result[index * INTERVAL_NUM + bin_num]++;
    }

  // move the results of local counters to global counters
  for (int i = 0; i < INTERVAL_NUM; ++i) {
    // Use atomic function to avoid race condition
    atomicAdd(final + i, result[index * INTERVAL_NUM + i]);
  }
}
```

Figure 3.25: The GPU implementation of the distributed sampling algorithm.

Table 3.2: Hardware specification and processing time of our implementations on a dataset with 1.1 billion objects (50 Gigabytes). We issued 30 samples each with 120k sampled points. Asterisks indicate estimated figures.

	Sequential	Hadoop on OpenCloud	Hadoop on EC2	SSD+GPU
# nodes	1	8	7	1
# cores	2	64 (but only used 32 Reducers)	35	448
CPU frequency (GHz)	2.3	2.83	~1.1	1.15
Aggregate Memory (GB)	4	128	12	3
Price		\$2/hour	\$1.3/hour	\$2.1/hour
Sampling (seconds)	500	512^{*}		360^{*}
Computation (seconds)	6750	329^{*}		78
Total time (seconds)	7250	881	2500	438^{*}

Discussion

In this section, we introduce the implementations of our algorithm in popular distributed computing frameworks. All the related hardware specification and running times are summarized in Table 3.2. Each implementation (MPI, Hadoop, SSD/GPU) has its own merit, and it really depends on data, computing resources, and other factors for one to choose the most suitable framework for him:

MPI. MPI is the most widely used and deployed framework on existing computing resources possessed by domain scientists. Since currently most large-scale cosmological simulations are conducted under MPI, subsequent analysis on MPI is a natural and convenience choice. Also the performance is expected to be better than Hadoop Since Hadoop introduces extra overhead.

There are a couple of extra works that is needed to be handled by MPI users, though, when the input data is large. For example, to improve the I/O performance of MPI, users usually need to partition the input files and transmit them to different compute nodes (if it has not been so), so multiple processes can read them simultaneously. This processing needs to be executed every time when we change the computing environment (for instance, if we want to add more nodes to conduct experiments, and if a compute node fails), so it may be inflexible.

Hadoop. Hadoop is designed for the batch processing of large-scale data, and our algorithm, especially the sampling step, falls into that category. For this kind of tasks, Hadoop saves users a lot of efforts by implementing some useful features in the framework. From users' point of view, programmers just need to put all the input data into the Hadoop File System without worrying about the partition of input dataset, which is an issue for MPI when the input data is big. Hadoop further provides user-defined functions to conveniently partition input as they want. Several other useful properties are also provided, including the automatic handling of failed jobs and very slow jobs (*stragglers*), and so on.

Thus, comparing to MPI, the ideal case to use Hadoop is toward a large amount of data (so a MPI user may need to partition the data himself), and a large heterogeneous computer cluster that needs to be shared with others (So there might be more failures and/or some very slow tasks). Hadoop generally prevails in those situations.

SSD+GPU. We provided another implementation using SSD and GPU. SSD is suitable for random access (sampling), and GPU is compelling to work on "regular" computationally intense tasks (pairwise computation on samples). Currently both SSD and GPU are not as popular as normal hard disks and CPU respectively, but they are catching up. GPU especially have been deployed on some new supercomputers: Out of the 10 most powerful supercomputers in the world, three have already taken advantage of GPU acceleration. It is possible that more and more tasks running on CPU now will be shifted to GPU in the near future.

3.5 Current and Future Use of the Proposed Technique

Giving a large dataset, how the correlation functions are calculated in related works?

There are many previous works that calculate correlation functions toward a large dataset. [Dolence and Brunner, 2008] used MPI and OpenMP to parallelize the *kd*-tree calculation with up to 10^3 processors, and they evaluate the scalability of their code with a dataset with 10 million objects. [Chhugani et al., 2012] pushed further into this direction, optimizing the parallel *kd*-tree calculation from SIMD, thread, and node levels. Their largest experiment was running on 25,600 cores with 1.7 billion objects, which takes 5.3 hours to finish, although they only processed 10 distance ranges. [March et al., 2012] improved the original *kd*-tree technique, and tested on a dataset with one million objects. [Ponce et al., 2011] developed a GPU implementation of the naive technique and tested it up to 7 million objects.

What are the current uses of correlation functions by astrophysicists?

Astrophysicists calculate correlation functions on both astronomical sky surveys and cosmological simulations. In Section 3.1 we have introduced one of its applications in digital sky surveys (capturing the Baryon Acoustic Oscillations signals). In that context, the amount of computation is already enormous. For example, in order to get the results like Figure 3.1, correlation functions are needed to be calculated on both multi-million galaxies datasets and hundred-million random generated datasets, and the procedure needs to be repeated for a considerable number of times to get a statistically significant result. Using the MPI tree-based code, our collaborators indicated that it already takes several days on the Hopper supercomputer[27] in National Energy Research Scientific Computing Center.

It is not very expensive for astrophysicists to access those supercomputers as they do not need to pay actual money for use. However, there are still some restrictions and costs for using them: For one thing, astrophysicists need to write proposals for their potential use. For another thing, they have to make related data (for example, cosmological simulation) public. Even if one acquires the permission to use the supercomputers, in many cases he cannot use it 24/7 (The MassiveBlack simulation can only be run for a consecutive 24 hours each week). All these factors make supercomputers not a handy way to analyze very large datasets.

With respect to the analysis of large-scale cosmological simulations, the same argument also applies Again, since those simulations are usually conducted on supercomputers, a distributed correlation function code using MPI is a natural solution. However, there are surely needs when people cannot easily access supercomputers or want to run a preliminary smaller experiment. Moreover, current cosmological simulations produce much larger amount of data, which already achieved tens of billions objects as of now. For that amount of data, even the biggest

[27] Hopper with 131,072 cores and 221 TB memory: http://www.nersc.gov/users/computational-systems/hopper/

supercomputer cannot run the *kd*-tree algorithm in a short amount of time. Then our hybrid method provides an easy way to quickly come up with some estimation with controlled error.

To that end, we have already used our distributed hybrid technique to analyze one cosmological simulation. Figure 3.26 shows the correlation function results we conducted on a 3D density map [Dessup *et al.*, 2013]. For that dataset, the simulation cube was divided to 137 million (512^3) sub-cubes, and each sub-cube contains a density value which indicates the amount of matter in it. We successfully observed the expected BAO peak in this dataset using our distributed hybrid method. In the future, these density-map simulations with finer granularity will generate even larger datasets, where our method will be even more attractive.

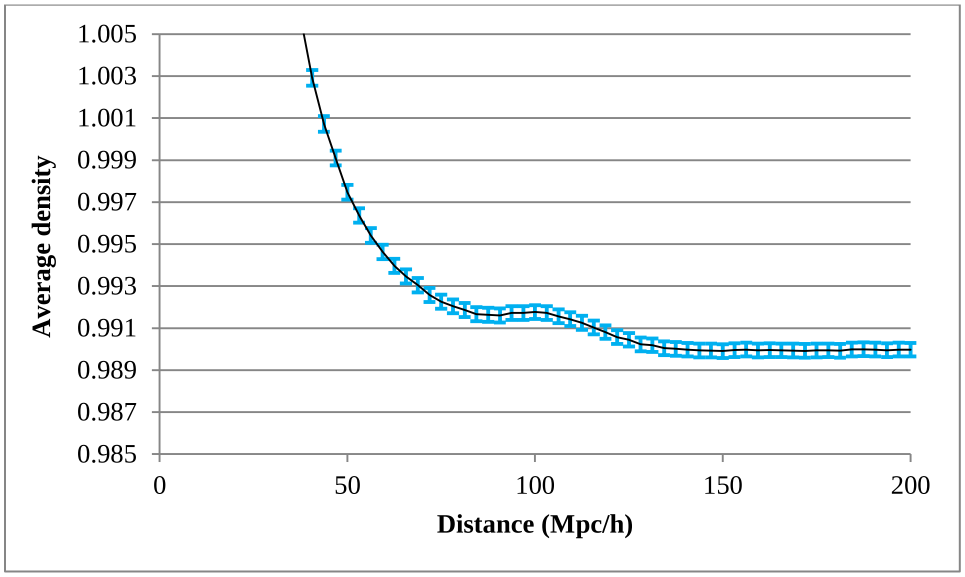


Figure 3.26: Correlation function results on a density map [Dessup *et al.*, 2013]. In this experiment we calculated the correlation function in a simulation cube with the side length of 1600Mpc/h. The cube is split into 512^3 sub-cubes. Each sub-cube is represented by a floating-point number indicating the overall matter density in it. Error bars in the figure represents 95% confidence interval. Similar to Figure 3.1, a peak arises around 100Mpc/h, indicating that more material is split by that distance.

How the error introduced in the sampling-based methods is perceived by astrophysicists?

Some astrophysicists reminded us to be alert of the use of subsampling. Their reason is that for both sky surveys and cosmological simulations, the amount of data collected/simulated are via precise calculation in order to achieve some statistics (e.g. eliminating counting error to acceptable level), since collecting/simulating excessive data is expensive. As a result, if we apply sampling on those datasets, some information will definitely be lost[28] (for example, we cannot use sampling to correctly calculate correlation functions at a very small distance). This effect, together with the fact that astrophysicists have spent million or billion dollars to acquire those data, may make them wary of our technique. Instead they would rather prefer spending more and use supercomputers to get exact results.

Despite the above argument, I want to point out that in some scenarios the results that astrophysicists generated already have a large error bar. Take Figure 3.1 as an example, where the errors shown in the graph can be as high as 100%[29]. It is reasonable that astrophysicists will try their best to not introduce any further error, but they will consider our technique if the introduced extra error is small, controllable, and above all, fast to come out.

3.6 Conclusion

We have presented a hybrid approximate algorithm for building the histogram of all pairwise distances between celestial objects, which allows fast accurate approximation for dataset with billions of objects. We also explore different distributed computing frameworks and analyze their properties. Our work has helped astronomers to study the property of expanding universe.

[28] ... unless we select a considerable number of samples or sampled objects. For example, if the dataset contains a billion objects, then 1000 samples with a million objects each would be more acceptable to astrophysicists, since in that way we will actually touch most of the input data.

[29] In Figure 3.1, the main source of error at small distances is shot-noise (counting error). The main source of errors at large distance is the boundary effect: Sky surveys are finite, and more uncertainty comes up when we consider the area outside the surveyed area. For a larger distance, there are more qualifying pairs that will stretch out of the surveyed area, thus leading to a higher variance.

Chapter 4 Indexing a Large-Scale Database of Astronomical Objects

In the next two chapters, we introduce our efforts to accelerate the solutions of two astrophysics problems. Although each problem also deals with billions of astronomical data entries, these problems exhibit different behavior, in that the information users query always touch a small proportion of the whole dataset. Consequentially, with careful data organization on disk to exploit data locality, the system we built can quickly answer user queries, using only one desktop machine. In this chapter, we introduce a low-level implementation to construct and query astronomical objects [Fu et al., 2012b]. In the next chapter, we use database techniques to handle the merge events of black holes.

4.1 Background

When astronomers analyze telescope images, they check whether the newly observed objects appear in available catalogs of known objects. Due to atmospheric and optical distortions, the positions of celestial objects in a telescope image may change slightly from observation to observation. As a result, the retrieval of exact catalog matches would be inadequate. Astronomers need to retrieve catalog objects that are *close* to the newly observed objects.

Straightforward matching algorithms, such as a linear search through a catalog, are too slow for analyzing a stream of newly incoming imaging data on a large catalog. We have developed a new technique for indexing massive catalogs and matching newly observed objects. On a standard desktop computer, it takes less than a second to match all objects in an image to a catalog with two billion objects.

4.2 Problem

Assume that we have a catalog of known celestial objects, and we also have obtained many new images. Typically, each image covers a square region of the sky, whose area is several square degrees. For example, the area of each image in the Sloan Digital Sky Survey is 1.5 square degrees. An image may contain from a few hundred to a few hundred thousand objects, depending on the image size and the telescope resolution.

The position of each object is represented by two values, called *right ascension* and *declination*, which define its equatorial coordinates (Figure 4.1). The right ascension, which is the celestial equivalent of longitude, ranges from 0.0 to 360.0 degrees; the declination, the celestial equivalent of latitude, ranges from −90.0 degrees (which represents the South Pole) to 90.0 degrees (the North Pole). We ignore the third spatial coordinate, that is, the distance from Earth to the object, since it is not directly observable and usually unknown during the initial stages of the image processing.

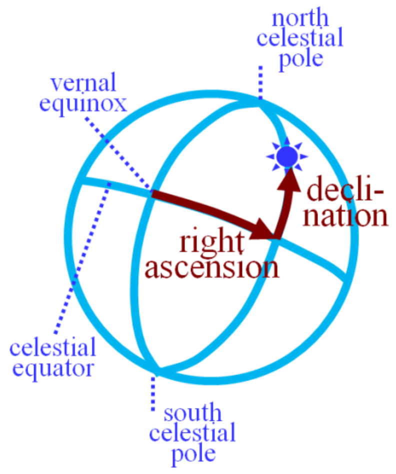


Figure 4.1: The representation of a celestial object in spherical coordinates.

Furthermore, astronomers also record the apparent magnitude of each object, which is the logarithm of its brightness. The apparent magnitude value serves as the “third coordinate”, which is used to identify the object along with its two spherical coordinates.

We assume that the edges of an image are parallel to the directions of right ascension and declination (Figure 4.2). For each image, we want to find the matches for all its objects. Specifically, for each object *p* in the image, we are looking for an object *q* in the catalog such that:

- Among all objects in the image, *p* is the nearest to *q*.
- Among all objects in the catalog, *q* is the nearest to *p*.
- The distance between *p* and *q* in the two-dimensional spherical coordinates is at most 1 arc second, which is 1/3600 of a degree. The value of 1 arc second reflects the maximal possible observation error due to atmospheric and optical distortions.
- The difference between the apparent magnitudes of *p* and *q* is smaller than a given constant *C*.

If the catalog contains an object *q* that satisfies all these constraints, we call it the *match* for *p*.

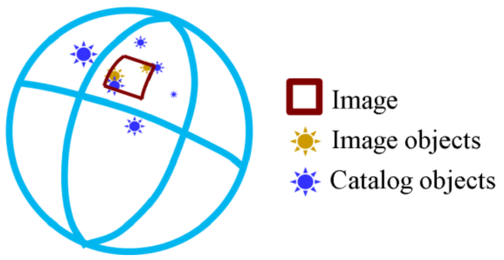


Figure 4.2: Example of the matching problem. There are two objects in the image, both of which have catalog matches.

4.3 Solution

The size of modern astronomical catalogs exceeds the memory of desktop computers. For example, suppose that each celestial object is stored as a 14−byte record: 4 bytes for its right ascension, 4 bytes for its declination, 2 bytes for its apparent magnitude, and 4 bytes with a pointer to the respective record with more information about the object in an external database. Then a catalog with one billion objects takes 14 Gigabytes, which would not fit the memory of a regular desktop. We therefore store the catalog on disk and load only parts relevant to processing a given image.

We first describe the organization of the catalog on disk. We then present the retrieval procedure that identifies the relevant part of the catalog and loads it into memory. Finally, we explain the in-memory matching.

4.3.1 Indexing

The indexing procedure arranges the catalog objects on disk, with the purpose to minimize the number of disk accesses during the retrieval of objects relevant to processing a given image.

We split the celestial sphere into multiple longitudinal strips, so for each image the retrieval procedure would only access a few of strips. These strips are parallel to the direction of right ascension, and each strip is exactly one degree wide (Figure 4.3). In Section 4.5, we will further discuss the choice of the specific strip width and the reason for setting it to one degree in the current system.

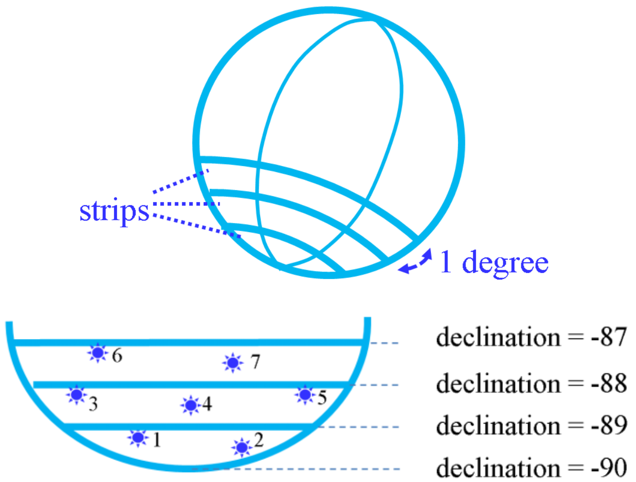


Figure 4.3: Indexing procedure. Top: The celestial sphere is divided into one-degree-wide strips. Bottom: In this example we assume that there are seven objects in the catalog, which are distributed among three strips. For each strip, the objects within the strip are sorted by their right ascension, and stored as a separate file.

The objects within a strip are stored as a separate file on disk, where the objects in the file are in sorted order by their right ascension.

Since the whole catalog usually does not fit in memory, we cannot process all data in one pass. The described procedure is implemented indirectly in two passes. During the first pass, we read all catalog objects, and put them to the corresponding files without sorting. In the second pass, we load each file into memory, sort its objects, and store the file in sorted order.

If we split the celestial sphere into S strips, and N catalog objects are about uniformly distributed among those strips, then the time complexity of this procedure is $O(N \cdot \lg(N/S))$, and the number of disk accesses during its execution is $O(N)$.

4.3.2 Retrieval

Given an image, we need to retrieve the catalog objects that may potentially match the image objects. Since a matching catalog object must be within 1 arc second from a newly observed object, the possible matches for an image may be located at most 1 arc second away from the image area. We retrieve all the catalog objects that are near the axis-aligned minimum bounding box of the image, which is the smallest rectangle that covers all image objects, with sides parallel to the directions of right ascension and declination (Figure 4.4).

We calculate the axis-aligned minimum bounding box of the image, and extend it by one arc second on all sides.

We then retrieve all catalog objects inside the extended bounding box from the catalog files, which is done in three steps. The first step is to locate the strips that overlap the extended bounding box; the second is binary search within each respective file, which identifies all catalog objects whose right ascension value falls inside the extended bounding box; the third is to load all the related objects to memory.

To analyze the time complexity of the retrieval procedure, we again assume that the celestial sphere is split into S strips, and N catalog objects are about uniformly distributed among those strips. We further assume that the cost of each disk access is c_1 in average, and the cost of loading each object from disk to memory is c_2. If the extended bounding box of the image covers s strips and contains n catalog objects, then it takes $O(s \cdot \lg(N/S) \cdot c_1)$ to use binary search to locate the extended bounding box of image on the catalog, and $O(n \cdot c2)$ to load related catalog objects to memory.

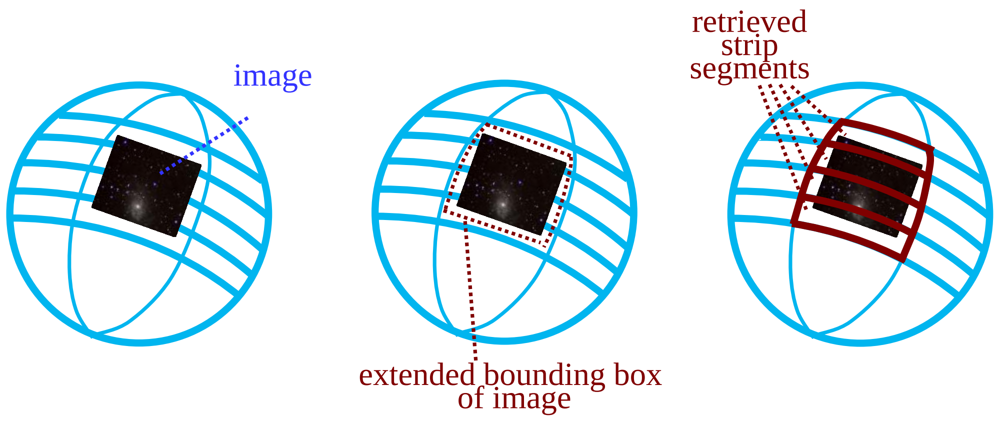


Figure 4.4: Retrieval procedure. Given an image, its axis-aligned minimum bounding box is calculated and extended by 1 arc second on all sides. Then, the part of the catalog that covers the extended bounding box of the image (that is, the retrieved strip segments in the figure) is loaded into memory.

4.3.3 Matching

The last main step is to identify matches among the objects loaded into memory. If the image contains M objects, and L objects are extracted from the catalog in the retrieval procedure, a naive matching algorithm would take $O(M \cdot L)$ time, which is impractically slow for real-time matching of images against large-scale catalogs. We next provide a more efficient technique.

The developed approach is similar to the "strip" idea used in the indexing procedure. Specifically, we further subdivide the retrieved strips into thinner sub-strips. These sub-strips are also parallel to the direction of right ascension, and each sub-strip is exactly one arc second wide. The objects within each sub-strip are again sorted by their right ascension, which allows the use of binary search for identifying close catalog objects for each image object.

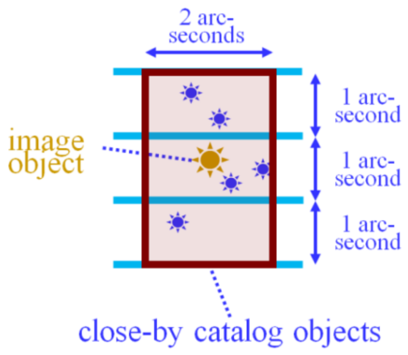


Figure 4.5: Illustration of the matching procedure. For each image object, we extract all catalog objects that are at most one arc second away, and then calculate their exact distances to the image object.

For each image object, since its match can be at most one arc second away, we consider only the catalog objects in its three nearby sub-strips, that is, its own sub-strip and the two adjacent sub-strips. Among these three sub-strips, we use binary searches to locate the image objects that are at most 1 arc second away. We illustrated the matching procedure in Figure 4.5 and give pseudocode in Figure 4.6.

4.4 Experiments

We have evaluated the running time of each procedure described in Section 4.3: the indexing procedure, the retrieval procedure, and the matching procedure. We have used synthetic catalog data with a random uniform distribution of objects across the sky.

We have run the experiments on a desktop computer with Pentium Xeon 2.8 GHz dual quad core, 16GB memory, and 7200 RPM 160 GB disk. The described algorithms are implemented in Java 1.6. All data points in the summary graphs are the mean of three runs with system caches flushed between runs.

4.4.1 Indexing

We show the running time of the indexing procedure in Figure 4.7. As discussed in Section 4.3.1, its time complexity is $O(N \cdot \lg(N/S))$ where N is the number of objects in the catalog and S is the number of strips, which matches the observed empirical results. Specifically, the running time is about $1.85 \cdot 10^{-7} \cdot N \cdot \lg(N/S)$ seconds. It takes about 6,000 seconds (1.7 hours) to index a catalog of two billion objects. Note that this procedure has to be run only once for the given catalog, and occasionally rerun later after updates of the overall catalog.

Input:

$q_1, q_2, \ldots, q_M$: Image objects.

$c_1, c_2, \ldots, c_L$: Extracted catalog objects that are possible to match the image objects. This is the output of the retrieval procedure.

Output: Possible matches for image objects.

BestI[1, 2, …, M] and DistanceI[1, 2, …, M]: BestI[i] stores the index of the closest catalog objects to q_i. The distance between q_i and $c_{\mathrm{BestI}[i]}$ is stored in DistanceI[i].

BestC[1, 2, …, L] and DistanceC[1, 2, …, L]: BestC[j] stores the index of the closest image objects to c_j. The distance between c_j and $q_{\mathrm{BestC}[j]}$ is stored in DistanceC[j].

```
for $i$ = 1 to $M$ do DistanceI[$i$] = MAX; BestI[$i$] = 0
for $j$ = 1 to $L$ do DistanceC[$j$] = MAX, BestC[$j$] = 0
Split the extracted catalog area into sub-strips, and sort the catalog objects in each sub-strip
for $i$ = 1 to $M$ do
    // Find possible match for $q_i$
    Retrieve the nearby 3 sub-strips of $q_i$, and conduct binary searches in the three sub-strips to
    retrieve the catalog objects $r_1, r_2, \ldots, r_K$ that are at most 1 arc second away from $q_i$ (Figure 4.5).
    for $k$ = 1 to $K$ do
        //Assume $r_k = c_j$. Compute the distance between $q_i$ and $c_j$
        $d$ = distance($q_i$, $c_j$)
        if $d$ < DistanceI[$i$] then DistanceI[$i$] = $d$; BestI[$i$] = $j$
        if $d$ < DistanceC[$j$] then DistanceC[$j$] = $d$; BestC[$j$] = $i$
for $i$ = 1 to $M$ do
    // Output the possible match for $q_i$
    $match$ = BestI[$i$] // $c_{match}$ is the closest catalog object to $q_i$
    if $match$ > 0 && BestC[$match$] == $i$ then
        // $q_i$ is also the closest to $c_{match}$
        OutputMatch($q_i$, $c_{match}$)
```

Figure 4.6: Matching algorithm.

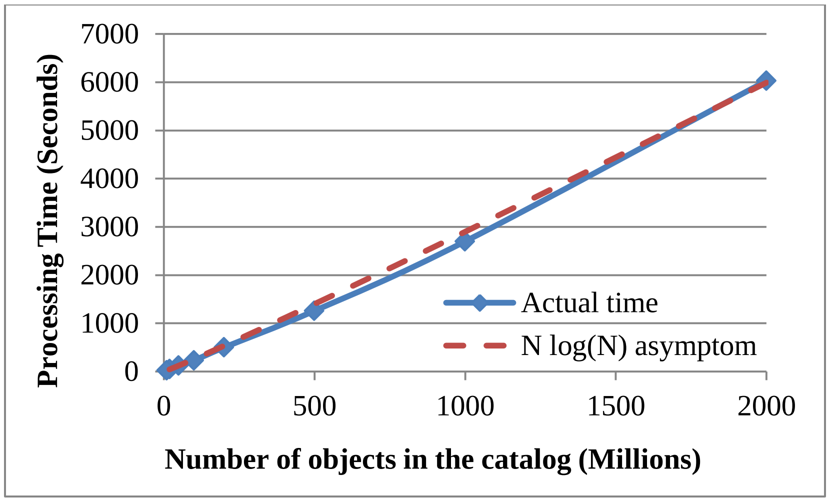


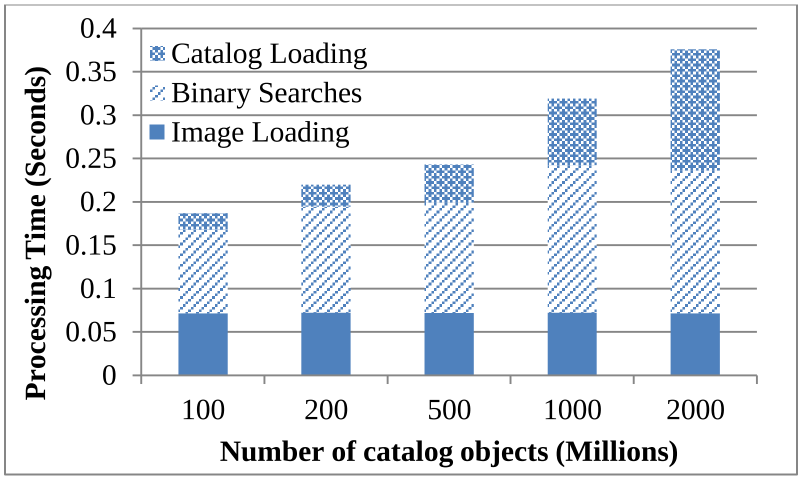
Figure 4.7: Running time of the indexing procedure.

4.4.2 Retrieval

The two main factors affecting the retrieval time are the number of objects in the catalog, and the area of the image. On the other hand, the number of objects in the image does not affect the retrieval time.

We breakdown the retrieval time into three parts: the time to load image objects from a file (image loading), the aggregate time on binary searches when we identify all the catalog objects falling in the bounding box of the image (binary searches), and the time to load the catalog objects to memory (catalog loading). Specifically, using the notations in Section 4.3.2, the retrieval time is about $0.0028 \cdot \lg(N / S) \cdot s + 9 \cdot 10^{-7} \cdot n$ seconds.

The top graph in Figure 4.8 shows the dependency of the retrieval time on the number of objects in the catalog. The bottom graph in Figure 4.8 shows the relationship between the retrieval time and the side length of the square image. It takes about 0.5 second for a large image (3.5×3.5 degrees) and a large catalog (2 billion objects).



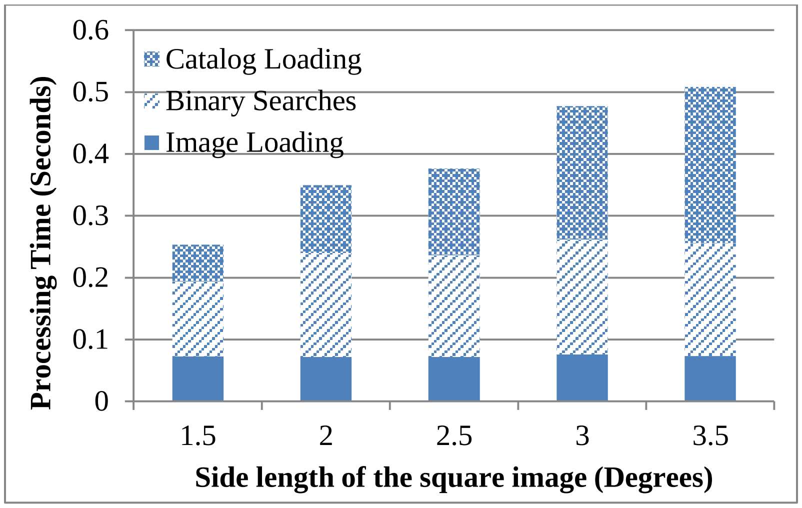
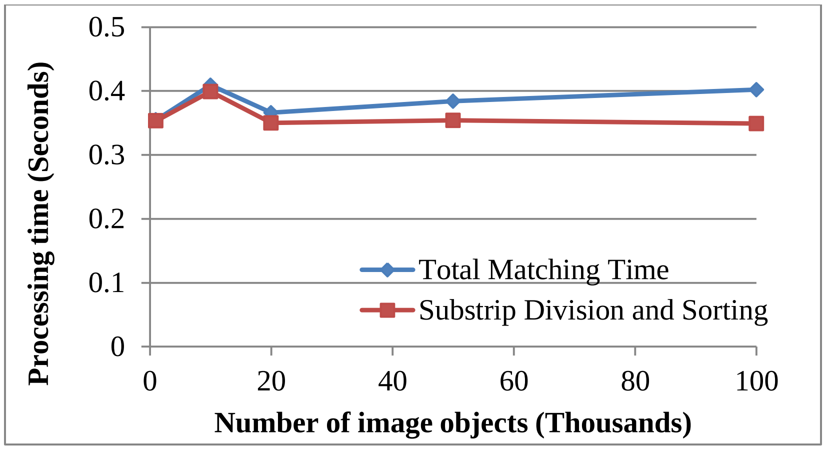


Figure 4.8: Retrieval time. We use 2.5 × 2.5 degree images and a catalog with 2 billion objects as the baseline. We show the dependency of the running time on the catalog size for 2.5 × 2.5 degree images (top), and the dependency of the time on the side length of the square image for a catalog with 2 billion objects (bottom).

4.4.3 Matching

We have evaluated the dependency of the matching time on two parameters: the number of objects in the image (top of Figure 4.9), and the side length of the image (bottom of Figure 4.9). The running time is under 0.6 second in all cases. Most of the running time is spent on the sub-strip division and sorting.



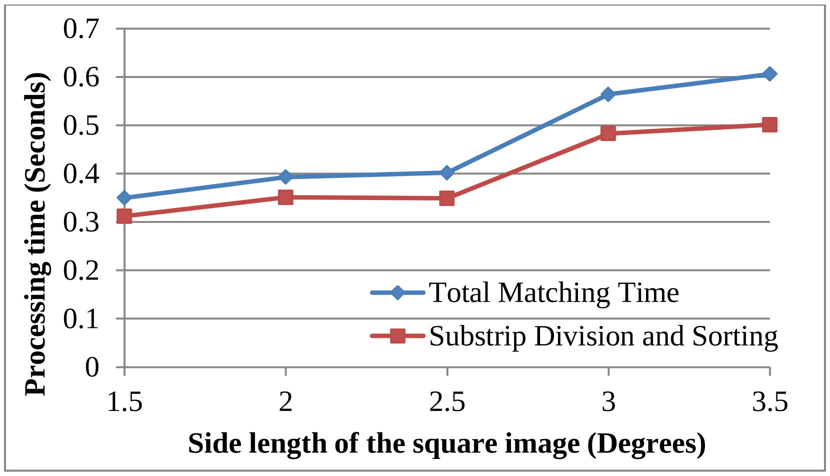


Figure 4.9: Matching time. The baseline experiment is with a catalog of 2 billion objects, and a 2.5 × 2.5 degree image with 100 thousands objects. We show the dependency of the matching time on the number of image objects (top), and the side length of the image (bottom).

4.5 Discussion

We next discuss some issues related to the problem and the proposed approach.

4.5.1 Indexing Method

We use a simple method of splitting the celestial sphere into strips. There are other standard ways to divide the sphere into multiple parts and index them, such as the Hierarchical Triangular Mesh [Szalay et al., 2005], that some datasets may benefit from.

4.5.2 Updating Catalog

We may need to perform occasional updates of the catalog to add newly discovered objects or delete some of the old objects. The described technique provides an efficient solution for such updates. To insert and delete celestial objects, we load each related strip into memory, make insertions and deletions, re-sort each strip, and store the updated strips on disk.

4.5.3 Width of Strips

We have set the strip width to one degree. We now explain the reason behind this choice.

The main related tradeoff is that, if the strips are too wide, we retrieve many catalog objects that do not match objects in the image; on the other side, if the strips are too narrow, we need to open too many files to conduct binary searches, thus incurring high disk-access costs. In Figure 4.10, we show the dependency of the retrieval time on the strip width, which confirms that the use of 1-degree strips leads to the fastest retrieval of 2.5 $\times$ 2.5 degree images.

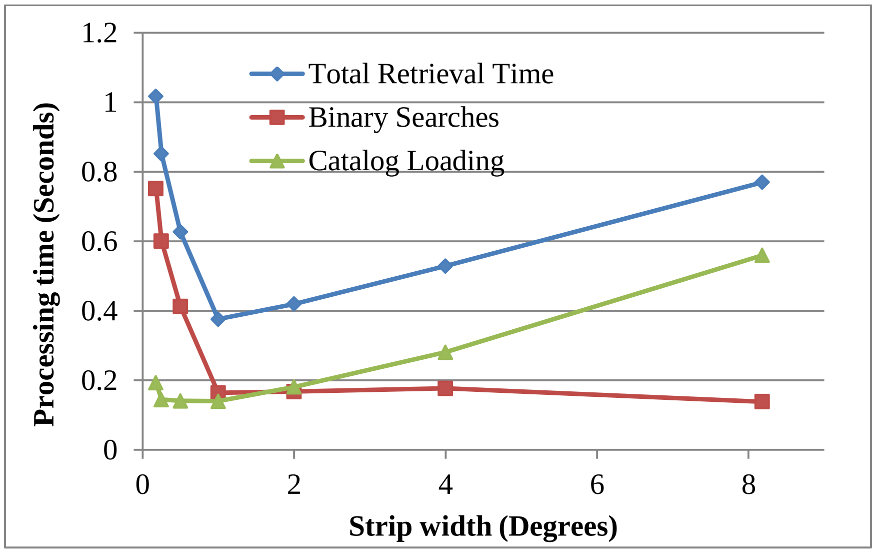


Figure 4.10: Impact of the strip width on the retrieval time for 2.5 $\times$ 2.5 degree images.

4.6 Related Work

The described matching problem can be formulated as a range query: Find all catalog objects that are within 1 arc second from an image object. Space-partitioning data structures, such as R-tree [Guttman, 1984] and *kd*-tree [Bentley 1975] and others [Wicenec and Albrecht, 1998] can be used for such queries. However, the retrieval based on these structures is significantly slower than the described technique, especially when the catalog is too big to store in memory.

Scientists from database community have also developed tools for organizing astronomical data [Baruffolo, 1999]. Although traditional database technologies are difficult to use for efficient handling large astronomical data [Pirenne and Ochsenbein, 1991], the emergence of new database technologies, namely, Object Data Management Systems, and Object-Relational Database Management Systems, now provides spatial indexes for astronomical data, and already used by researchers [Chilingarian *et al*., 2004].

For example, two widely-used open source databases, MySQL [30] and PostgreSQL [Douglas and Douglas, 2003], support R-tree spatial indexes ([Rigaux *et al*., 2002] and PostGIS[31]). It is a promising direction since the database approach is straightforward and easy to implement, and it provides more functions than a stand-alone implementation as our technique.

However, the scalability of current off-the-shelf solutions is severely limited. The experiments show that it takes over two thousand seconds (33 minutes) to load 50 million objects into the database and create the spatial index. Furthermore, the database approach requires 5 GB to store those 50 million objects, while our solution takes only 600 MB on the same amount of data.

4.7 Conclusion

If we have a catalog with billions of objects, how do we index it to support fast matching operations? We have tackled this problem on a standard desktop computer. We propose a way to organize the catalog on disk and dynamically loading relevant parts of the catalog into memory, thus achieving good performance. Experiments on a catalog with 2 billion objects show that building a catalog takes less than 2 hours, and the retrieval and matching for an astronomical image takes less than a second.

[30] http://dev.mysql.com/doc/refman/5.1/en/index.html
[31] http://postgis.refractions.net/

Chapter 5 Building and Querying Black Hole Merger Trees via Database

Large-scale N-body simulations play an important role in advancing our understanding of the formation and evolution of large structures in the universe. These computations require a large number of particles, in the order of 10–100 of billions, to realistically model phenomena such as the formation of galaxies. Among these particles, black holes play a dominant role on the formation of these structures. The properties of the black holes need to be assembled in merger tree histories to model the process where two or more black holes merge to form a larger one.

In the past, these analyses have been carried out with custom approaches that no longer scale to the size of black hole datasets produced by current cosmological simulations. We present algorithms and strategies to store, in relational databases, a forest of black hole merger trees [López *et al*., 2011]. We implemented this approach and present results with datasets containing 0.5 billion time series records belonging to over 2 million black holes. We demonstrate that this is a feasible approach to support interactive analysis and enables flexible exploration of black hole forest datasets. Our systems are deployed and utilized by astrophysicists in several of their projects.

5.1 Introduction

The analysis of simulation-produced black hole datasets is vital to advance our understanding of the effect that black holes have in the formation and evolution of large-scale structures in the universe. Increasingly larger and more detailed cosmological simulations are being developed and carried out to increase the statistical significance of the generated particle and black hole datasets. These higher resolution datasets are key to gain insight on the evolution of massive black holes.

The simulations store the data in a format that is not readily searchable or easy to analyze. Purpose-specific custom tools have often been preferred over standard relational database management systems (RDBMS) for the analysis of datasets in computational sciences. The assumption has been that the overhead incurred by the database will be prohibitive. Previous studies of black holes have used custom tools. However, this approach is inflexible: The tools often need to be re-developed for carrying out new studies and answering new questions. We recently faced this challenge when the existing tools could not handle the data sizes produced by our recent simulations.

As part of our goal of reducing the time to science, we decided to leverage RDBMS implementations to perform the analysis of black hole datasets. This approach enables fast, easy and flexible data analysis. A major benefit of the database approach is that now the astrophysicists are able to interactively ask ad-hoc questions about the data and test hypotheses by writing relatively simple queries and processing scripts. We present:

- A set of algorithms and approaches for processing, building and querying black hole merger tree datasets.
- A compact database representation of the merger trees.
- An evaluation of the feasibility and relative performance of the presented approaches.

Our evaluation suggests that it is feasible to support the analysis of current black hole datasets using a database approach. The rest of this chapter is structured as follows: In Section 5.2 we describe our motivating science application, the analysis of black hole datasets. We show background information and related work in Section 5.3, and various approaches for processing black hole datasets in Section 5.4. The evaluation will be presented in Section 5.5. We review the deployment and utilization of our systems, and discuss future work in Section 5.6. Finally we conclude in Section 5.7.

5.2 Black Holes in the Study of Cosmological Simulations

Black holes play an important role in the evolution of the universe. Astrophysicists have found that supermassive black holes exist at the center of most galaxies, including our own, and there are strong correlations between the black hole and its host galaxy, which indicates that black holes have a significant impact on how galaxies evolve.

Moreover, since black holes are usually very bright and thus easily detected, they act as *tracers* for astrophysicists to study galaxies even when the galaxies are too faint to be observed. Study how black holes cluster and how they grow provide valuable insights for astrophysicists to understand the process by which large-scale structures, such as galaxies, groups and clusters of galaxies, are organized in the universe. Structure formation and evolution in cosmology encompasses the description of the rich hierarchy of structures in the universe, from individual galaxies and groups to clusters of galaxies, and up to the largest scale filaments along which smaller structures align.

To study the processes, cosmological numerical simulations that cover a vast dynamic range of spatial and time scales are being developed. These need to include the effect of gravitational fields generated by superclusters of galaxies on the formation of galaxies, which in turn harbor gas that cools and makes stars and is being funneled into supermassive black holes the size of the solar system.

There are two conflicting requirements that make the study of hierarchical structure formation extremely challenging. To have a statistically significant representation of all structure in the universe, the studied volume needs to be large. However, the particle mass needs to be relatively small to adequately resolve the appropriate physics and the scale of the structures that emerge. This implies a need for an extremely large number of particles in the simulation, requiring in principle a dynamic range of 10^{10} or more.

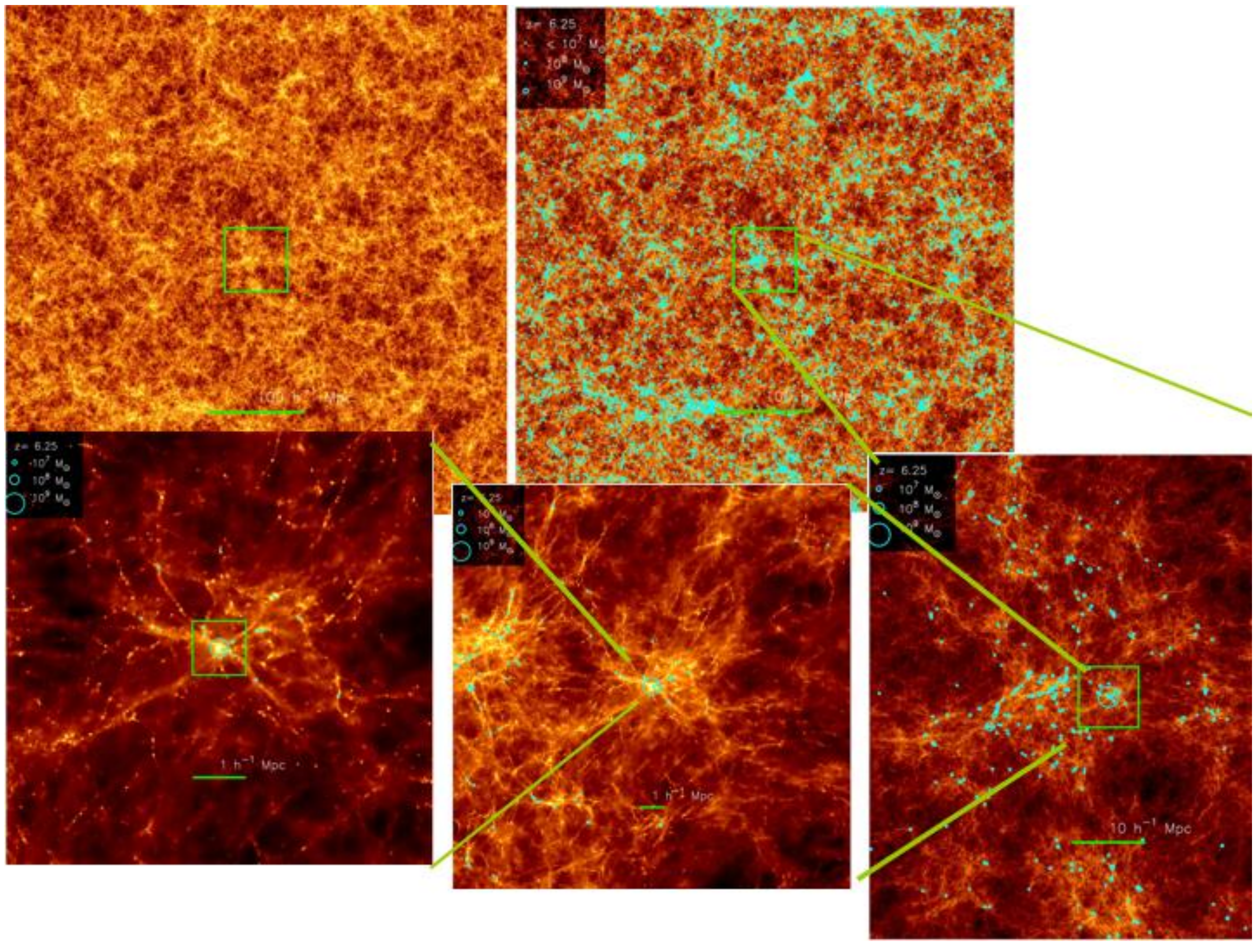


Figure 5.1: Visualizations of a large-scale dark matter simulation carried out by the researchers of the McWilliams Center for Cosmology. The top two pictures show slices of the simulation volume. The black holes are shown as bright dots (light color). Successive zoom factors (10X) of the highlighted boxes are shown in the bottom 3 frames.

Scientists at the CMU McWilliams Center for Cosmology and collaborating institutions use the parallel program GADGET-3 [Springel, 2005] to carry out large-scale cosmology simulations, MassiveBlack, on supercomputers with 100,000 CPU cores. A visualization of the result of these computations is shown in Figure 5.1. The simulations evolve an initial realization of a Lambda Cold Dark Matter (ΛCDM) cosmology over cosmological timescales, incorporating dark matter, gas, stars and black holes. The gravitational forces are calculated with a hybrid approach, named TreePM, which combines a hierarchical tree algorithm for short range forces with a particle-mesh algorithm for long-range forces. The gas is modeled using Smoothed Particle Hydrodynamics (SPH) which uses the Lagrangian method of discretizing mass into a set of particles with adaptive smoothing lengths [Springel and Hernquist, 2002], naturally providing a varying resolution from the densest regions at the center of massive galaxy halos to the diffuse voids between halos. Sub-resolution models are used to model star formation and supernova feedback (using the multi-phase model [Springel and Hernquist, 2003]) as well as black hole formation, accretion, and feedback [Springel *et al.*, 2005] [Di Matteo *et al.*, 2005].

Analysis of Black Hole Datasets. With the current simulation capabilities, we are now in a position to make predictions about the mass distribution in the inner regions of galaxies. In particular, recent observations imply that black holes with billion solar masses are already assembled and seen as the first quasars and large galaxies when the universe is only 800 million years old. As these objects are likely to occur in extremely rare high density peaks in the early universe, large computational volumes are needed to study them. An aim of high-resolution, large-volume simulations and associated analysis of the produced datasets is to explain the formation of these objects, which is a major outstanding problem in structure development.

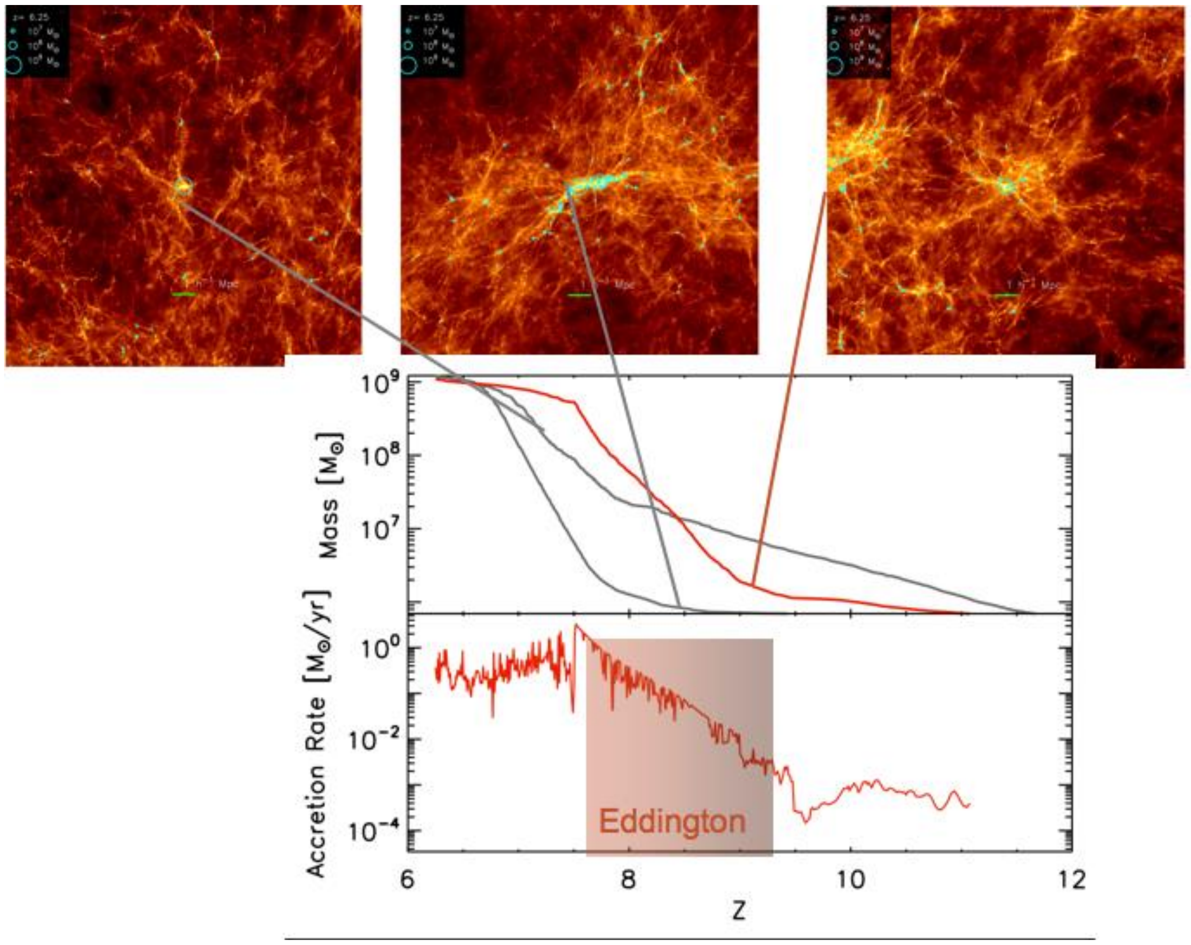


Figure 5.2: Sample black holes. This figure shows the gas distribution around two of the largest black holes in a snapshot from a recent simulation. The respective light curves for these black holes are shown in the plot, as well as the accretion rate history for the most massive one.

There are two general types of questions which we typically want to address using the black hole datasets. The first one is to investigate the black hole populations which exist at a specific time. These queries on the overall population of black holes have been used to study a wide variety of black hole properties, such as the number and density of black holes as a function of mass [Di Matteo et al., 2008] or luminosity [DeGraf *et al*., 2010a], the clustering properties of different populations of black holes [Degraf *et al*., 2010b], and the correlation between black holes and the galaxies in which they are found [Colberg and Di Matteo, 2008] [Di Matteo et al., 2008]. These questions require queries based on a specific redshift (i.e., simulation time), often selecting a subset of the black holes at that time based on their mass and accretion rate. The second type of questions requires looking at the detailed growth history of individual black holes. An example is shown in Figure 5.2. These histories can help us understand how black holes grow, the relative importance of black hole mergers vs. gas accretion, how the black hole luminosity varies with time, and how they depend on their surrounding environment [Colberg and Di Matteo, 2008] [Di Matteo et al., 2008]. At a high-level these analyses involve deriving, examining and correlating aggregate statistics of the black hole datasets obtained from the simulations. In particular, it requires obtaining the time history for a black hole including the information about which black holes merged.

Black Hole Datasets. Our cosmological simulations produce three types of datasets: *snapshots*, *group membership* and *black holes*. The snapshots contain complete information for all the particles in the simulation at a given time step. Snapshots have a high cost in terms of both time and storage space. For example, the snapshots of the latest simulations are each three terabytes in size. Only a few snapshots are stored per simulation (e.g., 30). The output frequency varies throughout the simulation, with snapshots being written more frequently at later times when the simulation exhibits a highly non-linear behavior. The group membership files contain the statistics of cluster structures, such as dark matter halos, as well as the group membership information for particles in the snapshots.

In addition, the black hole data is written to a separate set of text files at a much higher time-resolution to preserve the black hole information. This is feasible since the black holes only make up a relatively small fraction of the total number of particles in the simulation. Each of the compute hosts participating in the simulation produces a file containing the data associated with the black holes residing in that host. The files contain one of the following three types of records per line: black hole properties, near mergers, and merger events:

(1) The *black hole properties* are stored when they are re-calculated for a new time step. The stored properties include the id, simulation time, mass, accretion rate, position, velocity relative to the surrounding gas, local gas density, local sound speed, and local gas velocity. Records of this type make the most of the black hole output.

(2) *Near merger* records are produced when a pair of black holes are close enough to initiate a merger check, but are moving too rapidly past one another. The output record contains the simulation time, the id of each black hole, the velocity of the black holes relative to one another, and the local sound speed of the gas.

(3) *Merger event* records are stored when a pair of black holes merge with one another. The output record contains the ids and masses of the two black holes and the time at which

the merger occurred. A black hole merger tree comprises the set of merger event records along with the detailed property records for the black holes involved in the mergers.

5.3 Background and Related Work

The scientific computing community has developed several file formats, such as FITS [Wells et al., 1981], NetCDF [Rew and Davis, 1990] and HDF5 [Folk *et al.*, 1999], for storing datasets produced by numerical simulation, atmospheric and astronomy observations. These formats support efficient storage of the dense array-oriented data. However, these formats have limited mechanisms for indexing objects based on their values and fast retrieval of matches to specific queries. Specialized applications, e.g., in seismology [Tu et al., 2002] [Schlosser et al., 2008] and geographical information systems, have used indexing structures such as B-trees [Bayer and McCreight, 1970], R-trees [Guttman, 1984], octrees [Samet, 1990] [Tu et al., 2003] and *kd*-trees [Lee and Wong, 1977]. The use of RDBMSs for array oriented data, in systems such as rasdaman [Baumann *et al.*, 1997], is an emergent area of active research.

Database techniques have been adopted to manage and analyze datasets in a variety of science fields such as neuroscience [Lependu *et al.*, 2008], medical imaging [Cohen and Guzman, 2006], bioinformatics [Xu et al., 2009] and seismology [Yang *et al.*, 2007]. In astronomy, RDBMSs have been used to manage the catalogs of digital telescope sky surveys [Brunner *et al.*, 1999] [Ivanova *et al.*, 2007]. For example, the Sloan Sky Digital Survey (SSDS) has collected more than 300 million celestial objects to date [Abazajian *et al.*, 2009]. Database techniques have been used in observational astronomy datasets to perform spatial clustering [Szalay et al., 2002] and anomaly detection [Kaustav et al., 2008] among others. Various research groups have used distributed computing frameworks, such as MapReduce [Dean and Ghemawat, 2004], Pig [Olston *et al.*, 2008] and Dryad [Isard et al., 2007] for clustering and analysis of massive astrophysics simulations [Leobman *et al.*, 2009] [Fu et al., 2010] [Kwon *et al.*, 2010].

RDBMSs have not been as widely used for the analysis of cosmological simulations, in part due to the challenge posed by the massive multi-terabyte datasets generated by these simulations. The German Astrophysical Virtual Observatory (GAVO) has led in this aspect by storing the Millenium Run dataset in an RDBMS and enabling queries to the database through a web interface [Lemson and Springel, 2006]. GAVO researchers proposed a database representation for querying the merger trees of galactic halos.

In our collaboration with astrophysicists, we are using RDBMSs to support the analysis of cosmological simulation datasets. We present various techniques for building and querying the merger trees of black holes. We present a modified database representation for these trees that is compact and addresses the particular requirements of the black hole datasets produced by cosmological simulations. Union-Find is an algorithm that has near-linear running time and can be used to find the connected components of a graph [Galler and Fischer, 1964] [Tarjan, 1975]. Various approaches presented here are based on Union-Find, with adaptations to handle the specifics of the merger events representation produced by cosmological simulations. The non-RDBMS approach used to analyze previous black holes datasets is described below.

5.3.1 Non-DB Approach: Custom Binary Format

The analyses of black hole datasets produced by previous simulations used custom tools that employed a storage layout specifically designed for this purpose. The process comprises two steps: first generating a binary file containing all the black hole information, and then using tools to extract desired black hole data from that file. The binary file consists of a series of arrays, starting with the general properties at each timestep, including the number of black holes active at any given timestep, a list of those black holes, and what the previous/next timesteps are. Then there are a series of arrays, each containing one element for every timestep of every black hole. Most of these arrays contain the basic properties of the black holes as output by the simulation (mass, accretion rate, position, etc.) but there is also an array generated with the binary file, containing the array location of the black hole's progenitor at the previous timestep (or progenitors if the previous time step contained a merger with another black hole). This array acts as a form of indexing which can be used when querying the history of a given black hole by pointing directly to the previous timestep. However, it is only helpful for this specific query, and is inefficient as it does not exploit locality of reference.

The system was quite inflexible, as the queries were hardcoded into the program, only allowing three specific requests: (a) extracting the black hole properties at a specified time, (b) extracting the complete histories of all the black holes found in the merger tree of a given black hole, and (c) extracting the most-massive progenitor history for a given black hole (i.e. the history of the most massive black hole involved in each merger event). Because these queries are hardcoded into the software and rely upon the exact structure of the binary file (particularly the index pointing to the black hole(s) at the previous timestep), any other type of query required modifying the code, and would likely be very inefficient unless the binary file was re-produced with additional indexing arrays customized for that query.

The process for a user to execute queries was cumbersome. After producing the binary file, the user could immediately query the black holes for a specific time (query a), but the histories (queries b and c) could not be queried by black hole id, but rather required the array index within the binary file, which could only be extracted via query (a). So the user must first get the list of all black holes at the end of the simulation, get the array location for the desired black hole from that list, and use that to query for the desired history. This became more involved with our new simulation, since the complete dataset could not be handled with a single binary file, instead requiring a series of binary files. Extracting the history for a single black hole thus required sequentially querying each file, with each successive file needing to be queried for all the black holes found to be part of the merger tree in the previous file, making the querying system significantly more time-consuming and error prone. In practice, these queries could take as long as 24 hours to finish, which is unsatisfactory for interactive analysis.

5.4 Building and Querying Black Forest Databases

To support the types of queries described in Section 5.2, we first need to transform the data produced by the simulation into a tabular, relational representation suitable for use in an RDBMS. Then, the data analysis is carried out by querying the database and processing the results using the algorithms described below.

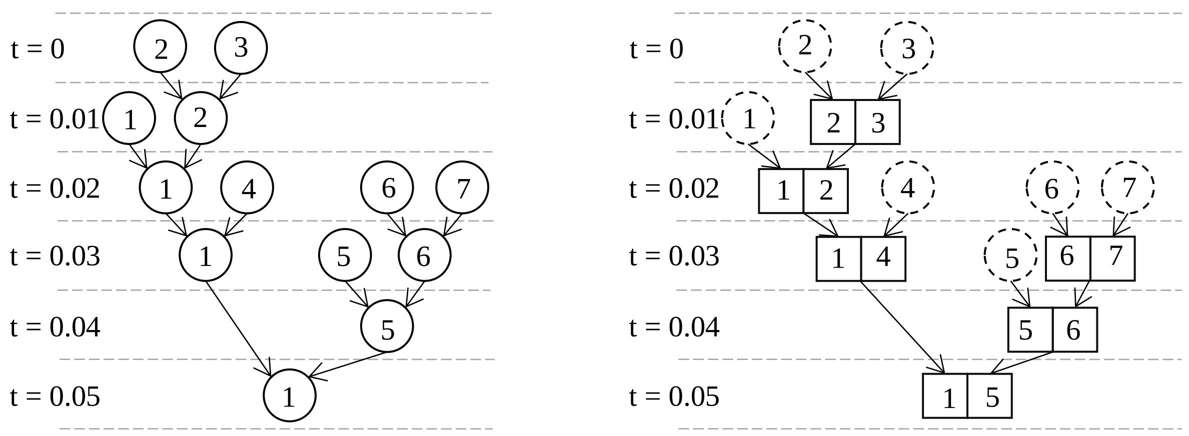


Figure 5.3: Merger tree representations.

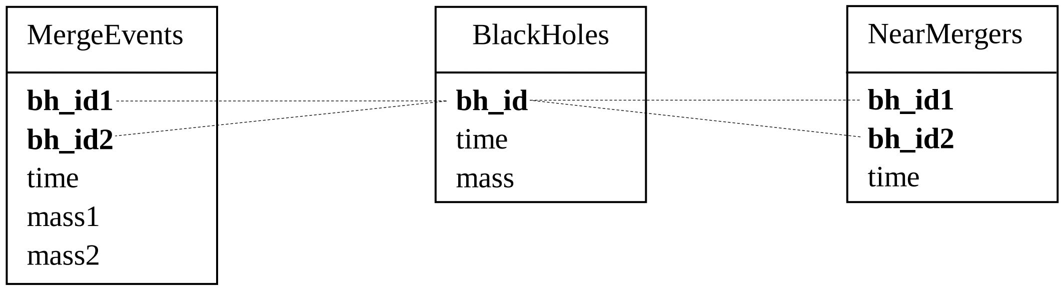


Figure 5.4: Basic schema for the black holes database. The MergeEvents (ME) table contains the ids and masses of the black holes that merged (bh_id1, bh_id2, mass1, mass2), and the time of the event. We always assume that bh_id1 will be passed along after the merging. The BlackHoles (BH) table contains the high-resolution time history of all the black holes in the dataset. The bolded fields represent the primary key in each table. The bh_id1 and bh_id2 fields in the ME table are used as search keys to retrieve the history of the respective black holes from the BH table.

5.4.1 Database Design

A simple schema to represent the black holes dataset comprises three main tables as shown in Figure 5.4: BlackHoles (BH), MergerEvents (ME), NearMergers (NM). These tables correspond to the three different types of entries present in a black hole dataset produced by a simulation. Splitting the black hole output dataset into these tables is achieved through a set of pre-processing scripts that cleanup, transform and build the database. The database also contains auxiliary indices and tables to keep summary information and track particle provenance, e.g., which processor hosted the particle, which file a record came from. Storing the merger event records in a separate table enables fast construction of the merger trees. In general, the merger events account for a small size of the black holes dataset. The ME table will often fit entirely in the page cache or can be loaded into the application memory for processing. Notice that the ME records do not have explicit links to other ME records that belong to the same merger tree. Different approaches for building and querying the merger trees follow. We use Python[32] to implement the procedures and interact with RDBMS.

5.4.2 Approach 1: Recursive DB Queries

The types of analysis described in Section 5.2 require retrieving the detailed time history for the black holes of interest that make up a merger tree. The procedure consists of two conceptual steps: (1) building the merger tree from the ME table to obtain the ids of the black holes in the tree; (2) querying the BH table to retrieve the associated history for the black holes. For explanation purposes, assume that the input for a query is the id of a black hole of interest (*qbhid*). The desired output for step 1 is the ids of all the black holes in the same merger tree as *qbhid*. This can be easily generalized to build multiple merger trees (from a list of black holes of interest), and to retrieve a subset of the tree or the most massive progenitor path.

The recursive DB approach (see Figure 5.5 for more details) works as follows. Given a *qbhid*, find the root of the merge tree by repeatedly querying the ME table. At each step, search for a record where bh2 equals the current value of bh. Once the root is found, recursively query the ME table for each of the root's children as shown in the BuildTree procedure (Python code). In the BuildTree procedure, *qresult* contains the left-most path for the subtree with the given root. The right child for each node in qresult is recursively added in the loop that iterates over qresult. Indices on the bh1 and bh2 fields are needed to speed up the queries.

This simple approach works well when only a small number of merger trees are being queried and the resulting trees have few records. However, it requires repeated queries to the database. The number of required queries is in the range [*m*, 2*m*], where *m* is the number of merger events in the tree. In most implementations, the repeated queries will dominate the running time due to the relatively high per-query cost involved in the communication with the database engine, query parsing and execution.

[32] http://www.python.org/

```
Type TreeNode { id, time, left, right }
Procedure BuildTree(bhroot, ctime):
// Recursively build a merger tree with bhroot as the root
// Find all the records that have the bh1 field = bhroot
qresult = SELECT bh2, time FROM ME
WHERE bh1 = bhroot AND time <= ctime ORDER BY time DESC
node = null; pnode = null; rnode = null
for (bh2, time) in qresult
        node = new TreeNode(id, time)
        node.right = BuildTree(bh2, time)
        if pnode is not null
                pnode.left = node // set left child for previous node in the result
        else rnode = node
        pnode = node
return rnode
```

```
Procedure BuildTreeDB(qbhid):
Input: qbhid is the query black hole id
Output: the merge tree that contains qbhid
// Find the root node of the merge tree
bh = qbhid
while bh is not null
        bhroot = bh
        bh = SELECT bh1 FROM ME WHERE bh2 = bh
// Now bhroot is the root node of the merge tree
// Then recursively build the merge tree
return Buildtree (bhroot, inf) // inf signals the simulation end time
```

Figure 5.5: Python code for the merge tree construction procedure.

5.4.3 Approach 2: In-Memory Queries

This approach consists of loading all the records from the ME table into a set in memory (MESet) and then looking up in MESet the events that belong to a tree. The algorithm is the following. Given a query *qbhid*, add it to a queue *pq* of pending black holes. For each element *bh* in the queue, fetch from MESet the records *r* that match *bh* (i.e., *r*.bh1 = *bh*). For each matching record *r*, add the corresponding *r*.bh2 to the *pq* queue. Repeat this process until every element of *pq* has been processed (i.e., the end of the queue is reached). At the end of the procedure, *pq* contains the ids belonging to the corresponding merger tree. The resulting ids are used to query the BH table to retrieve the detailed history for the black holes.

This approach issues a single heavy query to the ME table beforehand, and all subsequent query will benefit from the in-memory information. On the downside, it possibly requires large amounts of memory to hold the in-memory data structures, and thus it may not be suitable for a dataset with a very large number of merger events. The I/O cost of scanning the entire table can be amortized over queries for multiple merger trees.

5.4.4 Approach 3: In-Memory Forest Queries

This approach is a modification of the In-Memory Queries one. The basic idea is that instead of building the merger tree for a set of query *qbhid*s, the complete merger forest is built upon scanning the ME table. Although this approach incurs extra work to build all the trees, this cost is amortized when a large number of queries need to be processed. This approach is based on the Union-Find algorithm [Galler and Fischer, 1964] and adjusted to handle the peculiarities of the merger events representation, such as the fact that there is no explicit link that indicates the relationship between an ME node and its left child (bh1). The procedure builds the tree structure for each of the connected components.

This approach uses two associated set structures (e.g., hash tables or dictionaries). The first one, bh1Map maps from bh1 to the list of merger events that share the same value of bh1. The second one, bh2Map maps from bh2 to a single ME record that has the appropriate bh2 value. As the records are scanned from the database, they are added to bh1Map and bh2Map as shown in the BuildForestInMemory procedure (Figure 5.6). Then, the right-side links are created by iterating over the bh2Map and searching for the corresponding list of nodes in bh1Map, i.e., it creates a link where node.bh2 = node1.id. The left-side links are created by iterating over the lists found in bh1Map. During this loop, the root nodes for all the trees are determined using the findRootAndAddToForest procedure. This procedure only adds new trees to the forest and returns early when the root for a tree has already been found. Finally, the BuildForestInMemory procedure returns a tuple containing the forest and the maps from the ids to the tree nodes. Queries are processed by looking up the desired qbhid in either bh1Map or bh2Map to obtain a starting node in the tree from which the root of the tree can be obtained. From the root, all the nodes in the tree can be returned.

```
Procedure BuildForestInMemory(db):
Input: the DB with the ME table
Output: a forest containing all the merge trees in ME
// Scan over all ME records
cursor = SELECT bh1, bh2, time FROM ME;
foreach (bh1, bh2, time) in cursor
    node = new TreeNode(bh1, time, bh2)
    bh2Map.put(bh2, node) // Map from bh2 to this node
    bh1Map.addToList(bh1, node) // Map from bh1 to a node list
foreach node in bh2Map
    // Create the link for the right-side child
    node.right = bh1Map.get(node.bh2) // It may be null
forest = emptySet()
foreach lst in bh1Map
    sortbytime(lst)
    // Create links from lst[n-1].left to lst[n]
    createLinkOnBh1(lst)
    findRootAndAddToForest(lst, forest)
return (forest, bh1Map, bh2Map)
```

Figure 5.6: Python code for the BuildForestInMemory process.

5.4.5 Approach 4: ForestDB

The ForestDB approach builds on the techniques used in the In-Memory Forest approach. The basic idea is to build the black hole forest in the same way as in the in-memory case. Then tag each tree with an identifier (tid). The forest can be written back into a table in the database that we will call merger events forest (MF). This is done as a one-time pre-processing step. The schema for this table is the same as the ME's schema (see Figure 5.4), with the addition of the tid field. Two conceptual steps are performed at query time to extract a merger tree for a given qbhid. First, search the MF table for a record matching *qbhid*. The tid field can be obtained from the record found in this step. Second, retrieve from the MF table all the records that have the same tid. These two steps can be combined in a single SQL query. Moreover, the detailed history for the black holes in the tree can be retrieved from the BH table using a single query that uses tid as the selection criteria and joins the MF and BH tables. Indices on the bh1, bh2 and tid fields are required to speed up these queries. Alternatively, the indices on bh1 and bh2 can be replaced by an additional auxiliary indexed table to map from bhid to tid.

The MF table only stores the membership of the merger event records to a particular tree. Notice that the MF table does not explicitly store the tree structure, i.e., the parent-child relationships. Also, the MF table only stores the internal nodes of the merger tree. The leaves are not explicitly stored. Instead the relevant data (such as the leaf's bhid) is stored in the parent node. This makes for a more compact representation as it requires fewer records in the MF table.

5.5 Evaluation

We built a prototype implementation of the approaches described above using Python and SQLite[33]. Our evaluation aims to characterize the relative performance of these approaches and determine the feasibility of using RDBMSs in the analysis of black holes datasets. For this purpose, we ran a set of experiments using a dataset produced by the largest published cosmology simulation to date.

5.5.1 Workload

The dataset was produced by a ΛCDM cosmology simulation using the GADGET-3 [Springel, 2005] parallel program. The simulation ran at the National Institute for Computational Sciences (NICS) using all 99072 processors of the Kraken supercomputer and running a total of 7 days (wall-clock time). The simulation invoked 16,384 MPI processes, each with 6 threads. A snapshot of all the particles is 3 TB in size. The simulation contained 33 billion dark matter particles and 33 billion hydro particles in a box of 533.33Mpc/h in size. At the end of the simulation (z = 4.7), there are 2.4 million black holes. The size of the resulting black holes dataset is 84 GB. The black hole history table contains 420 million records corresponding to 3.4 million unique black holes and 1 million merge events. Figure 5.7 shows the distribution of tree sizes in number of merger events in the ME table.

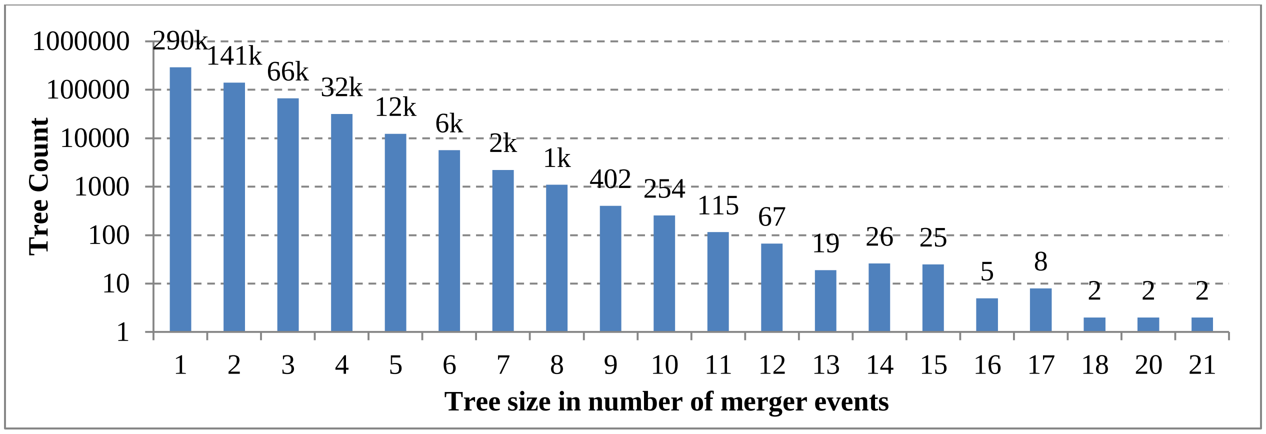


Figure 5.7: Distribution of tree sizes in the black holes dataset. The X axis is the size of a merger tree measured as the number of events in a tree. The Y axis is the number of trees of that size in $\log_{10}$ scale.

5.5.2 Storage Requirements

For our prototype implementation, we stored the BH data in SQLite (v3.7.3). We gathered the storage requirement of storing the BH dataset in the database and compared it to a raw binary representation. In the binary layout, the records are stored in a dense array of size $N \times sz$, where

[33] SQLite: An open-source RDMBS. http://www.sqlite.org/

N is the number of records in the table and *sz* is the size of an equivalent unpadded C structure. Table 5.1 shows the sizes of the main tables in SQLite and the corresponding size of the binary array representation. We built various indices required for speeding up the queries to the BH and ME tables. Table 5.2 contains the size of the indices and additional summary and provenance database.

Table 5.1: Size of main tables in the BH dataset

Table	- Records	- Tab. Size	- Bin. Size
BH	420M	50GB	22GB
ME	1M	49MB	26MB
NM	175M	13GB	4.6GB
Total		63GB	27GB

Table 5.2: Size of indices and auxiliary tables

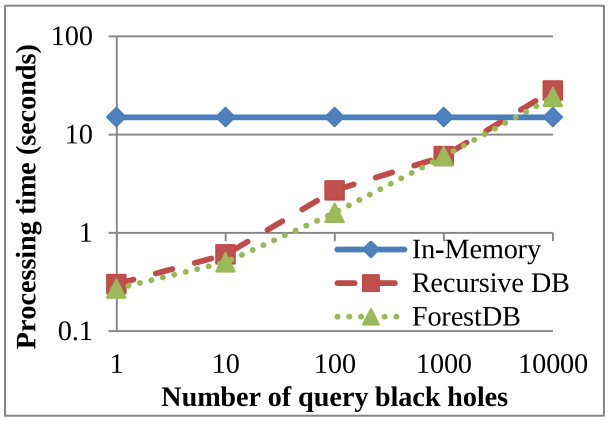
Item	Size
BH index on bhid	8GB
BH index on time	8GB
ME index on bh1	17MB
ME index on bh2	15MB
MF table	53MB
MF indices bh1,bh2	32MB
MF index on tid	12MB
Aux. and provenance data	6GB

5.5.3 Performance

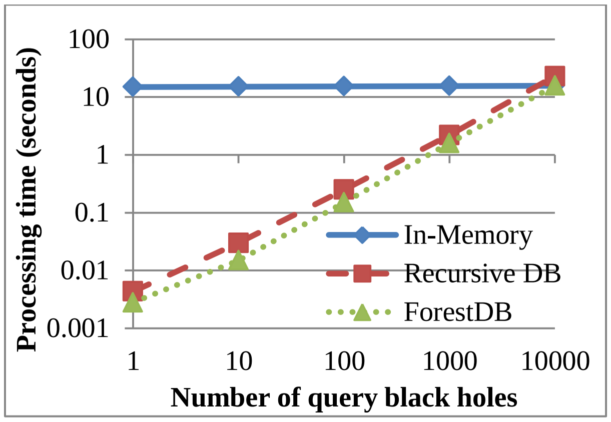
To characterize the performance of the developed approaches, we conducted a series of micro benchmark experiments that correspond to the steps involved in answering queries for the detailed time history of merger trees.

Setup. The experiments were run on a server host with 2 GHz dual core Intel(R) Xeon(R) CPUs, 24GB of memory and a 0.5 TB software RAID 1 volume over two SATA disks. The OS was Linux(R) running a 2.6.32 kernel. Our prototype implementation of the different approaches is written in Python (v3.1) and the data is stored in SQLite.

Building Merger Trees. The first set of micro benchmark experiments corresponds to the steps needed to build the merger trees for a set of query black holes (qbhs). We compared three of the approaches explained in Section 5.4: (a) Recursive DB – RDB (Section 5.4.2), (b) In-memory – IM (Section 5.4.3), and (c) Forest DB – FDB (Section 5.4.5). The In-memory Forest approach (Section 5.4.4) was only used to build the tables for Forest DB. For these experiments we selected black holes (qbhs) that belonged to merger trees in the ME table. We timed the process of satisfying a request to build one or more merger trees specified by the requested qbhs. The processing time includes the time required to issue and execute the database query, retrieve and post-process the result to build the trees.



(a) Cold OS cache

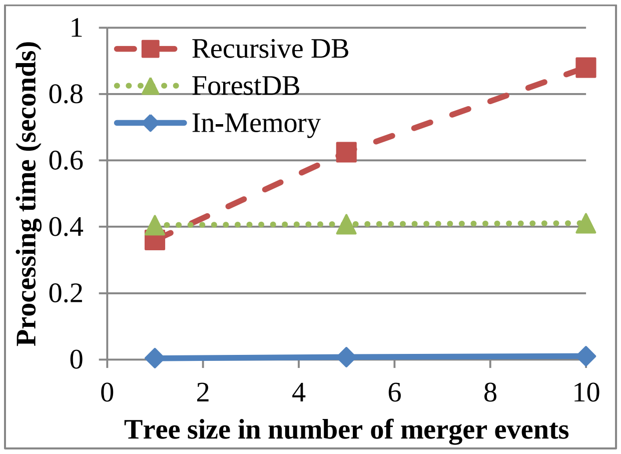


(b) Warm OS cache

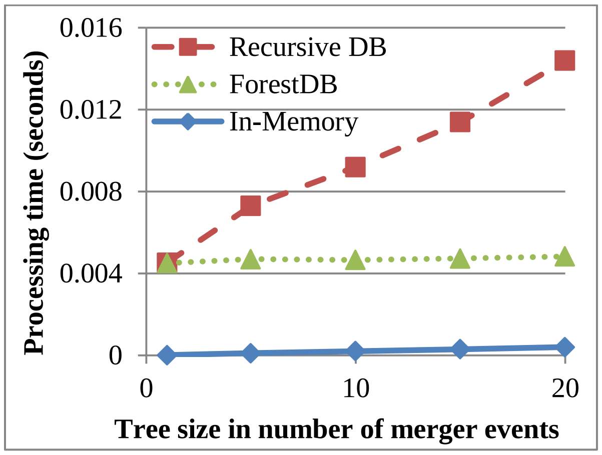
Figure 5.8: Running time to obtain the merger trees for the different approaches. These results correspond to a tree of size 5. The X axis is the number of trees being queried at once in a batch. The Y axis is the elapsed time in seconds (log scale) to retrieve the corresponding records from the ME table. The cases with cold (a) and warm (b) OS caches are shown.

In the first experiment, we kept the tree size fixed at 5 and varied the number of black holes for which a tree is requested (number of qbhs). The results for the different approaches are shown in Figure 5.8. The X axis is the qbh count varying from 1 to 10,000. The Y axis shows the processing time (seconds) in log scale. For qbh counts less than 1K, both the RDB and FDB approaches are faster than the In-Memory approach. The RDB approach is not as expensive as we originally thought for small queries. It was surprising to find out that for the cold OS cache setup (Figure 5.8a), the processing time for RDB and FDB does not differ significantly. For the warm OS cache, there is a constant (in log scale) difference between RDB and FDB. The IM approach pays upfront a relatively large cost of 15 seconds to load the entire ME table, then the processing cost per requested qbh is negligible, and thus can be amortized for a large number of qbhs.

Figure 5.9 shows the effect of the merger tree size on the request processing time. In this experiment the requests were grouped by tree sizes (X axis = 1, 5, 10, 15, 20). This experiment was performed with a warm OS cache and cold database cache. The initial load time for the IM approach is not included in the processing time shown in the graphs, only the time to build the tree in memory. The running time for the RDB approach increases as the trees get larger. This is due to the larger number of queries to the ME table needed to process each tree in the recursive approach. The FDB approach requires a single query to the ME table per requested tree.

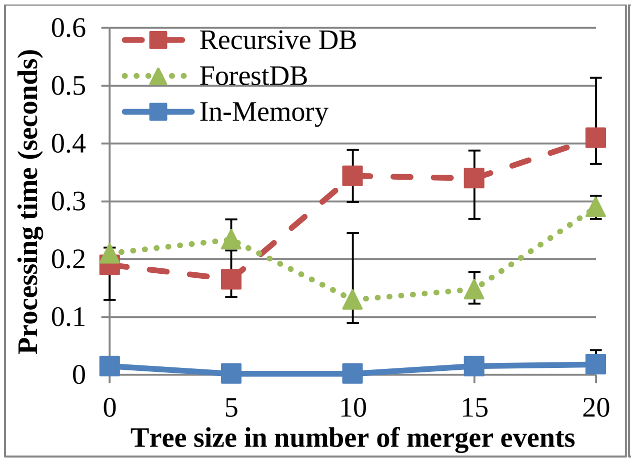


(a) Batch size = 250 qbhs

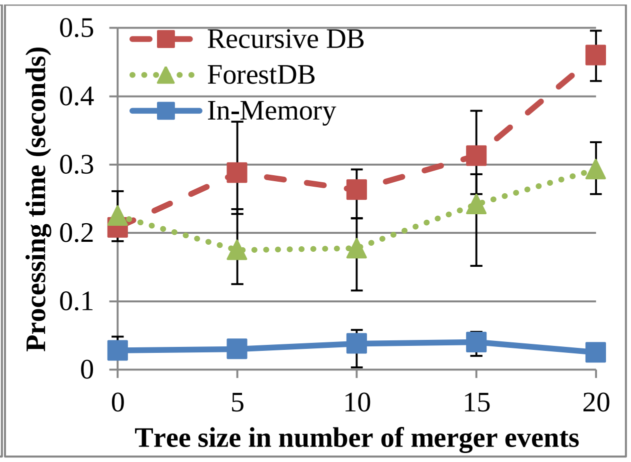


(b) Batch size = 2 qbhs

Figure 5.9: Processing time for building the merger trees using various approaches. This experiment was performed with a **warm** OS cache and a cold DB cache. The X axis is the size of the resulting tree; (a) and (b) show the time to process 250 qbhs and 2 qbhs per request respectively. The Y axis is the elapsed time to build the number of trees of each size.



(a) Request size = 2 qbhs



(b) Average over requests for 2 qbhs

Figure 5.10: Processing time for building the merger trees for 2 qbhs per request with **cold** OS and DB caches. The X axis corresponds to the tree size. The Y axis is the elapsed time in seconds to build the trees. (a) shows the time for a particular request set with 2 qbhs. The average processing time for multiple 2-qbhs sets is shown in (b).

Figure 5.10 shows the processing time according to tree size for requests of size 2 qbhs and cold OS cache. In these experiments, a request for 2 qbhs (of a given tree size) is repeated 9 times with cold cache. The minimum average and maximum are shown for each point. Figure 5.10(a) shows the result for a particular set of request qbhs, and Figure 5.10(b) shows the aggregate across different 2-qbh request sets. The difference between the RDB and FDB approaches is inconclusive in this scenario, especially for small trees. In this case, the request time is dominated by the characteristics of the I/O devices, rather than the particular approach.

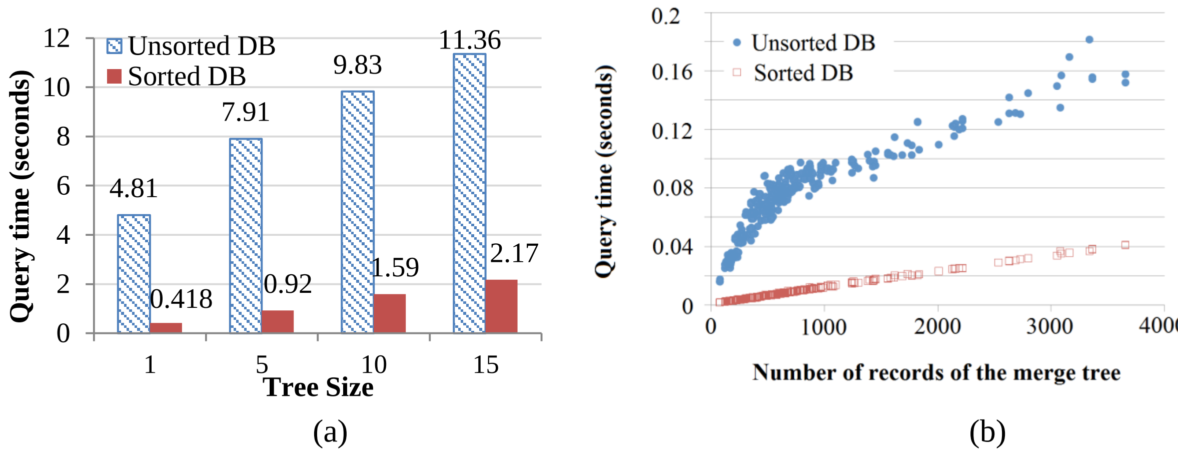


Figure 5.11: Time to retrieve the detailed BH history from the BH table for merger trees of various sizes. The running times for queries to sorted and unsorted BH tables are shown. Figure (a) shows the elapsed time grouped by tree size. Figure (b) shows the same data grouped by the number of BH records comprising the merger trees.

Retrieving the Time History for Merger Trees. In the second set of experiments, we retrieved the detailed time history for a set of trees retrieved in the previous step. This entails retrieving from the BH table all the records for the corresponding BH in a given merger tree. For each tree size (1, 5, 10, 15), we retrieved the BH histories for 100 trees of that size. Figure 5.11a shows the elapsed time in seconds to retrieve the detailed records from the BH table. The times are shown for an unsorted indexed BH table and a BH table sorted by the black hole id. As expected for this query pattern, sorting by the BH id is beneficial. Figure 5.11b shows the elapsed time according to the number of records that were retrieved from the BH table. Each data point corresponds to a merger tree that resulted in retrieving the number of BH records shown in the X axis. The Y axis is the elapsed time in seconds for the unsorted and sorted BH tables.

5.5.4 Pre-processing

Creating the BH database involves loading the data and creating the needed indices. Although this is a one-time cost, it is a necessary step to enable the data analysis. If this process takes too long, it may become a barrier for adoption of the DB approach. The initial loading operation presents a good opportunity for performing needed data cleanup operations such as removing duplicate or unwanted records. Loading the database takes 10 hours using our pre-processing scripts in our platform. Under closer examination we determined that the process was CPU intensive. Its running time can be greatly reduced with a compiled language implementation and by processing in parallel subsets of the data being loaded. Creating the MF table takes 55–65s once the data is loaded. The pre-processing time of the current implementation is already a significant improvement over the previous custom approach where creating the binary files required days. In the custom approach, the data of recent simulations could not be loaded into a single file in a single step. Instead, multiple binary files need to be created to deal with the larger data sizes. For example, creating the binary file for the last fraction of the most recent simulation took approximately 12 days. The resulting files cannot be easily queried at once. In contrast, with the DB approach, not only the initial pre-processing time is lower, but also all the data is in a single database that can be queried in a consistent and flexible manner.

5.6 System Utilization and Future Work

Our systems are very helpful in several astrophysics projects ([DeGraf et al., 2012a] [DeGraf et al., 2012b] [Di Matteo et *al*., 2012] [Khandai *et al.*, 2012]). Astrophysicists usually use it to extract individual black hole histories (query (b) and (c) in Section 5.3.1), and to get a full list of black holes at any given time (query (a) in Section 5.3.1). Occasionally, they use the system to query other useful information, like getting all the black holes in a given time frame, or acquiring the statistics of merger events. These extractions normally are an intermediate step, followed by subsequent analysis written in IDL[34] or C++.

Comparing to the old non-DB approach, RDBMS solution is faster and more flexible. It is easier to add new types of queries (like to get all black holes in a given time frame), or new post-processed properties of specific black hole entries (for example, host galaxy mass). The database approach lets us easily add an index on a new column if it turns out helpful, and we can make additional tables which are added to the database based on the current information. All of above features are not easily extensible to the old approach.

Additionally, the RDBMS approach makes it more convenient to link the black hole dataset to databases and tables of other particle or galaxy properties from the same simulation. This functionality is currently being implemented and not yet functional, and the goal is to not only provide an easier way for data access, but also make the database publicly accessible, ideally through an online-query system, like the SkyServer interface of Sloan Digital Sky Survey[35].

Now that scientists are able to easily query the data, they are able to carry more involved and extensive types of analysis, which in turn results in more complex queries and access patterns. One of the next steps is to add other types of data, such as information about galaxy halos, to correlate black holes with their surrounding environment. Adding new types of data brings challenges due to the size of those other data sources. Current black hole datasets can be managed with RDBMS using a single server-class host. The size of these datasets will increase as cosmology simulations grow larger. Alternative, more efficient approaches will be needed to manage and analyze these coming datasets. To address the challenges of scaling to larger data

[34] IDL programming language, http://www.exelisvis.com/ProductsServices/IDL.aspx
[35] http://skyserver.sdss.org/public/en/tools/search/sql.asp

sizes, there are a new generation of tools that will leverage distributed, scalable, structured table storage systems such as Bigtable [Chang *et al.*, 2006] and Cassandra [Lakshman and Malik, 2010]. Currently, the data analysis is delayed by the time required to load the simulation output into the database. Although it is a one-time cost, the time required to load multi-terabyte and petabyte datasets is potentially long, in the order of multiple days on relatively small computer clusters. We are exploring efficient in-situ processing and bulk loading mechanisms to address this issue.

5.7 Conclusion

Rapid, flexible analysis of black hole datasets is key to enable advances in astrophysics. We presented a set of algorithms for processing these data using a database approach. The database approach is not only flexible, but also exhibits good performance to support interactive analysis. Our approach has been used by astronomers to analyze large-scale astronomical data.

Chapter 6 Quasar Detection

6.1 Background

Astronomers observe the universe via modern telescopes. Sometimes, it is not easy to identify celestial objects only through telescope images. For example, the Andromeda Galaxy (catalogued as *M31*) – the nearest spiral galaxy to our Milky Way galaxy, was considered a nebula when first discovered. Later in 1864, William Huggins discovered that the spectrum of M31 is continuum – very different from that of a nebula. But people still thought M31 is a nearby object. Finally in 1925, Edwin Hubble successfully measured the distance of M31, and determined that it is not a cluster of stars or gas within our galaxy, but an entirely separate galaxy.

Although nowadays we have modern high-resolution telescopes, the same problem still exists: When using telescopes to observe an astronomical object, which looks like "shining dots", sometimes we cannot tell what it really is. So in this chapter, we use machine learning techniques to help astrophysicists better identify astronomical objects from telescope images. We focus on one specific type of objects, *quasars*.

A quasar is an unusually bright galaxy, which is visible to us even if it is far away from the earth. Astronomers are interested in studying the properties of quasars because it provides information about remote regions of the universe, and thus advance the study of universe expansion.

It is hard to identify quasars accurately, because regular telescopes do not explicitly provide the distance information of an object, and quasars look similar to regular galaxies and other types of objects. We help astrophysicists better classify quasars. Specifically, given a dataset of astronomical objects, we identify quasars from all other types of objects. Two groups of machine learning algorithms are applied to this problem:

Supervised Learning: This is the basic and simple case, where we have already acquired the labels of all objects[36] and use them to train a classifier on new data. To solve this problem, we apply a variety of supervised learning algorithms, including decision trees, support vector machines, and *k*-nearest neighbors.

Active Learning: In this scenario, we assume the labeled data do not come for free, and we want to train an accurate classifier while minimizing the number of labels. In reality, labeling astronomical objects requires additional measurements to acquire and analyze their spectrum, which costs a couple of dollars per object. As a result, for future massive sky surveys, it may be very expensive to get the labels of all objects. Active learning handles this kind of situations: assuming that a user can provide only a small amount of labels, and an active learning algorithm interacts with the user and automatically select appropriate objects for labeling [Settles, 2010].

[36] That is to say, for each object, we already know whether it is a quasar or not.

Active learning technique is usually more effective than the ordinary supervised learning techniques: It requires fewer labels to get a classifier with comparable accuracy.

6.2 Related work

The detection of quasars from multicolor imaging data dated back to Sandage and Wyndham [Sandage and Wyndham, 1965]. Richards *et al.* identified quasars using fluxes of astronomical objects in five color bands (U, G, R, I and Z). Using non-parametric Bayesian classification together with fast kernel density estimation [Gray and More, 2003], their algorithms achieved 65–95% precisions and 70–95% recalls, varying on different datasets [Richards *et al.*, 2002] [Richards *et al.*, 2004] [Richards et al., 2009].

Despite the high precision and recall, the solution is less powerful to identify extra-bright objects, due to interlopers like white dwarfs and faint low-metallicity F-stars. The solution also has trouble classifying high-redshift quasars ($z > 2.2$, where z represents redshift), where the precision drops below 50% [Richards et al., 2009].

6.3 Data and experimental setup

We conducted experiments on a dataset provided by our collaborators in University of Pittsburgh. The dataset, which is manually selected as possible quasar candidates, contains 8,200 objects. We observed the apparent magnitudes of each object – the logarithm of its brightness – as its feature. Object brightness is measured in five color bands, so we acquired a five-dimensional vector of numeric features for each object. The dataset also includes the true label of each object, determining whether it is a quasar or not (Figure 6.1).

6.4 Solution

6.4.1 Supervised learning

We ran a series of machine learning experiments using Java and Weka [Holmes et al., 1994]. We considered the following four supervised learning techniques:

C4.5 – A decision tree classifier [Quinlan, 1992] that learns and organizes a set of decision rules in a tree structure that determines the label of each object.

***k*-NN** – A *k*-Nearest Neighbor classifier. The label of each object is determined by the majority of its *k* closest objects.

SVM – A Support Vector Machine classifier. SVM chooses from possible hyperplanes that separate the objects of two classes. The chosen hyperplane represents the largest separation, or margin, between the two classes.

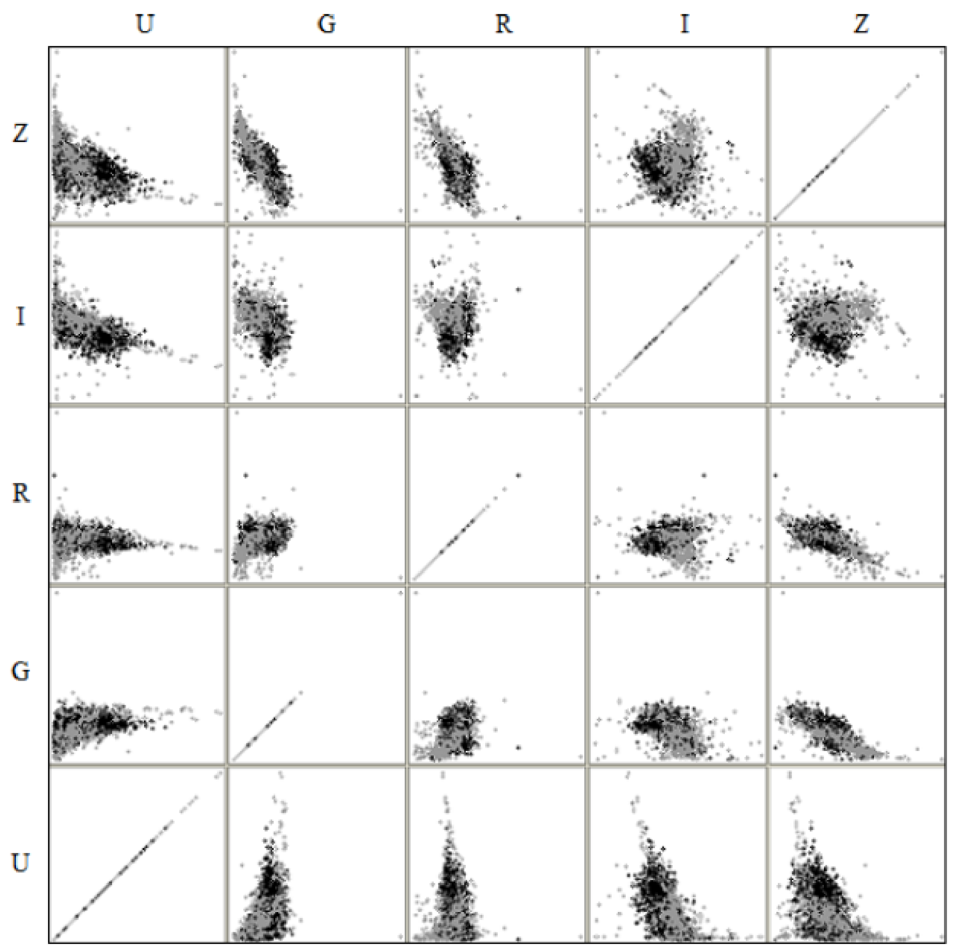


Figure 6.1: The distribution of objects in pairwise feature space. The features are the magnitude of objects in five color bands (U, G, R, I, Z). 8,200 objects are shown. Quasars are plotted in black and non-quasars are plotted in grey.

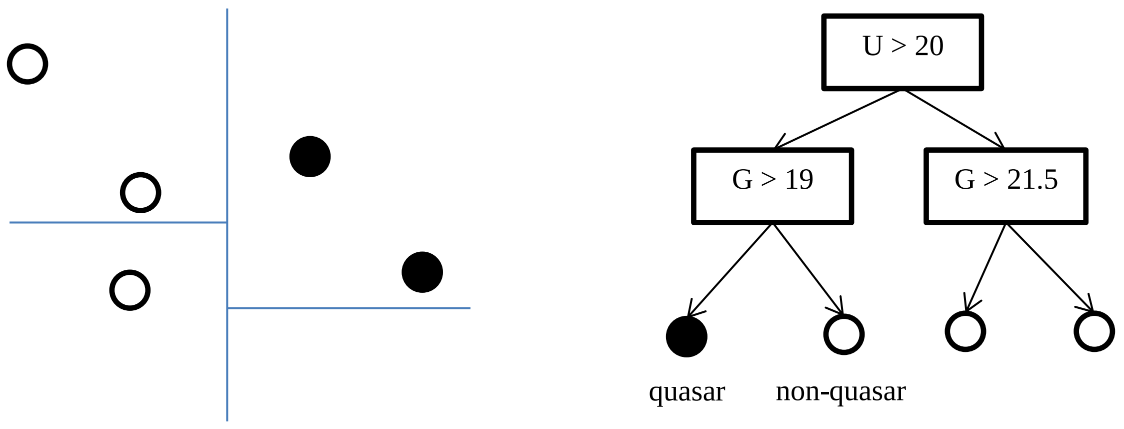
Majority Vote – A majority vote on the results of C4.5, *k*-NN, and SVM.

Illustrations of the three techniques are shown in Figure 6.2.

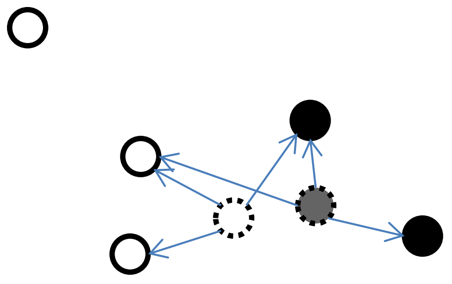
We used 10-fold cross-validation to test the performance of each technique, and report the average precision (#correctly_classified_quasar / #classified_quarsars), average recall (#correctly_classified_quasar / #true_quasars), and their standard deviations in Table 6.1.

Table 6.1: Precision, recall, and their standard deviations of the supervised learning techniques

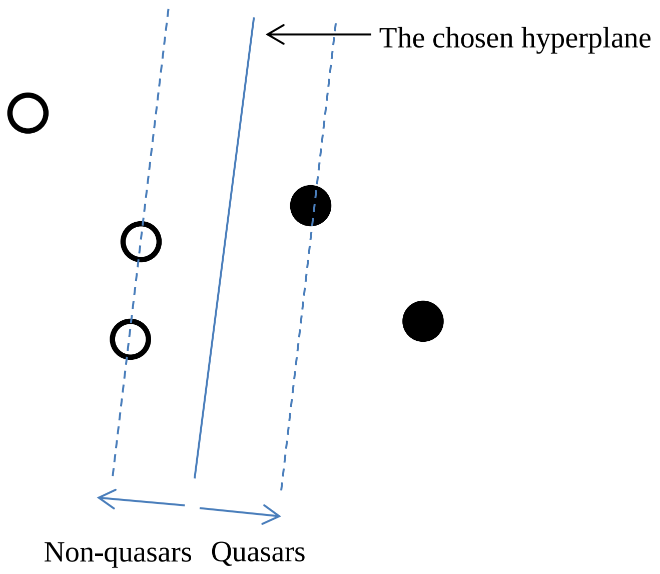
	Average Precision	Precision Stdev	Average Recall	Recall Stdev
C4.5	0.726	0.008	0.612	0.004
k-NN	**0.737**	0.01	0.612	0.003
SVM	0.456	0.023	**0.699**	0.058
Majority Vote	**0.812**		0.295	



(a) Decision tree.



(b) *k*-nearest neighbor. Here *k* is set to 3.



(c) SVM.

Figure 6.2: The three techniques we use to train a classifier.

Among the results, the performance of C4.5 and *k*-NN are similar, with around 72% precision and 61% recall. SVM exhibit a poor precision (46%), but its recall is better (70%). The majority vote has the highest precision (81%), but its recall is the lowest (30%).

6.4.2 Active learning

In this section we apply active learning algorithms to the quasar detection problem. Under this scenario, we assume that most of the input objects are unlabeled and the acquisition of labels is not free. An active learning algorithm automatically decides which objects to query and actively asks human for labels. Comparing to supervised learning techniques, it usually requires fewer labels to train an equally good classifier. Consequently it is suitable for the applications where the acquisition of labels is expensive.

Formally, assume that the input objects $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_\mathrm{n})\}$, where x_i stands for a feature vector, and $y_i \in \{-1, 1\}$ is its binary label. Assume further that at phase t ($t \in \mathbb{Z}^+$ and $t > 2$), L is a set of t–1 objects whose true labels have already been learnt[37], and U is set of the remaining objects whose labels are not yet revealed. An active learning algorithm will train a classifier C_L on L, and choose an unlabeled object (*query*) $(x_i, y_i) \in U$ for human to label. After the label y_i is discovered, the query (x_i, y_i) will be added to L and removed from U. Then the learning algorithm will train another classifier, and choose the next object to label, and so on.

Next we introduce a baseline, and three active learning techniques [Baram *et al.*, 2004]. All active learning algorithms use SVMs as their classifiers.

Baseline. We use a simple random-selection as the baseline for other active learning algorithms. The baseline learner randomly chooses an object in U as query. Any active learning algorithm needs to beat this baseline to have any practical value.

Simple algorithm. After an SVM classifier C_L is trained over L, the learner will choose the object that is the closest to the hyperplane of C_L as the next object to query. This algorithm has the effect of quickly reducing the size of possible hyperplanes [Tong and Koller, 2001].

Kernel Farthest First (KFF). Given the set of labeled instances L, the KFF algorithm chooses the next object x_i which is the farthest to L (here the distance from a point to a point set is defined as the minimal distance to any point in the set). We measure the distances in the kernel space of SVM. Intuitively this algorithm is comprehensible because it chooses the query that is the most dissimilar to any labeled object.

Self-Confidence. The decision boundary of a SVM classifier is a linear hyperplane:

$$\boldsymbol{w} \cdot \boldsymbol{x_i} - b = \mathbf{0}$$

[37] Initially (when t = 3) L contains two randomly selected objects, one with label 1 and one with label –1.

Here · denotes dot product, and ***w*** and *b* are the trained parameters for the decision boundary. To further get a probabilistic estimation for each object, we apply a normal logistic regression model to the above formula:

$$P_L(y_i = 1 | \boldsymbol{x_i}) = \frac{e^{\mathbf{w} \cdot x_i - \mathbf{b}}}{1 + e^{\mathbf{w} \cdot x_i - \mathbf{b}}}$$

$$P_L(y_i = -1 | \boldsymbol{x_i}) = \frac{1}{1 + e^{\mathbf{w} \cdot x_i - \mathbf{b}}}$$

Assume that we have trained a SVM classifier C_L and its probabilistic model P_L at phase *t*. Now, for each unlabeled object and its label $(x_i, y_i) \in U$, we train a new SVM model and a probabilistic model with $L \cup (x_i, y_i)$, and estimate the "self-estimated log-loss" of the remaining unlabeled objects on the new model:

$$E\left(C_{L\cup(x_i,y_i)}\right) = -\frac{1}{|U-1|}\sum\nolimits_{(x',y')\in U, x'\neq x_i} P_{L\cup(x_i,y_i)}(y'|x')\log\left(P_{L\cup(x',y')}(y'|x')\right)$$

After that, we estimate the average expected loss of labeling x_i:

$$Loss(x_i) = \sum\nolimits_{y_i} P_L(y_i|x_i)\, E\left(C_{L\cup(x_i,y_i)}\right)$$

, and finally the learner choose the x_i with the smallest $Loss(x_i)$.

Note that the computational complexity of the Self-Confidence algorithm is high: At each phase, the Self-Confidence algorithm trains $2|U|$ SVM classifiers, which is costly especially for a large dataset when most of the objects are unlabeled. Thus we implemented a modified algorithm: at each phase, we do not evaluate the loss function of each unlabeled object, but only a fixed number of randomly selected unlabeled objects (in the following experiments, the constant is set to 10).

To evaluate the effectiveness of an active learning algorithm, here we look at the classification precision on each classifier C_L. At the beginning of our experiments we left out a test set *S* that is randomly selected from *U*, and applied each C_L to the test set to calculate the classification precision, defined by #objects_classified_correctly / |*S*|.

For each active learning algorithm we repeated our experiment for 30 times. Each time, two initial labeled objects were randomly chosen. The average precision of each algorithm is reported in Figure 6.3.

It is surprising to see that two of the active learning techniques, Simple and KFF, performed worse than the baseline, which indicates that they are worse than the random selection. On the other side, the Self-Confidence algorithm outperformed the baseline, although their difference is not obvious.

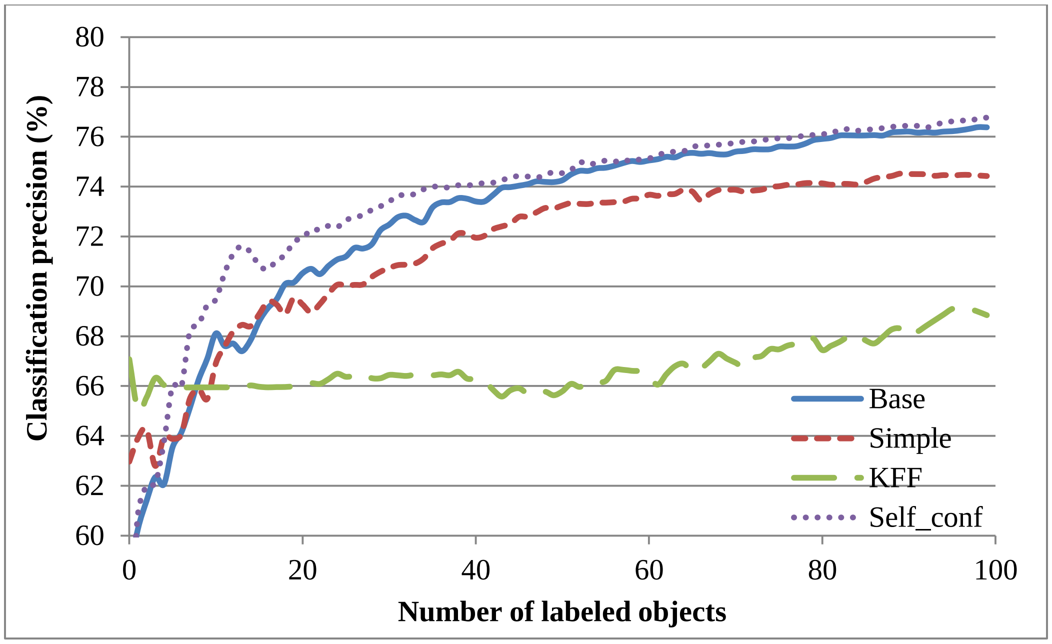


Figure 6.3: The performance of different active-learning techniques: random baseline (Base), simple algorithm (Simple), Kernel Farthest First (KFF), and Self-Confidence (Self_conf).

6.5 Conclusion

In this chapter we report our initial efforts on the quasar detection problem. We utilized two kinds of machine learning techniques, supervised learning and active learning, to automatically identify quasars from other kinds of objects. Our experiments showed that supervised learning techniques, especially the majority vote on different algorithms, lead to high precision. However, our active learning algorithms did not hold expected advantage. As the future sky surveys keep getting larger, automatic analysis become more attractive, and it is interesting to see how the above techniques perform on future larger datasets.

Chapter 7 Conclusion and Future Directions

In this thesis I introduce our efforts to analyze large-scale astronomical datasets, which currently contain billions of celestial objects. We show that there is wide area for collaboration between computer science and astronomy. Specifically, we utilize various computer science techniques, including algorithm, distributed computing, database, hardware, and machine learning to provide fast and scalable solutions toward big data.

As more and more astronomical data will be available in the near future, there are multiple potential opportunities for large-scale astronomical data analysis. Other than the future works introduced at the end of each chapter, I feel the following research directions interesting and promising:

- A basic topic is how to store and process large amount of available data in the near future. As astronomers estimated, the initial computer requirements for Large Synoptic Survey Telescope (LSST) [Ivezic et al., 2008] are already at 100 teraflops of computing power and 15 petabytes of storage, which calls for more efficient algorithms, carefully designed systems, as well as flexible software frameworks.
- One possible research direction is to make use of a new wave of database techniques to better store and process big data. One of the new trends is called "Column-oriented" databases (Vertica[38], MonetDB [Boncz 2002]), which store database tables as sections of columns, not on rows. In an extreme case (e.g. if a query only covers a small number of the columns in a table), a column-oriented database can answer user queries ten time faster than traditional databases [Stonebraker *et al*., 2007]. Previous work already applied column-oriented database to astronomical field, which used MonetDB to store SDSS's Skyserver [Ivanova *et al*., 2007], and it is worthwhile to see whether this direction is favorable on larger databases.
- Another new database technique is "No-SQL" (Bigtable [Chang *et al*., 2006], Cassandra [Lakshman and Malik, 2010]), where these new database systems provide a faster and more scalable solution, although not supporting all the functionalities of ordinary relational databases. For example, Google's Bigtable system provide only one data model, (key, value) pairs. Users can only add, remove or search for information on keys. It is interesting to see whether the data retrieval patterns to astronomical databases can be represented by the simplified data models, and if so, these new databases can provide a more scalable and robust alternative to existing systems.
- Most future digital surveys will store many time slices of data. For example, Panoramic Survey Telescope and Rapid Response System (Pan-STARRS) [PanStarrs] [Kaiser et al., 2002] can map the entire sky in just 40 hours. How to utilize these multiple slices of data becomes an interesting topic. For example, how to locate a celestial object

[38] http://www.vertica.com/

among datasets at different times? Are there any spatial indexes that can accelerate such queries? Is there any related data mining problems? And so on.

- As more and more astronomical data become available, most will not have human labels. For example, LSST is expected to take more than two hundred thousand pictures per year, much more than what humans are expected to review. This is a big opportunity for active learning techniques, which aims to learn a concept while minimizing the number of labels it needs from human. We have provided preliminary results about the use of active learning algorithms in Chapter 6, but it will be more challenging to consider similar problems at larger scale. Another potential direction is to use the “wisdom of crowds”, i.e. crowdsourcing to quickly collect human labels [Lintott et al., 2008].

Bibliography

[Abazajian *et al.*, 2009] Kevork N. Abazajian *et al*. The Seventh Data Release of the Sloan Digital Sky Survey. The Astrophysical Journal Supplement Series, 182(2), pages 543–558, 2009

[Anderson *et al.*, 2012] Lauren Anderson *et al*. The clustering of galaxies in the SDSS-III Baryon Oscillation Spectroscopic Survey: Baryon Acoustic Oscillations in the Data Release 9 Spectroscopic Galaxy Sample. MNRAS (2012) 427.

[Baram *et al.*, 2004] Yoram Baram, Ran El-Yaniv, and Kobi Luz. Online Choice of Active Learning Algorithms. The Journal of Machine Learning Research, Volume 5, 2004, Pages 255-291.

[Baruffolo, 1999] Andrea Baruffolo. R-trees for astronomical data indexing. Astronomical Data Analysis Software and Systems VIII, ASP Conference Series, 172, 1999.

[Baumann *et al.*, 1997] P. Baumann, P. Furtado, R. Ritsch, and N. Widmann Geo/environmental and medical data management in the rasdaman system. In 23th VLDB Conf. (1997), pp. 548–552.

[Bayer and McCreight, 1970] R. Bayer, and E. M. McCreight. Organization and maintenance of large ordered indices. In Proc. of SIGFIDET Workshop on Data Description and Access (Rice University, Houston, TX, USA, Nov 15-16 1970), ACM, pp. 107–141.

[Bayucan *et al.*, 1999] Albeaus Bayucan, Robert L. Henderson, Casimir Lesiak, Bhroam Mann, Tom Proett, and Dave Tweten. Portable Batch System (PBS), External Reference Specification. MRJ Technology Solutions, 2672 Bayshore Parkway, Suite 810, Mountain View, CA 94043, 1999.

[Belussi and Faloutsos, 1995] Alberto Belussi and Christos Faloutsos. Estimating the selectivity of spatial queries using the „correlation" fractal dimension. In Proceedings of the Twenty-First International Conference on Very Large Data Bases, pages 299–310, 1995.

[Bentley 1975] J. L. Bentley. Multidimensional binary search trees used for associative searching. Communications of the ACM, 18(9), pages 509–517, 1975.

[Boncz 2002] P. A. Boncz. Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications. Ph.d. thesis, Universiteit van Amsterdam, May 2002.

[Botzler *et al.*, 2004] C. S. Botzler, J. Snigula, R. Bender, and U. Hopp. Finding structures in photometric redshift galaxy surveys: an extended friends-of-friends algorithm. Monthly Notices of the Royal Astronomical Society (NMRAS) 349 (Apr. 2004), 425–439.

[Brunner *et al*., 1999] R. J. Brunner, J. Gray, P. Kunszt, D. Slutz, A. S. Szalay, and A. Thakar Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey. Tech. Rep. MSR-TR-99-30, Microsoft Research, June 1999.

[Chandra *et al*., 2000] R. Chandra, R. Memon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. Parallel Programming in OpenMP. Morgan Kaufmann, 2000. ISBN 1558606718.

[Chang *et al*., 2006] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, 7:205–218, 2006.

[Chhugani *et al*., 2012] Jatin Chhugani, Changkyu Kim, Hemant Shukla, Jongsoo Park, Pradeep Dubey, John Shalf, and Horst D. Simon. Billion-particle SIMD-friendly two-point correlation on large-scale HPC cluster systems. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012.

[Chilingarian *et al*., 2004] I. Chilingarian, O. Bartunov, J. Richter, and T. Sigaev. PostgreSQL: The suitable DBMS solution for astronomy and astrophysics. In ASP Conference Series, 314, ADASS XIII, pages 225–228, 2004.

[Cohen and Guzman, 2006] S. Cohen, and D. E. Guzman. Sql.ct: Providing data management for visual exploration of ct datasets. In Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM) (2006).

[Colberg and Di Matteo, 2008] Colberg J. M. and Tizianna Di Matteo. Supermassive black holes and their environments. Monthly Notices of the Royal Astronomical Society (NMRAS) 387 (July 2008), 1163–1178.

[Couchman *et al*., 1996] H. M. P. Couchman, F. R. Pearce, and P. A. Thomas. Hydra code release, 1996.

[Davis *et al*., 1985] Davis M., Efstathiou G., Frenk C., and White S. The evolution of large scale structure in the universe dominated by cold dark matter. Astrophysical Journal 292 (1985), 371–394.

[Dean and Ghemawat, 2004] Jeff Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI), 2004.

[DeGraf *et al*., 2010a] Colin DeGraf, Tizianna Di Matteo, and Volker Springel. Faint-end quasar luminosity functions from cosmological hydrodynamic simulations. Monthly Notices of the Royal Astronomical Society (NMRAS) 402 (Mar. 2010), 1927–1936.

[Degraf *et al.*, 2010b] Colin DeGraf, Tizianna Di Matteo, and Volker Springel. Quasar Clustering in Cosmological Hydrodynamic Simulations: Evidence for mergers. ArXiv e-prints (May 2010).

[DeGraf *et al.*, 2012a] Colin DeGraf, Tiziana Di Matteo, Nishikanta Khandai, Rupert Croft, Julio López, and Volker Springel. Early Black Holes in Cosmological Simulations: Luminosity Functions and Clustering Behaviourar. Monthly Notices of the Royal Astronomical Society, Volume 424, Issue 3, pp. 1892-1898, 2012.

[DeGraf *et al.*, 2012b] Colin DeGraf, Tiziana Di Matteo, Nishikanta Khandai, and Rupert Croft. Growth of Early Supermassive Black Holes and the High-redshift Eddington Ratio Distribution. The Astrophysical Journal Letters, Volume 755, Issue 1, 2012.

[Dessup *et al.*, 2013] Tommy Dessup *et al*. Unnamed Transcript, 2013.

[Dikaiakos and Stadel, 1996] M. D. Dikaiakos and J. Stadel. A performance study of cosmological simulations on messagepassing and shared-memory multiprocessors. In ICS Conference (1996).

[Di Matteo *et al.*, 2005] Tizianna Di Matteo, Volker Springel, and Lars Hernquist. Energy input from quasars regulates the growth and activity of black holes and their host galaxies. Nature 433 (Feb. 2005), 604–607.

[Di Matteo *et al.*, 2008] Tiziana Di Matteo, Jörg Colberg, Volker Springel, Lars Hernquist, and Debora Sijacki. Direct cosmological simulations of the growth of black holes and galaxies. Astrophysical Journal, 676(2), 2008.

[Di Matteo *et al.*, 2012] Tiziana Di Matteo, Nishikanta Khandai, Colin DeGraf, Yu Feng, Rupert Croft, Julio López, and Volker Springel. Cold Flows and the First Quasars. The Astrophysical Journal Letters, Volume 745, Issue 2, 2012.

[Dolence and Brunner, 2008] Joshua Dolence and Robert J. Brunner. Fast Two-Point Correlations of Extremely Large Data Sets. In Proceedings of the 9th LCI International Conference on High-Performance Clustered Computing, 2008.

[Douglas and Douglas, 2003] Korry Douglas and Susan Douglas. PostgreSQL. SAMS Publishing, 2003.

[Eisenstein and Hut, 1998] D. J. Eisenstein, and P. Hut. Hop: A new group finding algorithm for N-body simulations. Astrophysical Journal (ApJ) 498 (1998), 137–142.

[Fan *et al.*, 2011] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. DiskReduce: Replication as a Prelude to Erasure Coding in Data-Intensive Scalable Computing. PDL Technical Report (CMU-PDL-11-112), 2011.

[Folk *et al.*, 1999] M. Folk, A. Cheng, and K. Yates. Hdf5: A file format and i/o library for high performance computing applications. In Proc. of Supercomputing (SC) (1999).

[Fu *et al.*, 2010] Bin Fu, Kai Ren, Julio López, Eugene Fink, and Garth Gibson. DiscFinder: A Data-Intensive Scalable Cluster Finder for Astrophysics. In Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, 2010.

[Fu *et al.*, 2012a] Bin Fu, Eugene Fink, Garth Gibson and Jaime Carbonell. Exact and Approximate Computation of a Histogram of Pairwise Distances between Astronomical Objects. First Workshop on High Performance Computing in Astronomy (AstroHPC 2012), held in conjunction with the 21st International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC 2012), June 19, 2012, Delft, the Netherlands.

[Fu *et al.*, 2012b] Bin Fu, Eugene Fink, Garth Gibson, and Jaime Carbonell. Indexing a large-scale database of astronomical objects. In Proceedings of the Fourth Workshop on Interfaces and Architecture for Scientific Data Storage, 2012.

[Galil and Italiano, 1991] Z. Galil, and G. F. Italiano. Data structures and algorithms for disjoint set union problems. ACM Comput. Surv. 23, 3 (1991), 319–344.

[Galler and Fischer, 1964] B. Galler and M. Fischer. An improved equivalence algorithm. Communications of the ACM (CACM) 7, 5 (1964), 301–303.

[Gardner *et al.*, 2006] Jeffrey P. Gardner, Andrew Connolly, and Cameron McBride. A framework for analyzing massive astrophyisical datasets on a distributed grid. In Proc. conf. on Astronomical Data Analysis & Software Systems (ADASS XVI) (2006).

[Gelb and Bertschinger, 1994] James M. Gelb and Edmund Bertschinger. Cold dark matter 1: The formation of dark halos. Astrophysical Journal, 436 (1994), 467–490.

[Ghemawat *et al.*, 2003] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In Proc. 19th Symposium on Operating Systems Principles (SOSP'03) (New York, NY, USA, 2003), ACM, pp. 29–43.

[Gill *et al.*, 2004] Stuart P. D. Gill, Alexander Knebe, and Brad K. Gibson. The evolution of substructure - I. A new identification method. Monthly Notices of the Royal Astronomical Society (NMRAS) 351 (June 2004), 399–409.

[Gott *et al.*, 2005] J. Richard Gott III, Mario Jurić, David Schlegel, Fiona Hoyle, Michael Vogeley, Max Tegmark, Neta Bahcall, and Jon Brinkmann. A Map of the Universe. The Astrophysical Journal, Volume 624, pages 463–484, 2005.

[Gottloeber, 1997] S. Gottloeber. Galaxy tracers in cosmological n-body simulations. In Proceedings of the Potsdam Cosmology Workshop: Large Scale Structure: Tracks and Traces (Sept 1997).

[Gray and Moore, 2000] Alexander Gray and Andrew Moore. “N-body” problems in statistical learning. Advances in Neural Information Processing Systems 13, pages 521–527. MIT Press, 2000.

[Gray and More, 2003] Alexander Gray and Andrew Moore. Nonparametric density estimation: Toward computational tractability. In Proceedings of the Third SIAM International conference on Data Mining, 2003.

[Guttman, 1984] Antonin Guttman. R-Trees: A dynamic index structure for spatial searching. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 47–57, 1984.

[Heitmann *et al*., 2008a] Katrin Heitmann, Martin White, Christian Wagner, Salman Habib, and David Higdon. The Coyote Universe I: Precision determination of the nonlinear matter power spectrum. Astrophysical Journal, 715(1), 2008.

[Heitmann *et al*., 2008b] Katrin Heitmann, Zarija Lukic, Patricia Fasel, Salman Habib, Michael S. Warren, Martin White, James Ahrens, Lee Ankeny, Ryan Armstrong, Brian O’Shea, Paul M. Ricker, Volker Springel, Joachim Stadel, and Hy Trac. The cosmic code comparison project. Computational Science and Discovery 1, 1 (Oct. 2008), 015003.

[Heitmann *et al*., 2010] Katrin Heitmann, Martin White, Christian Wagner, Salman Habib, and David Higdon. The Coyote Universe. I. Precision Determination of the Nonlinear Matter Power Spectrum. The Astrophysical Journal, Volume 715, pages 104–121, 2010.

[Holmes *et al*., 1994] Geoffrey Holmes, Andrew Donkin, and Ian H. Witten. Weka: A machine learning workbench. In Proceedings of the Second Australia and New Zealand Conference on Intelligent Information Systems, 1994.

[Huchra and Geller, 1982] J. P. Huchra and M. J. Geller. Groups of galaxies. I - Nearby groups. Astrophysical Journal (ApJ) 257 (June 1982), 423–437.

[Isard *et al*., 2007] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, pages 59–72, 2007.

[Ivanova *et al*., 2007] M. Ivanova, N. Nes, R. Goncalves, and M. Kersten. Monetdb/sql meets skyserver: the challenges of a scientific database. In Proc. of the 19th International Conference on Scientific and Statistical Database Management (SSDBM 2007) (2007), p. 13.

[Ivezic *et al*., 2008] Zeljko Ivezic *et al*. LSST: from science drivers to reference design and anticipated data products. http://www.lsst.org/lsst/science/overview, 2008.

[Kaiser *et al*., 2002] Nick Kaiser *et al*. Pan-STARRS: A Large Synoptic Survey Telescope Array. In Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series (Dec.

2002), J. A. Tyson & S.Wolff, Ed., vol. 4836 of Society of Photo-Optical Instrumentation Engineers (SPIE) Conference, pp. 154–164.

[Kaustav *et al*., 2008] Kaustav Das, Jeff Schneider, and Daniel B. Neill. Anomaly pattern detection in categorical datasets. In Proc. 14th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2008) (August 2008).

[Khandai *et al.*, 2012] Nishikanta Khandai, Yu Feng, Colin DeGraf, Tiziana Di Matteo, and Rupert Croft. The formation of galaxies hosting z˜6 quasars. Monthly Notices of the Royal Astronomical Society, Volume 423, Issue 3, pp. 2397–2406, 2012.

[Kwon *et al*., 2009] YongChul Kwon, Dylan Nunley, Jeffrey P. Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. Scalable clustering algorithm for n-body simulations in a shared-nothing cluster. Tech. Rep. UWCSE-09-06-01, University of Washington, June 2009.

[Kwon *et al*., 2010] YongChul Kwon, Dylan Nunley, Jeffrey P. Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. Scalable clustering algorithm for n-body simulations in a shared-nothing cluster. In Proceedings of the 22nd Scientific and Statistical Database Management (SSDBM) (2010).

[Lakshman and Malik, 2010] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review archive, Volume 44 Issue 2, April 2010, Pages 35–40.

[Lasker *et al*., 2008] Barry Lasker *et al*. The second-generation Guide Star Catalog: Description and properties. The Astrophysical Journal, 136, pages 735–766, 2008.

[Lee and Wong, 1977] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. Acta Informatica, 9 (1977), 23–29.

[Lemson and Springel, 2006] Gerard Lemson and Volker Springel. Cosmological simulations in a relational database: Modelling and storing merger trees. In Astronomical Data Analysis Software and Systems (2006), No. XV in ASP Conference Series, p. 212.

[Leobman *et al*., 2009] Sarah Loebman, Dylan Nunley, YongChul Kwon, Bill Howe, Magdalena Balazinska, and Jeffrey P. Gardner. Analyzing massive astrophysical datasets: Can pig/hadoop or a relational dbms help? In Proc. IASDS (2009).

[Lependu *et al.*, 2008] Paea LePendu, Dejing Dou, Gwen A. Frishkoff, and Jiawei Rong. Ontology database: A new method for semantic modeling and an application to brainwave data. In Scientific and Statistical Database Management (2008), vol. 5069, pp. 313–330.

[Lintott *et al*., 2008] Chris J. Lintott, Kevin Schawinski, Anže Slosar, Kate Land, Steven Bamford, Daniel Thomas, M. Jordan Raddick, Robert C. Nichol, Alex Szalay, Dan Andreescu, Phil Murray, Jan Vandenberg. Galaxy Zoo: morphologies derived from visual inspection of galaxies from the Sloan Digital Sky Survey. Monthly Notices of the Royal Astronomical Society, Volume 389, Issue 3, pp. 1179–1189.

[Liu *et al*., 2003] Ying Liu, Wei-keng Liao, and Alok Choudhary. A. Design and evaluation of a parallel hop clustering algorithm for cosmological simulation. Proc. Parallel and Distributed Processing International Symposium (April 2003), 8 pp.

[Liu *et al*., 2008] Hauyu Baobab Liu, B. C. Hsieh, Paul T. P. Ho, Lihwai Lin, and Renbin Yan. A new galaxy group finding algorithm: Probability friends-of-friends. Astrophysical Journal (ApJ) 681, 2 (2008), 1046–1057.

[Low *et al*., 2011] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin and J. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. In the 26th Conference on Uncertainty in Artificial Intelligence (UAI), Catalina Island, USA, 2010.

[López *et al*., 2011] Julio López, Colin Degraf, Tiziana DiMatteo, Bin Fu, Eugene Fink and Garth Gibson. Recipes for Baking Black Forest Databases: Building and Querying Black Hole Merger Trees from Cosmological Simulations. In Proceedings of the 23rd International Conference Scientific and Statistical Database Management (SSDBM 2011).

[March *et al*., 2012] William B. March, Andrew J. Connolly, and Alexander G. Gray. Fast Algorithms for Comprehensive N-point Correlation Estimates. In Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Beijing, China, 2012.

[Morton, 1966] G. M. Morton. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. IBM Germany Scientific Symposium Series, 1966.

[MPI, 1993] MPI Forum. MPI: A Message Passing Interface. In Proceedings of the ACM/IEEE conference on Supercomputing, pages 878–883, 1993

[Olston *et al*., 2008] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign languge for data processing. In Proceedings of the SIGMOD Conference (2008), pp. 1099–1110.

[PanStarrs] PanStarrs Consortium. Panoramic survey telescope & rapid response system (PanSTARRS). http://pan-starrs.ifa.hawaii.edu.

[Peebles, 1980] Jim Peebles. The Large Scale Structure of the Universe. Princeton University Press, 1980.

[Pfitzner and Salmon, 1996] David W. Pfitzner and John K. Salmon. Parallel halo finding in N-body cosmology simulations. In 2nd Conference on Knowledge Discovery and Data Mining (KDD-96) (1996), AAAI Press, pp. 26–31.

[Pfitzner *et al.*, 1997] David W. Pfitzner, John K. Salmon and Thomas Sterling. HaloWorld: Tools for parallel cluster finding in astrophysical N-body simulations. Data Mining and Knowledge Discovery 1, 4 (December 1997), 419–438.

[Pirenne and Ochsenbein, 1991] B. Pirenne and F. Ochsenbein. The Guide Star Catalogue in STARCAT. Space Telescope European Coordinating Facility Newsletter, 15, pages 17–19, 1991.

[Ponce *et al.*, 2011] Rafael Ponce, Miguel Cardenas-Montes, Juan Jose Rodriguez-Vazquez, Eusebio Sanchez, and Ignacio Sevilla. Application of GPUs for the Calculation of Two Point Correlation Functions in Cosmology. In Proceedings of the 21st Astronomical Data Analysis Software and Systems (ADASS XXI), Paris, 2011.

[Quinlan, 1992] J.R. Quinlan. C4.5: Programs for machine learning. Morgan Kaufmann, 1992

[Reichman *et al.*, 2011] O. J. Reichman, M. B. Jones, M. P. Schildhauer. Challenges and Opportunities of Open Data in Ecology. Science, Volume 331, No. 6018, pages 703–705.

[Rew and Davis, 1990] R. K. Rew and G. P. Davis. The unidata netcdf: Software for scientific data access. In Proc. of the Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology (1990), pp. 33–40.

[Richards *et al.*, 2002] Gordon T. Richards et al., Spectroscopic target selection in the Sloan digital sky survey: The quasar sample. The Astronomical Journal, 123:2945, 2002.

[Richards *et al.*, 2004] Gordon T. Richards et al., Efficient photometric selection of quasars from the Sloan digital sky survey: 100,000 z<3 quasars from Data Release One. The Astrophysics Journal Supplement Series, 155:257–269, 2004.

[Richards *et al.*, 2009] Gordon T. Richards et al., Efficient photometric selection of quasars from the Sloan digital sky survey: II ~1,000,000 quasars from Data Release Six. The Astrophysics Journal Supplement Series, 180:67, 2009.

[Rigaux *et al.*, 2002] Philippe Rigaux, Michel Scholl and Agnes Voisard. Spatial Databases: With Application to GIS. Morgan Kaufman, Los Altos, 2002.

[Samet, 1990] Hanan Samet. The Design and Analysis of Spatial Data Structures. Addison-Wesley, Reading, MA, 1990.

[Sandage and Wyndham, 1965] Allan Sandage and John D. Wyndham. On the Optical Identification of Eleven New Quasi-Stellar Radio Sources. The Astrophysics Journal, 141:328–332, 1965.

[Schlosser *et al.*, 2008] Steven W. Schlosser, Michael P. Ryan, Ricardo Taborda, Julio López, David R. O'Hallaron, and Jacobo Bielak. Materialized community ground models for large-scale earthquake simulation. In Proc. Conference on Supercomputing (SC'08) (Piscataway, NJ, USA, 2008), ACM/IEEE, IEEE Press, pp. 1–12.

[Settles, 2010] Burr Settles. Active Learning Literature Survey. Computer Science Technical Report 1648, University of Wisconsin-Madison, 2010.

[Springel and Hernquist, 2002] Volker Springel and Lars Hernquist. Cosmological smoothed particle hydrodynamics simulations: the entropy equation. Monthly Notices of the Royal Astronomical Society (NMRAS) 333 (July 2002), 649–664.

[Springel and Hernquist, 2003] Volker Springel and Lars Hernquist. Cosmological smoothed particle hydrodynamics simulations: a hybrid multiphase model for star formation. Monthly Notices of the Royal Astronomical Society (NMRAS) 339 (Feb. 2003), 289–311.

[Springel, 2005] Volker Springel. The cosmological simulation code GADGET-2. Monthly Notices of the Royal Astronomical Society (NMRAS) 364 (Dec. 2005), 1105–1134.

[Springel *et al*., 2005] Volker Springel, Tiziana Di Matteo, and Lars Hernquist. Modelling feedback from stars and black holes in galaxy mergers. Monthly Notices of the Royal Astronomical Society (NMRAS) 361 (Aug. 2005), 776–794.

[Springel *et al*., 2008] Volker Springel, Jie Wang, Mark Vogelsberger, Aaron Ludlow, Adrian Jenkins, Amina Helmi, Julio F. Navarro, Carlos S. Frenk, and Simon D. M. White. M. The Aquarius Project: the subhaloes of galactic haloes. Monthly Notices of the Royal Astronomical Society (NMRAS) 391 (Dec. 2008), 1685–1711.

[Stonebraker *et al*., 2007] Michael Stonebraker, Chuck Bear, Uğur Çetintemel, Mitch Cherniack, Tingjian Ge, Nabil Hachem, Stavros Harizopoulos, John Lifter, Jennie Rogers, and Stan Zdonik. One Size Fits All? – Part 2: Benchmarking Results. In Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR), 2007

[Swan, 2013] Melanie Swan. The quantified self: Fundamental Disruption in Big Data Science and Biological Discovery. Big Data. June 2013: 85–99.

[Szalay *et al*., 2002] Alexander S. Szalay, Tamas Budavari, Andrew Connolly, Jim Gray, Takahiko Matsubara, Adrian Pope, and Istvan Szapudr. Spatial clustering of galaxies in large datasets. Astronomical Data Analysis II 4847, 1 (2002), 1–12.

[Szalay *et al*., 2005] Alex Szalay, Jim Gray, Gyorgy Fekete, Peter Kunszt, Peter Kukol, and Ani Thakar. Indexing the sphere with the hierarchical triangular mesh, 2005. Technical Report MSR-TR-2005-123.

[Tarjan, 1975] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. Journal of ACM 22, 2 (1975).

[Tong and Koller, 2001] Simon Tong and Daphne Koller. Support vector machine active learning with applications to text classification. Journal of Machine Learning Research, volume 2, pages 45–66.

[Tu *et al.*, 2002] Tiankai Tu, David R. O'hallaron, and Julio C. López. Etree – a database-oriented method for generating large octree meshes. In Proceedings of the Eleventh International Meshing Roundtable (Ithaca, NY, Sep 2002), pp. 127– 138.

[Tu *et al.*, 2003] Tiankai Tu, David R. O'hallaron, and Julio C. López. The Etree library: A system for manipulating large octrees on disk. Tech. Rep. CMU-CS-03-174, Carnegie Mellon School of Computer Science, Pittsburgh, PA, July 2003.

[UW-aFOF] University of Washington N-Body Shop. aFOF: Approximate group finder for N-body simulations. www-hpcc.astro.washington.edu/tools/afof.html.

[UW-FOF] University of Washington N-Body Shop. FOF: Find groups in N-body simulations using the friends-of-friends method. www-hpcc.astro.washington.edu/tools/fof.html.

[UW-Skid] University of Washington N-Body Shop. Skid: A tool to find gravitationally bound groups in n-body simulations. www-hpcc.astro.washington.edu/tools/skid.html.

[Ward *et al.*, 2013] R. Matthew Ward, Robert Schmieder, Gareth Highnam, and David Mittelman. Big data challenges and opportunities in high-throughput sequencing. Systems Biomedicine 2013, Volume 1, Issue 1, pages 23–28.

[Wells *et al.*, 1981] D. C. Wells, E. W. Greisen, and R. H. Harten. FITS - a flexible image transport system. Astronomy and Astrophysics Supplement Series, 44 (1981), 363.

[White, 2002] Martin White. The mass function. The Astrophysical Journal 143 (2002), 241.

[Wicenec and Albrecht, 1998] A. J. Wicenec and M. Albrecht. Methods for structuring and searching very large catalogs. Astronomical Data Analysis Software and System VII, ASP Conference Series, 145, 1998

[Tantisiriroj *et al.*, 2011] Wittawat Tantisiriroj, Seung Woo Son, Swapnil Patil, Samuel J. Lang, Garth Gibson, Robert B. Ross. In Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, 2011.

[Xu *et al.*, 2009] Weijia Xu, Stuart Ozer, and Robin R. Gutell. Covariant evolutionary event analysis for base interaction prediction using a relational database management system for RNA. In Proc. 21th Conf. on Scientific and Statistical Database Management (SSDBM) (2009).

[Yang *et al.*, 2007] Yuan-Sen Yang, Shang-Hsien Hsieh, Keh-Chyuan Tsai, Shiang-Jung Wang, Kung-Juin Wang, Wei-Choung Cheng, and Chuan-Wen Hsu. ISEE: Internet-based simulation for earthquake engineering - part i: Database approach. Earthquake Engineering & Structural Dynamics 36 (2007), 2291–2306.

[Zaharia *et al*., 2010] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. USENIX HotCloud 2010. June 2010.

[Zhu, 2005] Xiaolin Zhu. Semi-Supervised Learning Literature Survey. Computer Science Technical Report 1530, University of Wisconsin-Madison, 2005.