Multi-versioned Data Storage and Iterative Processing in a Parallel Array Database Engine

Emad Soroush

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2014

Reading Committee:

Magdalena Balazinska, Chair

Dan Suciu

Bill Howe

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Multi-versioned Data Storage and Iterative Processing in a Parallel Array Database Engine

Emad Soroush

Chair of the Supervisory Committee:
Associate Professor Magdalena Balazinska
Computer Science and Engineering

Scientists today are able to generate data at an unprecedented scale and rate. For example the Sloan Digital Sky Survey (SDSS) generates 200GB of data containing millions of objects on each night on its routine operation. The large hadron collider is producing even more data today which is approximately 30PB annually. The Large Synoptic Survey Telescope (LSST) also will be producing approximately 30TB of data per night in a few years. Also, in many fields of science, multidimensional arrays rather than flat tables are standard data types because data values are associated with coordinates in space and time. For example, images in astronomy are 2D arrays of pixel intensities. Climate and ocean models use arrays or meshes to describe 3D regions of the atmosphere and oceans. As a result, scientists need powerful tools to help them manage massive arrays.

This thesis focuses on various challenges in building parallel array data management systems that facilitate massive-scale data analytics over arrays.

The first challenge with building an array data processing system is simply how to store arrays on disk. The key question is how to partition arrays into smaller fragments called chunks that form the unit of IO, processing, and data distribution across machines in a cluster. We explore this question in ArrayStore, a new read-only storage manager for parallel array processing. In ArrayStore, we study the impact of different chunking strategies on query processing performance for a wide range of operations, including binary operators and user-defined functions. ArrayStore also proposes two

new techniques that enable operators to access data from adjacent array fragments during parallel processing.

The second challenge that we explore in building array systems is the ability to create, archive, and explore different versions of the array data. We address this question in TimeArr, a new append-only storage manager for an array database. Its key contribution is to efficiently store and retrieve versions of an entire array or some sub-array. To achieve high performance, TimeArr relies on several techniques including virtual tiles, bitmask compression of changes, variable-length delta representations, and skip links.

The third challenge that we tackle in building parallel array engines is how to provide efficient iterative computation on multi-dimensional scientific arrays. We present the design, implementation, and evaluation of ArrayLoop, an extension of SciDB with native support for array iterations. In the context of ArrayLoop, we develop a model for iterative processing in a parallel array engine. We then present three optimizations to improve the performance of these types of computations: incremental processing, mini-iteration overlap processing, and multi-resolution processing.

Finally, as motivation for our work and also to help push our technology back into the hands of science users, we have built the AscotDB system. AscotDB is a new, extensible data analysis system for the interactive analysis of data from astronomical surveys. AscotDB provides a compelling and powerful environment for the exploration, analysis, visualization, and sharing of large array datasets.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

DEDICATION

To my parents for their endless support and encouragement over the years

Chapter 1

INTRODUCTION

Today's Web-based companies such as Google, Microsoft, Facebook, and others are growing in popularity. These companies commonly accumulate logs from monitoring how their services are being used (*i.e.*, streams of search queries, click streams, low-level network flows, etc). Mining all this monitoring data can help companies provide better services to their users: from personalized product recommendations to quality-of-service assessment, and sophisticated product design and marketing strategies. While several systems exist for large-scale data analytics (*e.g.*, parallel database systems, MapReduce-type systems), each tool satisfies only a subset of today's data analysis needs. As a result, all major players are developing new tools and platforms for data analytics [13, 44, 57, 70].

Interestingly, this trend is not restricted to businesses. Sciences are also increasingly becoming data-driven. From small research labs to large communities [39, 54], scientists across a variety of disciplines including astronomy, biology, physics, oceanography, and climatology are all dealing with large-scale data analytics. For example, the Sloan Digital Sky Survey (SDSS) [95] generates 200GB of data each night on its routine operation. Additionally, the next generation of telescopic sky surveys such as the Large Synoptic Survey Telescope (LSST) [54] will generate 30TB of imagery data every night or up to a hundred Petabytes of data for the duration of the survey. A large team of developers is building a specialized data analysis pipeline to handle this data onslaught. As other examples, the Earth Microbiome Project [26] expects to produce 2.4 petabytes in their metagenomics effort and the Large Hadron Collider [11] generates data which is approximately 30 petabytes annually. Because of the size of the data they need to analyze, scientists today can increasingly benefit from using data management systems to organize and query their data.

1.1 Scientific Application Requirements

Scientists with extreme data base requirements complain about the inadequacy of modern data processing tools [101]. Many advocate that one should move away from the relational model and adopt a multidimensional array data model [28, 102]. The main reason is that, in many fields of science, multidimensional arrays rather than flat tables are standard data types because data values are associated with coordinates in space and time. For example, images in astronomy are 2D arrays of pixel intensities. Climate and ocean models use arrays or meshes to describe 3D regions of the atmosphere and oceans. They simulate the behavior of these regions over time by numerically solving the governing equations. Cosmology simulations model the behavior of clusters of 3D particles to analyze the origin and evolution of the universe. One approach to managing this type of array data is to build array libraries on top of relational engines, but many argue that simulating arrays on top of relations can be highly inefficient [79, 102]. Scientists also need to perform a variety of operations on their array data such as feature extraction [49], smoothing [87], and cross-matching [66], which are not built-in operations in relational Database Management Systems (DBMSs). Those operations also impose different requirements than relational operators on a data management engine. As a result, many engines are being built today to support multidimensional arrays natively [3, 28, 87, 110].

To illustrate the need for processing arrays and performing array-oriented operations, consider the following example from the astronomy domain:

Example 1.1.1. Consider the LSST image database. Telescope images are 2D arrays of pixels. When taken over time, the images form a 3D array of pixel intensities. Three types of analysis are commonly performed on this data:

When analyzing telescope images, some sources (a "source" can be a galaxy, a star, etc.) are too faint to be detected in one image but can be detected by stacking multiple versions of images from the same location on the sky. The stacking of LSST images is called *co-addition*. Figure 1.2(a) and 1.2(b) illustrate a single LSST image and its corresponding co-added image. More objects are visible in the co-added image. LSST will undertake repeated exposures over ten years with each image partially overlapping with hundreds of others. Co-addition of LSST images involves *grouping* all the pixel values from the same location on the sky followed by some *aggregate* computation. To enable efficient stacking and comparison of these images, the data can benefit from being partitioned and

grouped on array dimensions. While pipelines are being designed by the LSST team to handle this image processing task and create catalogs of detected objects, the truly transformative science will come from providing scientists with the ability to directly query basic operations on the pixel-level raw data, and to enable interactive and exploratory computation and visualization of that data.

Before the co-addition is applied, astronomers often run a *sigma-clipping* noise-reduction algorithm. The analysis in this case has two steps: (1) outlier filtering with "sigma-clipping" and then (2) image co-addition. The "sigma-clipping" algorithm consists in grouping all pixels on their sky coordinates. For each location, the algorithm computes the mean and standard deviation of the flux (light intensity). It then sets to null all cell values that lie a user-specified number of standard deviations away from the mean. The algorithm iterates by re-computing the mean and standard deviation. The cleaning process terminates once no new cell values are filtered out. Figure 1.2(c) represents the effect of "sigma-clipping" before stacking images. The filtering process removes noises that are a side-effect of the LSST pipeline. The "sigma clipping" analysis involves a wide set of operations. This analysis not only illustrates the need for efficient implementation of basic operations such as *filtering*, *grouping*, and *join*, but also more complex *iterative* computations on arrays.

Once telescope images have been cleaned and co-added, the next step is typically to extract the actual sources from the images. *Source detection* is a time consuming operation that should be parallelized by breaking down the large raw image into multiple smaller images. This algorithm is often implemented as a user-defined function that involves a full scan of the raw data followed by a grouping phase that groups observations detected in different partitions into ones that represent the same object. Another way to implement this algorithm is to run it in an iterative fashion. An iterative computation can be expensive to execute but it is easier to express (only requires grouping and aggregation) and also avoids the overhead of the final cross-partition merging phase. In a simplified version of the iterative source detection algorithm, each non-empty cell is initialized with a unique label and is considered to be a different object. At each iteration, each cell resets its label to the minimum label value across its neighbors. Two cells are neighbors if they are adjacent. This procedure continues until the algorithm converges and no more cell values change. A key question is how to access adjacent cells that fall outside a partition boundary and can thus be stored on a different

machine. A nice solution is to provide each partition with a margin of overlapping data along its boundaries. The challenges, however, are how to select the appropriate amount of data overlap when partitioning the array and how to keep the overlap data up to date. This source detection analysis illustrates the need for properly handling parallel array computations that involve overlapped data and iterations. Figure 1.2(d) shows the result of running the source detection algorithm on the example LSST image. □

Overall, we argue that a data management system for large-scale scientific data analytics should meet the following requirements:

Support Structural Information at Scale: Scientists typically work with multidimensional measurements such as arrays and meshes, because their data values are associated with coordinates in space and time. Data management systems for scientific applications should thus provide efficient support for array data and powerful analytics on that data. Additionally, those engines should adopt effective techniques for parallel processing to bring scalability. To scale query processing, data management systems often partition and process data in a shared-nothing cluster. However, this is challenging to provide for complex data types such as arrays due to the data dependency that exists in structural information (e.g., the source detection application from example 1.1.1 operates on neighborhoods of cells and thus requires overlap data).

Support No Overwrite: Today's scientific organizations commonly collect data over time (e.g., time series data or different versions of their data). An important requirement that scientists have for any kind of data processing engine is the ability to create, archive, and explore different versions of their data [102]. Hence, a no-overwrite storage manager with efficient support for querying old versions of the data is a critical component of any large-scale data management system. There is a long line of research on this topic, from temporal databases [46, 52, 75, 96], conventional version-control systems [12], to video compression codecs [17]. However, none of those efficiently supports all aspects of *creating*, *archiving*, and *exploring* different versions of an array.

Support Iterative Applications: Many data analysis tasks today require iterative processing. Current distributed data processing platforms such as parallel DBMSs and Hadoop [36] do not have

built-in support for iterative programs, which makes it difficult for them to support these types of operations. Several systems have been developed that support iterative big data analytics [10, 30, 53, 93, 109]. For example Twister [27], Daytona [4], and HaLoop [10] extend MapReduce to preserve state across iterations and to provide support for a looping construct. However, none of these is an array engine. Modern data processing engines must treat iterative computations as first-class citizens.

Support Interactive and Exploratory Analytics: A truly transformative data analytics tool should provide a compelling and powerful environment for the exploration, analysis, visualization, and sharing of large datasets. We regard data analytics as a cycle consisting of a sequence of exploration and analysis phases. Exploration is well supported with graphical interfaces while analysis is well supported with programmatic interfaces. In order to maintain the cycle of data exploration and analytics, it is crucial to provide a seamless interaction between those two phases. As a result, a rich set of graphical and programmatic interfaces that supports an interactive and seamless switching context is essential. Although there are some initial efforts [5] that emphasize the graphical and programmatic interfaces as first-class citizens for data analytics tools, none yet provides a seamless interaction between exploratory and analysis phases.

Today's DBMSs do not handle these requirements well. Existing DBMSs are based on the relational model, which is inefficient for the types of data used for complex data analytics [79]. They also do not sufficiently scale [40]. Data warehousing solutions based on the MapReduce framework [23] such as Hive [40] are an alternate solution. They are not the final answer to data analytics problems today, though, because they are not naturally designed to process data with important structural information. For example, one of the algorithms commonly used across a variety of disciplines is k-nearest neighbors. The source detection algorithm on LSST images (Example 1.1.1) from the astronomy domain is another example. Those algorithms are hard to execute efficiently in MapReduce because of the lack of natural support for locality information.

Array engines strive to satisfy this set of requirements. They naturally provide efficient support for structural information both in terms of storage and data access by better capturing the multi-dimensional nature of the measurement data. In array engines, structural information is associated with each cell through its dimension values. Those dimensions provide a natural index for the data.

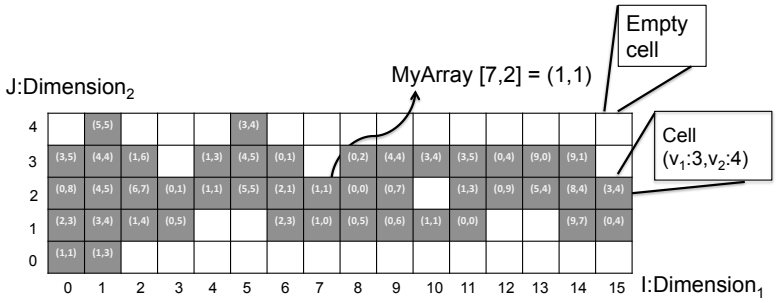
Array engines also help the user through the data exploration and analysis by providing a rich set of built-in operators, including common relational operators such as *Select*, *Filter*, and *Join*, and array-specific operators such as *Slicing*, *Dicing*, and Regrid [87]. Consider the LSST images example (Example 1.1.1): Array engines are a good candidate to store those images. Storing the LSST images in a large multidimensional array with sky and time dimensions can yield significant savings in storage space since sky dimensions are not stored. It also provides efficient indexing of pixel positions both on the sky and in time. Array-engines can help through the noise reduction process in the LSST images as well. All the grouping, aggregate computations, and subtraction of outlier cells are based on the sky and time coordinates which are implicitly *indexed* by storing and processing images as a multidimensional array. Array engines provide parallel processing by using data partitioning and overlap processing techniques.

Array engines capabilities are very promising, however, the research challenge is to understand *effective* partitioning strategies in the context of a complex query workload, comprising not only range-queries, but also binary operations such as joins and complex user-defined functions, and to develop *efficient* techniques that provide fast access to neighboring cells. The latter is challenging especially for boundary cells of already partitioned data. Fast access to neighboring cells is vital for algorithms such the source detection in the LSST images where access to adjacent cells is a fundamental unit of computation. Efficient support for iterative applications and efficient support for versioning array data are two requirements that are not well-studied in the context of array engines and require major improvements.

1.2 SciDB: Open-source DBMS with Inherent Support for Array Processing

SciDB [87] is an open-source parallel database management system where the core data model is a multi-dimensional array. SciDB supports arrays *natively*. That is, SciDB is designed and implemented from the ground-up based on an array data model rather than being layered on top of an existing DBMS [3, 19, 28, 110]. Figure 1.1 illustrates an example array, called `MyArray`, that is supported by SciDB. `MyArray` is a 2D (16×5) array with two integer dimensions I and J and two attributes v_1 and v_2. Each permutation of coordinates specifies a cell in the array. Cells can be empty. Each non-empty cell contains a set of attribute-values. `MyArray` in Figure 1.1 is defined as follow: `Array MyArray <`v_1`: double, `v_2`: double>[int `I`=0:15,16,0,`

Figure 1.1 Scidb array data model example: `MyArray` is a 2D (16×5) array with two integer dimensions I and J and two double attributes v_1 and v_2.



`int` J=`0:4,5,0]`. The first parameter in each dimension specifies starting index and ending index, e.g. $0 : 15$ in dimension I in `MyArray`. To store arrays on disk, SciDB partitions them into sub-arrays called *chunks*. The second parameter specifies the chunk size in that dimension. In this example `MyArray` has only one chunk. The third parameter defines the margin of overlapped data between two chunks. `MyArray` has no overlap data. Dimensions in SciDB can be integer or non-integer. In case dimensions are non-integers, such as strings, SciDB internally uses a mapping function from the source dimension type to integers. Attributes in SciDB can be atomic types, user-defined types, or even other arrays. The latter are called *nested* arrays. For further information about SciDB, we refer the reader to the SciDB overview paper [87].

1.3 Thesis Outline and Contributions

This thesis focuses on *parallel array data management systems*, which hold the promise to be well-suited for large-scale, complex array analytics. In the context of building an array-based system to facilitate massive-scale data analytics, the contributions of this thesis are as follows:

Array Data Processing and Storage: A critical component of making an array data processing system is to build an efficient storage layer, similar to the Hadoop Distributed File System (HDFS) which is a key component of the Hadoop data processing system. A standard approach to storing an array is to partition it into sub-arrays called chunks. Each chunk is typically the size of a storage block. Chunking an array helps alleviate "dimension dependency" [94], where the number of blocks read from disk depends on the dimensions involved in a range-selection query rather than just the

range size. The challenge of building a storage manager for arrays is to choose the appropriate chunk shape and structure. Do they need to be fixed in terms of number of bytes or they need to have a fixed volume shape? What is the appropriate chunk size for a given workload? Should the system use single level chunking or hierarchical chunking? Prior work [14, 61, 94] studies the tuning of chunk shape, size, and layout on disk and across disks for range-selection queries. We refer the reader to Section 6.2 for a more detailed discussion of related work. In contrast, in Chapter 2, we study the impact of different chunking strategies on query processing performance for a wide range of operations, including binary operators and user-defined functions. In Chapter 2, we present the design, implementation, and evaluation of ArrayStore [99], a storage manager for complex, parallel array processing. For efficient processing, ArrayStore partitions an array into chunks and we show that a two-level chunking strategy with regular chunks (fixed volume) and regular sub-chunks, called tiles, leads to the best and most consistent performance for a varied set of operations both on a single node and in a shared-nothing cluster. ArrayStore also enables operators to access data from adjacent array fragments during parallel processing. We present two new techniques to support this need: one leverages ArrayStore's two-level storage layout and the other one uses additional materialized views. Both techniques cut runtimes in half in our experiments compared with state-of-the-art alternatives.

Array Data Versioning: As discussed above, a storage manager for an array engine must provide efficient support for different versions of the array data. This feature can naturally be supported in an array-based engine. For example to support no overwrite, an array database can simply add a hidden "time" dimension to every array. Data is loaded into the array at the time indicated in the loading process. Subsequent updates, inserts, or bulk loads add new data at the time they are running, without discarding the previous information. Hence, for a given cell, moving along the time dimension will indicate the sequence of updates to the cell. A common use-case for data versions is simply the ability to point at a data value or a collection of values and say "show me the history of this data". In many applications, the number of updates and subsequently the length of the "time" dimension is not known in advance. Also many updates only touch small fraction of the whole array. As a result, in building an append-only array-based engine, we have to address challenging questions such as: What is the right chunk representation for a versioned array? How to efficiently support updates and data versioning in array chunks? How to efficiently support queries for the history of a sub-array? We

address those questions in Chapter 3, where we present the design, implementation, and evaluation of TimeArr [98], a new storage manager for an array database. Its key contribution is to efficiently store and retrieve versions of an entire array or some sub-array. TimeArr also introduces the idea of approximate exploration of an array's history. To achieve high performance, TimeArr relies on several techniques including virtual tiles, bitmask compression of changes, variable-length delta representations, and skip links. TimeArr enables users to customize their exploration by specifying both the maximum degree of approximation tolerable and how it should be computed. Experiments with a prototype implementation and two real datasets from the astronomy and earth science domains demonstrate up to a factor of six performance gain with Timarr's approach compared to the naive versioning system in SciDB.

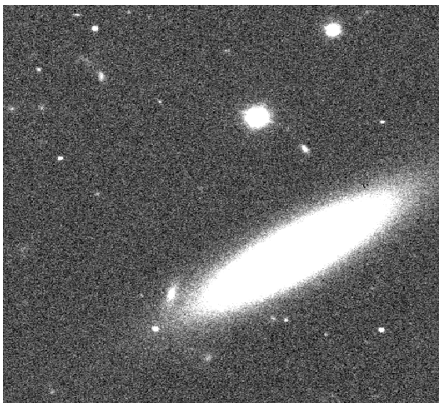
Iterative Array Processing: Many data analysis tasks today require iterative processing: machine learning, model fitting, pattern discovery, flow simulations, cluster extraction, and more. The need for efficient iterative computation extends to analysis executed on multi-dimensional scientific arrays. While it is possible to implement iterative array computations by repeatedly invoking array queries from a script, this approach is highly inefficient. To support these iterative tasks efficiently, array engines such as SciDB should have native support for array iterations. In Chapter 4, we present the design, implementation, and evaluation of ArrayLoop [62], an extension of SciDB with native support for array iterations. In the context of ArrayLoop, we develop a model for iterative processing in a parallel array engine. We then present three optimizations to improve the performance of these types of computations: incremental processing, mini-iteration overlap processing, and multi-resolution processing. Experiments with 1 TB of publicly available LSST images [83] show that our optimizations can significantly improve runtimes by up to 4X for real queries on LSST data

AscotDB: As motivation for our work and also to help push our technology back into the hands of science users, we have built the AscotDB [62, 106] system. AscotDB is a new, extensible data analysis system for the interactive analysis of data from astronomical surveys. In Chapter 5, we present the design and implementation of AscotDB. AscotDB is a layered system: It builds on SciDB to provide a shared-nothing, parallel array processing and data management engine. To enable both exploratory and deep analysis of the data, AscotDB provides both graphical and programmatic (the

latter is not a contribution of this thesis) interfaces with seamless switching between these two modes.

In summary, in the context of the AscotDB project and also motivated by other array-processing applications, this thesis work focuses on the following critical challenges for a parallel array management system: 1) Efficient storage management mechanisms to store arrays on disk (ArrayStore, Chapter 2). 2) Efficient support for updates and data versioning (TimeArr, Chapter 3). 3) Native support for efficient iterative computations (ArrayLoop, Chapter 4).

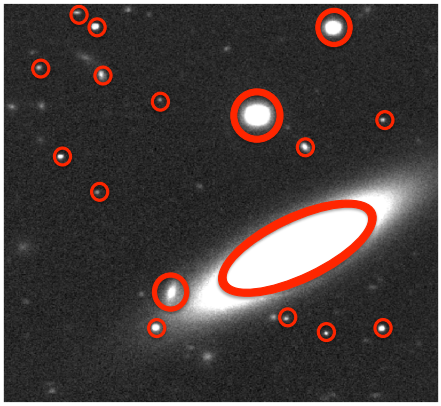
Figure 1.2 A motivating example from the astronomy domain that represents (a) a single LSST image, (b) the equivalent stacked image, (c) the same stacked image but after applying the sigma-clipping noise reduction algorithm, and (d) the result of feature extraction on that image.



(a) Single Image

(b) Co-added image

(c) Co-added + sigma-clipped image



(d) Image annotated with extracted features

Chapter 2

ARRAYSTORE: STORAGE MANAGER FOR COMPLEX, PARALLEL ARRAY PROCESSING

One major challenge with building an array data processing system is simply how to store arrays on disk. We explore this question in ArrayStore [97, 99]. ArrayStore is a new read-only storage manager for parallel array processing. ArrayStore supports a parallel and complex workload comprising not only range-queries, but also binary operations such as joins and user-defined functions.

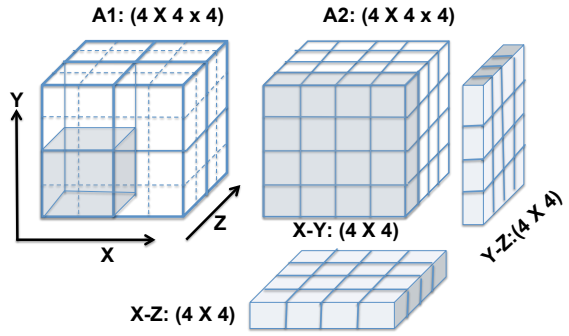
2.1 Requirements, Challenges, and Contributions

In this chapter, we address the following key question: *what is the appropriate storage management strategy for a parallel array processing system?* Unlike most other array-processing systems being built today [3, 19, 28, 110], we are not interested in building an array engine on top of a relational DBMS, but rather building a specialized storage manager from scratch. In this chapter, we consider read-only arrays and do not address the problem of updating arrays.

There is a long line of work on storing and indexing multidimensional data (see Section 6.2). A standard approach to storing an array is to partition it into sub-arrays called chunks [88] as illustrated in Figure 2.1. Each chunk is typically the size of a storage block. Chunking an array helps alleviate "dimension dependency" [94], where the number of blocks read from disk depends on the dimensions involved in a range-selection query rather than just the range size.

Requirements The design of a parallel array storage manager must thus answer the following questions (1) what is the most efficient array chunking strategy for a given workload, (2) how should the storage manager partition chunks across machines in a shared-nothing cluster to support parallel processing, and (3) how to efficiently support array operations that need to access data in adjacent chunks possibly located on other machines during parallel processing? Prior work examined some of these questions but only in the context of array scans and range-selection, nearest-neighbors,

Figure 2.1 (1) The 4x4x4 array A1 is divided into eight 2x2x2 chunks. Each chunk is a unit of I/O (a disk block or larger). Each X-Y, X-Z, or Y-Z slice needs to load 4 I/O units. (2) Array A2 is laid out linearly through nested traversal of its axes without chunking. X-Y needs to load only one I/O unit, while X-Z and Y-Z need to load the entire array.

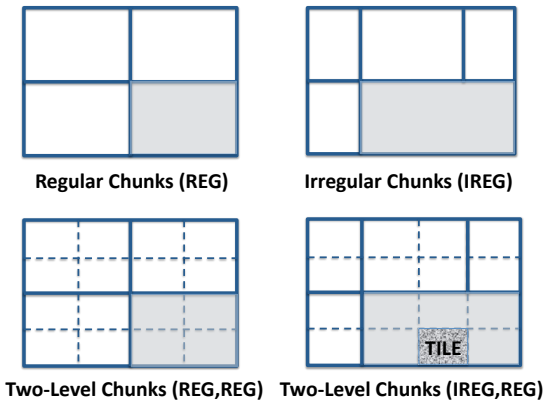


and other "lookup-style" operations [8, 14, 32, 55, 61, 73, 74, 86, 88, 94]. In contrast, our goal is to support a more varied workload as required by the science community [102]. In particular, we aim at supporting a workload comprising the following types of operations: (1) array slicing and dicing (*i.e.*, operations that extract a subset of an array [14, 15, 87]), (2) array scans (*e.g.*, filters, regrids [102], and other operations that process an entire array), (3) binary array operations (*e.g.*, joins, cross-match [66]), and (4) operations that need to access data from adjacent partitions during parallel processing (*e.g.*, canopy clustering [56]). We want to support both single-site and parallel versions of these operations.

Challenges The above types of operations impose *very different, even contradictory, requirements* on the storage manager. Indeed, array dicing can benefit from small, finely tuned chunks [32]. In contrast, user-defined functions may incur overhead when chunks are too small and processed in parallel [49] and they may need to efficiently access data in adjacent chunks. Different yet, joins need to simultaneously access corresponding pieces of two arrays, and they need a chunking method that facilitates this task. When processed in parallel, all these operations may also suffer from skew, where some groups of chunks take much longer to process than others [24, 48, 49], slowing down the entire operation. Binary operations also require that matching chunks from different arrays be co-located possibly causing data shuffling and thus imposing I/O overhead.

These requirements are especially hard to satisfy for sparse arrays (*i.e.*, an array is said to be

Figure 2.2 Four different chunking strategies applied to the same rectangular array. Solid lines show chunk boundaries of the logical array (sample chunks shaded). Inner-level tiles are represented by dashed lines (one tile is textured).

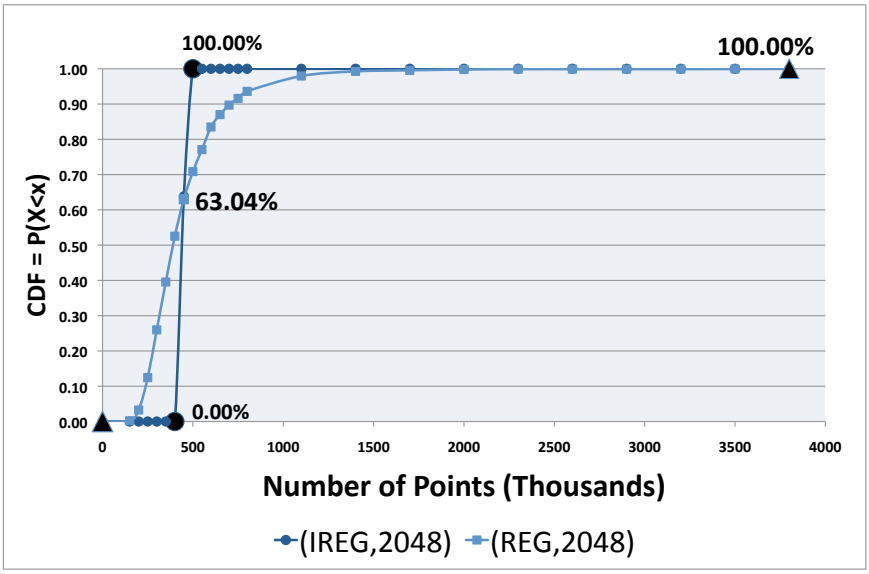


sparse when most of its cells do not contain any data) because data in a sparse array is unevenly distributed, which can worsen skew (*e.g.*, in one of our datasets, when splitting an array into 2048 chunks, we found a 25X difference between the chunk with the most and least amount of data). Common representations of sparse arrays in the form of an unordered list of coordinates and values also slow down access to subsets of an array chunk, because all data points must be scanned. We consider the problem of array storage in the context of SciDB whose goal is to provide a single storage engine with support for both dense and sparse arrays. The chunking problem is relatively simpler for dense arrays but is much harder for sparse arrays. In this chapter, we thus focus on sparse arrays. We assume there are no value indexes on these arrays.

Contributions We present the design, implementation, and evaluation of ArrayStore, a storage manager for parallel array processing. ArrayStore is designed to support *complex and varied* operations on arrays and parallel processing of these operations in a shared-nothing cluster. ArrayStore builds on techniques from the literature and introduces new techniques. The key contribution of the ArrayStore work is to answer the following two questions:

(1) *What combination of chunking and array partitioning strategies lead to highest performance under a varied parallel array processing workload?* (Sections 2.3.1 through 2.3.3). As in prior work, ArrayStore breaks arrays into multidimensional chunks, although we consider much larger chunks

Figure 2.3 Cumulative distribution function of number of points (*i.e.*, non-null cells) per chunk for regular (REG) and irregular (IREG) chunking in astronomy simulation snapshot S92 (Chapter 7). Both strategies use 2048 chunks. Large circles for IREG and large triangles for REG mark the 0% and 100% points in each distribution.



than prior work (hundreds of KBs to hundreds of MBs rather than a single disk block). We study four different array chunking techniques as summarized in Figure 2.2: regular chunks (REG), irregular chunks (IREG), and two-level chunks (IREG-REG and REG-REG).[1] In the case of regular chunks, the domain of each array index is divided into uniform partitions. For irregular chunks, we create chunk boundaries such that each chunk contains the *same amount of data* (in bytes), thus reducing possible skew when processing data chunks in parallel. Figure 2.3 illustrates the per-chunk data distribution differences when applying either the REG or IREG strategies. Finally, the basic idea behind the two-level approaches is to split an array into regular or irregular chunks, and then further divide each chunk into smaller regular fragments that we call *tiles*.

(2) *How to enable an operator to efficiently access data in neighboring array chunks during parallel processing?* (Section 2.3.4) We develop two new techniques to enable an operator to efficiently access a variable-amount of data in neighboring chunks during parallel processing. The

[1] In the ArrayStore work, we do not study indexing data within chunks, which is a complementary technique and could further speed-up some operations, nor data compression on disk.

first technique leverages directly our two-level REG-REG storage layout to enable an operator to efficiently read and process as much overlap data as needed. The second technique stores separate materialized views of increasingly distant overlapping data for each chunk.

We wrap the above techniques with a simple, yet flexible access method that we present in Section 2.4.

We implement ArrayStore and a set of representative operators in a standalone C++ system and evaluate the system on two real datasets from the science domain. The first one is a 74 GB dataset comprising two snapshots from an astronomy simulation [51] (3D data). The second dataset is the output of a flow cytometer from the oceanography domain [2] (6D data).

For the first question, *What combination of chunking and array partitioning strategies leads to the best overall performance under a varied workload?*, we show that a two-level REG-REG strategy leads to the best overall performance under a varied workload and requires the least tuning. Indeed, it provides high-performance for single-site processing of all operations in our workload and can be organized to avoid both skew and data shuffling during parallel processing. None of the other techniques simultaneously achieves all these goals. For the second question, *How to enable an operator to efficiently access data in neighboring array chunks during parallel processing?*, we show that ArrayStore's techniques outperform by a factor of 2X more naïve techniques where either overlap data is not explicitly supported or a pre-defined amount of overlap data is stored within or even separately from each chunk.

2.2 Problem Statement

We start with a more precise problem statement. We define an array similarly to Furtado and Baumann [32]: Given a discrete coordinate set $S = S_1 \times \ldots \times S_d$, where each $S_i, i \in [1, d]$ is a finite totally ordered discrete set, an array is defined by a d-dimensional domain $D = [I_1, \ldots, I_d]$, where each I_i is a subinterval of the corresponding S_i. Each combination of dimension values in D defines a *cell*. All cells in a given array have the same type T, which is a tuple as in a relational DBMS.

ArrayStore must efficiently support the types of array operations outlined in Section 2.1, which we formalize by presenting one or more representative operators for each type of operation.

Array Scan (*e.g.*, filter) Many operators process all chunks of an array such that each chunk can be processed independently of other chunks. Filter, $A' =$ FILTER(A,P), is representative of this type of operators (assuming no value-based indexes). Here, A is an input array and P is a predicate over cell values. The output array A' has the same dimensions as A such that if v is a vector of dimension values, A' contains $A(v)$ if $P(A(v))$ returns true, otherwise it contains $null$. A parallel filter, $A' =$ P_FILTER(A,P), can be computed by partitioning A into N sets of non-overlapping chunks, with a partitioning strategy R. Second, FILTER(n_i,P) is applied independently to each partition $n_i \in N$. A' is the union of the results.

Array Dicing (*e.g.*, subsample) We also want to support unary range-selection (or dicing) operators. *Subsample* [102], $A' =$ SUBSAMPLE(A,P), is representative of this type of operators. Here A is an array and P is a predicate over A's dimensions. SUBSAMPLE returns an array A' that has the same number of dimensions as A, but a smaller number of dimension values. In the ArrayStore work, we study subsample operators, where P takes the form of a d-dimensional subinterval $d = [i_1, \ldots, i_d]$, and selects all cells that fall within this subinterval. Similar to filter, subsample can process an array's chunks independently of one another and is thus trivial to parallelize.

Binary Array Operation (*e.g.*, join) In addition to unary operators, we need to support binary operators such as *JOIN*. As representative operator, we consider a simple version of a structural join [87], $B =$ JOIN(A,A'), where A, A', and B are defined over the same d-dimensional domain D and each cell in B is the concatenation of cells in A and A'. As a concrete example, such a join operator can correlate an array of temperature values with an array of pressure values, outputting tuples that comprise both values for each combination of dimension values. In practice, joins can get more complex. For example, a cross-match [66] compares cell values that are near each other in two input arrays rather than being at the exact same location. However, the key requirement of bringing together and processing corresponding array chunks remains. It is the key type of operation that we want to support. To execute a join in parallel, the strategy that we adopt is to re-partition array A' such that all cells corresponding to cells in A get physically co-located. Each pair of array partitions can then be processed independently and the results can be unioned.

Overlap Operations (*e.g.*, clustering and volume-density) Many array operations cannot be computed by simply partitioning an array, processing its partitions independently, and unioning the result. Instead, processing each array fragment requires access to data in adjacent fragments. We consider two types of such *overlap-based* operators: (1) operators that need to see a fixed amount of adjacent data and (2) operators that need to see a bounded, though not fixed, amount of adjacent data. We use canopy clustering as representative of the former type of operators and a volume-density application as representative of the latter. We describe them further in Sections 2.3.4 and 2.4.

Non-requirements in ArrayStore, we do not include in our workload iterative operations nor operations that examine a large number of input cells stretching across the array to compute the value of an output cell: *e.g.*, data clustering operations where a cluster can span a large fraction of the array. We discuss the former in Chapter 4 and the latter in [97].

2.3 ArrayStore Storage Manager

In this section, we present the design of ArrayStore.

2.3.1 Basic Array Chunking

As in prior work on storage management for multidimensional data (see Section 6.2), ArrayStore takes the approach of breaking an array into fragments called *chunks* and storing these chunks on disk. We now present two types of chunking schemes studied in the literature and the two-level strategy that we develop in ArrayStore.

Regular Chunks (REG) The first approach of breaking an array into chunks is to use what are called *regular chunks* [25, 32], where all chunks have the same size in terms of the coordinate space. For example, consider a $3D$ astronomy simulation snapshot with dimensions (X, Y, Z) such that X=[-0.5:0.5],Y=[-0.5:0.5], and Z=[-0.5:0.5]. We can break the array into 256 regular chunks, by splitting each X, Y, and Z dimension into 8, 8, and 4 respectively. Each chunk in the array will then have size $0.125 * 0.125 * 0.25$. Regular chunks are commonly used for storing arrays on disk [14, 29, 74, 88]. Figure 2.2 illustrates this approach.

Irregular Chunks (IREG) Several schemes have also been proposed where an array is fragmented in a less regular fashion [15, 32]. In this chapter, we call all such strategies *irregular* chunking schemes and illustrate them in Figure 2.2. Irregular chunking can speed-up range-selection queries when the chunk size and shape is tuned to the workload [32]. While our goal is not to tune storage for such specific queries, we consider irregular chunking, because it may help reduce skew in parallel array processing. The key idea is to chunk the array such that each chunk covers a different volume of the logical coordinate space but holds the same amount of data [48] as shown in Figure 2.3.[2] One approach that has been proposed for creating such chunks [48] is to use a kd-tree [7], which splits a multidimensional space into increasingly small partitions considering the data distribution to ensure load balance between partitions. If chunks are irregular, they must be indexed to support efficient access to subsets of an array. In our ArrayStore implementation, we index chunks using an R-tree. Other indexes are possible, but we do not find that the index lookup time is a bottleneck in our experiments.

Two-level Chunks (REG-REG or IREG-REG) For either of the above strategies a question that arises is that of appropriate chunk size. Large chunks help amortize seek times when reading data from disk. They also help amortize any potential fixed-costs associated with processing a data chunk by an operator. However, for arrays containing sparse data, large chunks increase the amount of processing required if an operator only needs a subset of a chunk (*e.g.*, subsample or an operator accessing data from adjacent chunks) because the lack of internal chunk structure forces the operator to examine all data points within the chunk.

To address these contradictory requirements, an alternate approach is to create *two-level chunks*. The basic idea is to split an array into small, regular chunks but then combine them together to form larger chunks that are either regular (*REG-REG*) or irregular (*IREG-REG*) as illustrated in Figure 2.2. With this approach, the larger chunks are the unit of disk I/O, while the smaller *tiles* can be the unit of array processing. Regular chunks and tiles efficiently support binary operators on a single-node and across nodes because they facilitates the co-location and co-processing of matching cells across two arrays. In contrast, irregular chunks can help smooth-out data skew during parallel processing.

---

[2]For dense arrays, this approach is identical to regular chunks.

Two-level chunking has been studied before [82, 94], but only as a container to place multiple chunks on a single disk block. This approach is a form of IREG-REG, since regular tiles are grouped into irregular chunks. We push the idea further by not only using bigger chunks to amortize seek time overhead (unit of I/O) and operator overhead, but also by enabling operators to process different granularity of chunks as needed (see Section 2.4), by leveraging the two-level structure to efficiently support overlap processing (see Section 2.3.4), and by exploring the regular-regular (*REG-REG*) approach as an alternative to IREG-REG. Through experiments (see Section 2.5), we show that REG-REG is not only the simpler of the two strategies but also leads to highest performance under a varied workload.

2.3.2 Organizing Chunks on Disk

Each array in ArrayStore is represented with one data file and one metadata file. The data file contains the actual array values. The metadata file contains array meta information such as number of dimensions, total number of chunks, and in the case of regular chunking the number of chunks along each dimension. The metadata file also contains overlap information (see Section 2.3.4). For irregular chunking, a chunk index is stored in a separate file. In the ArrayStore work, we do not study how chunk layout on disk affects performance as it mostly matters for dicing queries [94]. For sparse arrays, only non-null cells are stored inside chunks and their order is arbitrary. The only way to access a particular cell in a chunk is thus to sequentially scan the cells inside the chunk. This approach avoids the overhead of creating an index within each chunk and we show that the two-level REG-REG storage management enables high performance even without such index.

2.3.3 Organizing Chunks across Disks

To support parallel array processing, ArrayStore can spread array chunks across multiple independent processing units or *nodes* (*i.e.*, physical machines, processes on the same machine, or other). For this, ArrayStore partitions an array into N *segments*, each holding a subset of the array chunks, not necessarily contiguous, and distributes each segment to a node.

We study the performance of several array partitioning strategies including (1) random (assign each chunk to a randomly selected segment), (2) round-robin (iterate over chunks in some order and

assign them to each segment in turn), (3) range (split the array into N disjoint ranges of chunks and assign all chunks within a range to a segment), or (5) block-cyclic (split the array into M regular *blocks* of N chunks each. Iterate over the chunks of a block in some pre-defined order and assign them to each of the N segments in turn). Block-cyclic is thus similar to round-robin but it helps spread dense array regions across more nodes (along all dimensions). For example, consider a 2D array $A_{4\times4}$ which consists of 16 chunks labeled 1 to 16 in row-major order (first row holds chunks $\{1, 2, 3, 4\}$, second row holds $\{5, 6, 7, 8\}$, etc). Block-cyclic partitions chunks in array $A_{4\times4}$ on 4 nodes such that chunks labeled $\{$1,3,9,11$\}$ are assigned to the first node, while in round-robin, that node contains $\{$1,5,9,13$\}$, all the chunks in the first array column. We do not study hash-partitioning, because it is equivalent to either random or a form of block-cyclic partitioning.

2.3.4 Overlap Data Support

When processing an array in parallel, ideally, one would like to process each array segment (or even chunk or tile) independently of the others and simply union the results. Many scientific array operations, however, cannot be parallelized using this simple strategy. Indeed, operations such as regression or clustering require that an operator considers data from a range of neighboring cells in order to produce each output cell. To illustrate the problem and our approach to addressing it, we use canopy clustering [56] as running example. In this section, we assume that the unit of parallel processing is an array chunk. We come back to tile-based and segment-based processing in Section 2.4

Canopy clustering is a fast clustering method typically used to create preliminary clusters that are then further processed by more sophisticated algorithms [56]. Canopy clustering can serve to cluster data points in a sparse array, such as the 3D astronomy or 6D flow-cytometer datasets. In fact, data clustering is commonly used in both domains [49].

The canopy clustering algorithm takes as input a distance metric and two distance thresholds $T1 > T2$. To cluster data points stored in a sparse array, the algorithm proceeds iteratively: it first removes a point at random from the array and uses it to form a new cluster. The algorithm then iterates over the remaining points. If the distance between a remaining point and the original point is less than $T1$, the algorithm adds the point to the new cluster. If the distance is also less than $T2$,

the algorithm eliminates the point from the set. Once the iteration completes, the algorithm selects one of the remaining points (*i.e.*, those not eliminated by the $T2$ threshold rule) as a new cluster and repeats the above procedure. The algorithm continues until the original set of points is empty. The algorithm outputs a set of canopies each of them with one or more data points.

Problems with Ignoring Overlap Needs To run canopy clustering in parallel, one approach is to partition the array into chunks and process chunks independently of one another. The problem is that points at chunk boundary may need to be added to clusters in adjacent chunks and two points (even from different chunks) within $T2$ of each other should not both yield a new canopy. A common approach to these problems is to perform a post-processing step [48, 49, 56]. For canopy clustering, this second step clusters canopy centers found in individual partitions and assigns points to these final canopies [56]. Such a post-processing phase, however, can add significant overhead as we show in Section 2.5.

Single-Layer Overlap To avoid a post-processing phase, some have suggested to extract, for each array chunk, an overlap area ϵ from neighboring chunks, store the overlap together with the original chunk [87, 91], and provide both to the operator during processing. In the case of canopy clustering, an overlap of size $T1$ can help reconcile canopies at partition boundary. The key insight is that the overlap area needed for many algorithms is typically small compared to the chunk size. A key challenge with this approach, however, is that even small overlap can impose significant overhead for multidimensional arrays. For example, if chunks become 10% larger along each dimension (only 5% on each side) to cover the overlapping area, the total I/O and CPU overhead is 33% for a 3D chunk and over 75% for a 6D one!

A simple optimization is to store overlap data separately from the core array and provide it to operators on demand. This optimization helps operators that do not use overlap data. However, operators that need the overlap still face the problem of having access to a single overlap region, which must be large-enough to satisfy all queries.

Multi-Layer Overlap Leveraging Two-level Storage In ArrayStore, we propose a more efficient approach to supporting overlap data processing. We present our core approach here and an important

Algorithm 1 Multi-Layer Overlap over Two-level Storage

1: **Multi-Layer Overlap over Two-level Storage**
2: **Input:** chunk $core\_chunk$ and predicate $overlap\_region$.
3: **Output:** chunk $result\_chunk$ containing all overlap tiles.
4: $ochunkSet \leftarrow$ all chunks overlapping $overlap\_region$.
5: $tileSet \leftarrow \emptyset$
6: **for all** Chunk $ochunk_i$ in $ochunkSet - core\_chunk$ **do**
7: Load $ochunk_i$ into memory.
8: $tis \leftarrow$ all tiles in $ochunk_i$ overlapping $overlap\_region$.
9: $tileSet \leftarrow tileset \cup tis$
10: **end for**
11: Combine $tilesSet$ into one chunk $result\_chunk$. **return** $result\_chunk$.

optimization below.

ArrayStore enables an operator to request an *arbitrary amount of overlap data* for a chunk. No maximum overlap area needs to be configured ahead of time. Each operator can use a different amount of overlap data. In fact, an operator can use a different amount of overlap data *for each chunk*. We show in Section 2.5, that this approach yields significant performance gains over all strategies described above.

To support this strategy, ArrayStore leverages its two-level array layout. When an operator requests overlap data, it specifies a desired range around its current chunk. In the case of canopy clustering, given a chunk that covers the interval $[a_i, b_i]$ along each dimension i, the operator can ask for overlap in the region $[a_i - T_1, b_i + T_1]$. To serve the request, ArrayStore looks-up all chunks overlapping the desired area (omitting the chunk that the operator already has). It loads them into memory, but cuts out only those tiles that fall within the desired range. It combines all tiles into one chunk and passes it to the operator. Algorithm 1 shows the corresponding pseudo-code.

As an optimization, an operator can specify the desired overlap as a a hypercube with a hole in the middle. For example, in Figure 2.4, canopy clustering first requests all data that falls within range L_1 and later requests L_2. For other chunks, it may also need L_3.

When partitioning array data into segments (for parallel processing across different nodes), ArrayStore replicates chunks necessary to provide a pre-defined amount of overlap data. Requests for additional overlap data can be accommodated but require data transfers between nodes.

Algorithm 2 Multi-Layer Overlap using Overlap Views

1: **Multi-Layer Overlap using Overlap Views**
2: **Input:** chunk $core\_chunk$ and predicate $overlap\_region$.
3: **Output:** chunk $result\_chunk$ containing requested overlap data.
4: Identify materialized view M to use.
5: $L \leftarrow layers\ l_i \in M$ that overlap $overlap\_region$.
6: Initialize an empty $result\_chunk$
7: **for all** Layer $l_i \in L$ **do**
8: Load layer l_i into memory.
9: Add l_i to $result\_chunk$.
10: **end forreturn** $result\_chunk$.

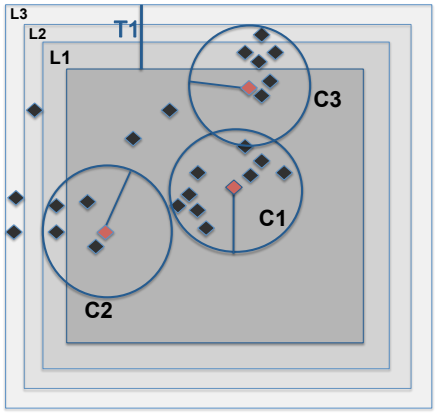
Multi-Layer Overlap through Materialized Overlap Views While superior to single-layer overlap, the above approach suffers from two inefficiencies: First, when an operator requests overlap data within a neighboring chunk, the entire chunk must be read from disk. Second, overlap layers are defined at the granularity of tiles.

To address both inefficiencies, ArrayStore also supports *materialized overlap views*. A materialized overlap view is defined like a set of onion-skin layers around chunks: *e.g.*, layers L_1 through L_3 in Figure 2.4. A view definition takes the form $(n, w_1, \ldots, w_d)$, where n is the number of layers requested and each w_i is the thickness of a layer along dimension i. Multiple views can exist for a single array.

To serve a request for overlap data, ArrayStore first chooses the materialized view that covers the entire range of requested data and will result in the least amount of extra data read and processed. From that view, ArrayStore loads only those layers that cover the requested region, combines them into a chunk and passes the chunk to the operator. Algorithm 2 shows the pseudo-code.

Materialized overlap views impose storage overhead. As above, a 10% overlap along each dimension adds 33% total storage for a 3D array. With 20% overlap, the overhead grows to 75%. In a 6D array, the same overlaps add 75% and 3X, respectively. Because storage is cheap, however, we argue that such overheads are reasonable. We further discuss materialized overlap views selection in Section 2.5.3.

Figure 2.4 Example of multi-layer overlap used during canopy clustering. $C2$ necessitates that the operator loads a small amount of overlap data denoted with $L1$. $C3$, however, requires an additional overlap layer. So $L2$ is also loaded.



Array Iterator Methods
open(Range r, PackRatio p)
boolean hasNext()
Chunk getNext() throws NoSuchElementException
Chunk getOverlap(Range r) throws NoSuchElementException
close()

Table 2.1: Access Method API

2.4 ArrayStore Access Method

ArrayStore provides a single access method that supports the various operator types presented in Section 2.2, including overlap data access. The basic access method enables an operator to iterate over array chunks, but how that iteration is performed is highly configurable.

Array Iterator API The array iterator provides the five methods shown in Table 2.4. This API is exposed to operator developers not end-users. Our API assumes a chunk-based model for programming operators, which helps the system deliver high-performance.

Method `open` opens an iterator over an array (or array segment). This method takes two *optional* parameters as input: a range predicate (`Range r`) over array dimensions, which limits the

iteration to those array chunks that overlap with r; the second parameter is, what we call the *packing ratio* (`PackRatio p`). It enables an operator to set the granularity of the iteration to either "tiles" (default), "chunks", or "combined". Tiles are perfect for operators that benefit from finely-structured data such as subsample. For this packing ratio, the iterator returns individual tiles as chunks on each call to `getNext()`. In contrast, the "chunks" packing ratio works best for operators that incur overhead with each unit of processing, such as operators that work with overlap data. Finally, the "combined" packing ratio combines into a single chunk all tiles that overlap with r. If r is "null", "combined" returns all chunks of the underlying array (or array segment) as one chunk. If an array segment comprises chunks that are not connected or will not all fit in memory, "combined" iterates over chunks without combining them. In the next section, we show how a binary operator such as join greatly benefits from the option to "combine" chunks.

Methods `hasNext()`, `getNext()`, and `close()` have the standard semantics.

Method `getOverlap(Range r)` returns as a single chunk all cells that overlap with the given region (defined by predicate r) and surround the current element (tile, chunk, or combined). Because overlap data is only retrieved at the granularity of tiles or overlap layers specified in the materialized views, extra cells may be returned. Overlap data can be requested for a tile, a chunk, or a group of tiles/chunks. However, ArrayStore supports materialized overlap views only at the granularity of chunks or groups of chunks. The intuition behind this design decision is that, in most cases, operators that need to process overlap data would incur too much overhead doing so for individual tiles and ArrayStore thus optimizes for the case where overlap is requested for entire chunks or larger.

Example Operator Algorithms in ArrayStore We illustrate ArrayStore's access method by showing how several representative operators (from Section 2.2) can be implemented.

Filter processes array cells independently of one another. Given an array segment, a filter operator can thus call `open()` without any arguments followed by `getNext()` until all tiles have been processed. Each input tile serves to produce one output tile.

Subsample. Given an array segment, a subsample operator can call `open(r)`, where r is the requested range over the array, followed by a series of `getNext()` calls. Each call to `getNext()`

Algorithm 3 Join algorithm.

```
1: JoinArray
2: input: array_1 and array_2, iterators over arrays to join
3: output: result_array, set of result array chunks
4: array_1.open(null, "chunk")
5: while array_1.hasNext() do
6:     Chunk chunk_1 = array_1.getNext()
7:     Range r = rectangular boundary of chunk_1
8:     array_2.open(r,"combined")
9:     if array_2.hasNext() then
10:        Chunk chunk_2 = array_2.getNext()
11:        result_chunk = JOIN(chunk_1, chunk_2)
12:        result_array = result_array ∪ result_chunk
13:    end if
14: end whilereturn result_array
```

will return a tile. If the tile is completely inside r, it can be copied to the output unchanged, which is very efficient. If the tile partially overlaps the range, it must be processed to remove all cells outside r.

Join. As described in Section 2.2, we consider a structural join [87] that works as follows: For each pair of cells at matching positions in the input arrays, compute the output cell tuple based on the two input cell tuples. This join can be implemented as a type of nested-loop join (Algorithm 3). The join iterates over chunks of the outer array, $array_1$ (it could also process an entire outer array segment at once), preferably the one with the larger chunks. For each chunk, it looks-up the corresponding tiles in the inner array, $array_2$, retrieves them all as a single chunk (*i.e.*, option "combined"), and joins the two chunks. In our experiments, we found that combining inner tiles could reduce cache misses by half, leading to a similar decrease in runtime.

All three operators above can directly execute in parallel using the same algorithms. The only requirement is that chunks of two arrays that need to be joined be physically co-located. As a result different array partitioning strategies yield different performance results for join (see Section 2.5).

Canopy Clustering. We described the canopy clustering algorithm in Section 2.3.4. Here we present its implementation on top of ArrayStore. The pseudo-code of the algorithm is omitted due to the space constraints. The algorithm iterates over array chunks. Each chunk is processed independently of the others and the results are unioned. For each chunk, when needed, the algorithm incrementally grows the region under consideration (through successive calls to `getOverlap()`)

to ensure that, every time a point x_i starts a new cluster, all points within $T1$ of x_i are added to the cluster just as in the centralized version of the algorithm. The maximum overlap area used for any chunk is thus $T1$. Points within $T2 < T1$ of each other should not both yield new canopies. In our implementation, to avoid double-reporting canopies that cross partition boundaries, only canopies whose centroids are inside the original chunk are returned.

Volume-Density algorithm. The Volume-Density algorithm is most commonly used to find what is called a *virial radius* in astronomy [50]. It further demonstrates the benefit of multi-layer overlap. Given a set of points in a multidimensional space (*i.e.*, a sparse array) and a set of cluster centroids, the volume-density algorithm finds the size of the sphere around each centroid such that the density of the sphere is just below some threshold T. In the astronomy simulation domain, data points are particles and the sphere density is given by: $d = \frac{\Sigma mass(p_i)}{volume(r)}$, where each p_i is a point inside the sphere of radius r. This algorithm can benefit from overlap: Given a centroid c inside a chunk, the algorithm can grow the sphere around c incrementally, requesting increasingly further overlap data if the sphere exceeds chunk boundary.

2.5 Evaluation

In this section, we evaluate ArrayStore's performance on two real datasets and on eight dual quad-core 2.66GHz Intel/AMD OpteronPentium-based machines with 16GB of RAM running RHEL5.

The first dataset is **Astro** dataset that is described in Chapter 7. The Astro dataset comprises several snapshots from a large-scale astronomy simulation [51] for a total of 74 GB of data. We used two snapshots, $S43$ and $S92$ from this dataset. Each snapshot represents the universe as a set of particles in a 3D space. Since the universe is becoming increasingly structured over time, data in snapshot $S92$ is more skewed than in $S43$. In Figure 2.3, the largest regular chunk has 25X more data points than the smallest one. The ratio is only 7 in $S43$ for the same number of chunks.

The second dataset is **OceanFlow** dataset that is described in Chapter 7. In this dataset, the data takes the form of points in a 6-dimensional space, where each point represents a particle or organism in the water and the dimensions are the measured properties. Each array in this dataset is approximately 7 GB in size. Join queries thus run on 14 GB of 6D data.

Notation	Description
(REG,N)	One-level, regular chunks. Array split into N chunks total.
(IREG,N)	One-level, irregular chunks. Array split into N chunks total.
(REG-REG, N1-N2)	Two-level chunks. Array split into N1 regular chunks and N2 regular tiles.
(IREG-REG,N1-N2)	Two-level chunks. Array split into N1 irregular chunks and N2 regular tiles.

Table 2.2: Naming convention used in experiments.

Table 2.2 shows the naming convention for the experimental setups. ArrayStore's best-performing strategy is highlighted

2.5.1 Basic Performance of Two-Level Storage

First, we demonstrate the benefits of ArrayStore's two-level REG-REG storage manager compared with IREG-REG, REG, and IREG when running on a single node (single-threaded processing). We compare the performance of these different strategies for the subsample and join operators, which are the only operators in our representative set that are affected by the chunk shape. We show that REG-REG yields the highest performance and requires the least tuning. Figures 2.5 and 2.6 show the results. In both figures, the y-axis is the total query processing time.

Array dicing query Figure 2.5(a) shows the results of a range selection query, when the selected region is a 3D rectangular slice of $S92$ (we observe the same trend in $S43$). Each bar shows the average of 10 runs. The error bars show the minimum and maximum runtimes. In each run, we randomly select the region of interest. All the randomly selected, rectangular regions are 1% of the array volume. Selecting 0.1% and 10% region sizes yielded the same trends. We compare the results for REG, IREG, REG-REG, and IREG-REG.

For both single-level techniques (REG and IREG), larger chunks yield worse performance than smaller ones because more unnecessary data must be processed (chunks are *misaligned* compared with the selected region). When chunk sizes become too small (at 262144 chunks in this experiment), however, disk seek times start to visibly hurt performance. In this experiment, the best performance is achieved for 65536 chunks (approximately 0.56 MB per chunk).

The disk seek time effect is more pronounced for REG than IREG simply because we used a different chunk layout for REG than IREG (row-major order *v.s.* z-order [94]) and our range-selection

queries were worst-case for the REG layout. Otherwise, the two techniques perform similarly. Indeed, the key performance trade-off is disk I/O overhead for small chunks *v.s.* CPU overhead for large chunks. IREG only keeps the variance low between experiments since all chunks contain the same amount of data.

The overhead of disk seek times rapidly grows with the number of dimensions: for the 6D flow cytometer dataset (Figure 2.5(b)), disk I/O increases by a factor of 3X as we increase the number of chunks from 4096 to 2097152 while processing times decreases by a factor of 2X. Processing times do not improve for the smallest chunk size (2097152) because our range-selection queries pick up the same amount of data, just spread across a larger number of chunks.

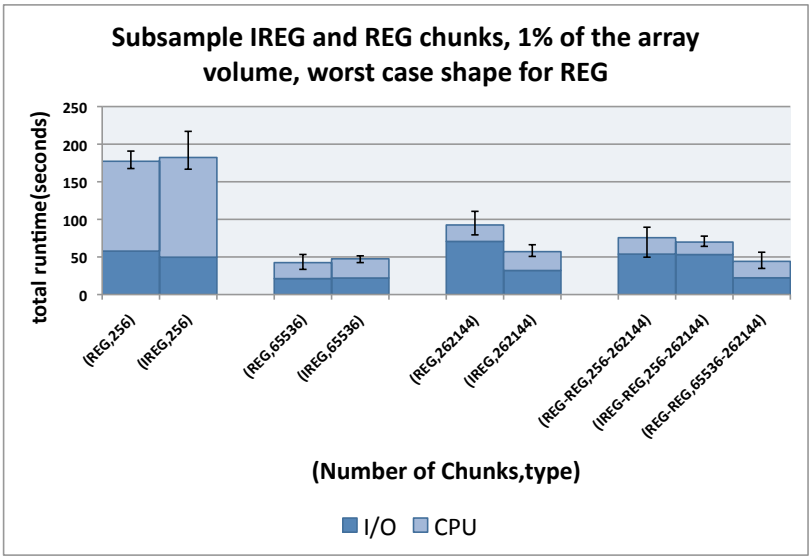
Most importantly, for these types of queries, the two-level storage management strategies are clear winners: they can achieve the low I/O times of small but not too small chunk sizes and the processing times of the smallest chunk sizes. The effect can be seen for both the 3D and 6D datasets. Additionally, the two-level storage strategies are significantly more resilient to suboptimal parameter choices, leading to consistently good performance. The two-level storage thus requires much less tuning to achieve high performance compared with a single-level storage strategy.

Join query Figure 2.6(a) shows the total query runtime results when joining two 3D arrays (two different snapshots or same snapshot as indicated). Figure 2.6(c) shows the results for a self-join on the 6D array.

We first consider the first three bars in Figure 2.6(a). The first bar shows the performance of joining two arrays, each using the IREG storage strategy. The second bar shows what happens when REG is used but the array chunks are misaligned: That is, each chunk in the finer-chunked array overlaps with multiple chunks in the coarser-chunked array. In both cases, the total time to complete the join is high such that it becomes worth to re-chunk one of the arrays to match the layout of the other as shown in the third bar. For each chunk in the outer array, the overhead of the chunk misalignment comes from scanning points in partly overlapping tiles in the inner array before doing the join only on subsets of these points.

The following two bars (A4 and A5) show the results of joining two arrays with different chunk sizes but with aligned regular chunks. That is, each chunk in the finer-chunked array overlaps with

Figure 2.5 Array dicing query on 3D and 6D datasets.



(a) Performance of array dicing query on 3D slices that are 1% of the array volume on S92. Two-level storage strategy yields the best overall performance and also the most consistent performance for different parameter choices.

Type	I/O time (Sec)	Proc. time (Sec)	Total
(REG,4096)	28	115	143
(REG,262144)	46	51	97
(REG,2097152)	90	66	156
(REG-REG,4096-2097152)	28	64	92

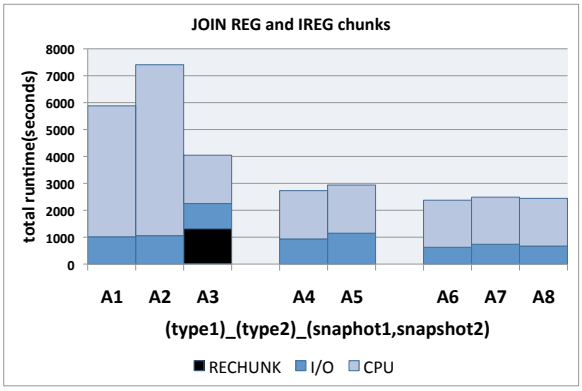
(b) Same experiment as above but on 6D dataset. The two-level strategy dominates the one-level approach again.

exactly one chunk in the coarser-chunked array. In that case, independent of how the arrays are chunked, performance is high and consistent. We tried other configurations, which all yielded similar results.

Interestingly, the overhead of chunk misalignment (always occurring with IREG and occurring in some REG configurations as discussed above) can rapidly grow with array dimensionality. The processing time of non-aligned 3D arrays is $3.5X$ that of aligned ones, while the factor is $6X$ for 6D arrays (Figure 2.6(c)).

Finally, the last three bars in Figure 2.6(a) show the results of joining two arrays with either one-level REG or two-level IREG-REG or REG-REG strategies. In all cases, we selected configurations where tiles were aligned. The alignment of inner tiles is the key factor to achieving high performance and thus all configurations result in similar runtimes.

Figure 2.6 Join query on 3D and 6D arrays.



(a) Join query performance on 3D array. Tile alignment is the key factor for the performance gain.

A1	(IREG,512)_(IREG,2048)_(92,43)
A2	(REG,512)_(REG,400)_(92,92)
A3	Rechunk(A2) + (REG,512)_(REG,2048)_(92,92)
A4	(REG,512)_(REG,2048)_(92,92)
A5	(REG,65536)_(REG,262144)_(92,92)
A6	(IREG-REG,256-262144)_(REG,2048)_(92,43)
A7	(REG-REG,256-262144)_(REG-REG,2048-262144)_(92,43)
A8	(REG,256)_(REG,2048)_(92,43)

(b) Notation.

Type	I/O time	Proc. time
(REG,REG)_NONALIGNED_6D	205	6227
(REG,REG)_ALIGNED_6D	221	988
(REG-REG,REG-REG)_ALIGNED_6D	222	993

(c) Join query performance on 6D array. Processing time of non-aligned configuration is 6X that of the aligned one.

Summary. The above experiments show that IREG array chunking does not outperform REG on array dicing queries and can significantly worsen performance in the case of joins. In contrast, a two-level chunking strategy, even with regular chunks at both levels can improve performance for some operators (dicing queries) without hurting others (selection queries and joins). The latter thus appears as the winning strategy for single-node array processing.

2.5.2 Skew-Tolerance of Regular Chunks

While regular chunking yields high performance for single-threaded array processing, an important reason for considering irregular chunks is skew. Indeed, in the latter case, all chunks contain the same amount of information and thus have a better chance of taking the same amount of time

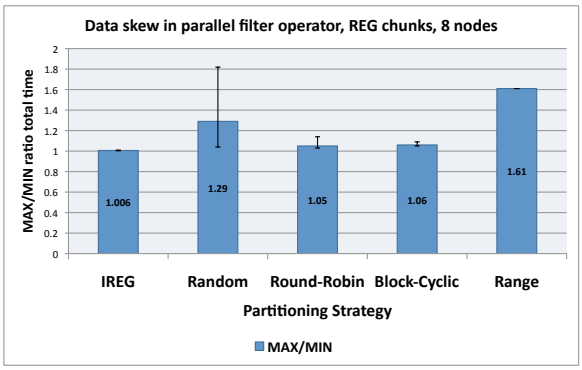
to process. In this section, we study skew during parallel query processing for different types of queries and different storage management strategies. We use a real distributed setup with 8 physical machines (1 node = 1 machine). To run parallel experiments, we first run the data shuffling phase and then run ArrayStore locally at each node. During shuffling, all nodes exchange data simultaneously using TCP. Note that in the study of data skew over multiple nodes, REG-REG and IREG-REG converge to REG and IREG storage strategies, respectively because we always partition data based on chunks rather than tiles.

Parallel Selection Figure 2.7 shows the total runtime of a parallel selection query on 8 nodes with random, range, block-cyclic, and round-robin partitioning strategies. All these scenarios use regular chunks. The experiment shows results for the S92 dataset (our most highly skewed dataset). The figure also shows results for IREG and random partitioning, one of the ideal configurations to avoid data skew. On the y-axis, each bar shows the ratio between the maximum and minimum runtime across all eight nodes in the cluster (*i.e.*, $MAX/MIN = \frac{max(r_i)}{min(r_j)}$ where $i, j \in [1, N]$ and r_i is equal to the total runtime of the selection query on node i). Error bars show results for different chunk sizes from 140 MB to 140 KB.

For REG, block-cyclic data partitioning exhibits almost no skew with results similar to those of IREG and random partitioning. Runtimes stay within 9% of each other for all chunk sizes. Runtimes in round-robin also stays within 14% for all chunk sizes. Performance is a bit worse than block-cyclic as the latter better spreads dense regions along *all* dimensions. For random data partitioning, skew can be eliminated with sufficiently small chunk sizes. The only strategy that heavily suffers from skew is range partitioning.

Parallel Dicing Similarly to selection queries in parallel DBMSs, parallel array dicing queries can incur significant skew when only some nodes hold the desired array fragments as shown in Figure 2.8. In this case, the problem comes from the way data is partitioned and is not alleviated by using an IREG chunking strategy. Instead, distributing chunks using the block-cyclic data partitioning strategy with small chunks can spread the load much more evenly across nodes. We measure a MAX/MIN ratio of just 1.11 with 4 nodes and 65536 chunks with a std deviation of 0.036.

Figure 2.7 Parallel selection on 8 nodes with different partitioning strategies on REG chunks. We vary chunk sizes from 140 MB to 140 KB. Error bars show the variation of MAX/MIN runtime ratios in that range of chunk sizes. Round-Robin and Block-Cyclic have the lowest skew and variance. Results for these strategies are similar to those of IREG.



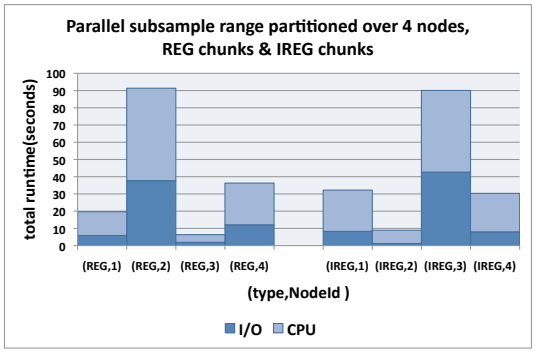
Parallel Join Array joins can be performed in parallel using a two-phase strategy. First, data from one of the arrays is shuffled such that chunks that need to be joined together become co-located. During shuffling, all nodes exchange data simultaneously using TCP. In our experiments, we shuffle the array with the smaller chunks. Second, each node can perform a local join operation between overlapping chunks.

Table 2.3 shows the percent data shuffled in an 8-node configuration. Shuffling can be completely avoided when arrays follow the same REG chunking scheme and chunks are partitioned deterministically. When arrays use different REG chunks, the same number of chunks are shuffled for all strategies. The shuffling time, however, is lowest for range and block-cyclic thanks to lower network contention. With range partitioning, each node only sends data to nodes with neighboring chunks. Block-cyclic spreads dense chunks better across nodes than round-robin and assigns the same number of chunks to each node unlike random. Range partitioning, however, exhibits skew in the ”local join phase” (Table 2.4), leaving block-cyclic as the winning approach.

Table 2.3 also shows the shuffling cost with IREG-REG chunks. Irregular chunks always suffer from at least some shuffling overhead. The best strategy, range partitioning, still shuffles 11% of data even when both arrays are ranged partition along the same dimension.

Summary. The above experiments show block-cyclic with REG chunks as the winning strategy:

Figure 2.8 Parallel subsample with REG or IREG chunks distributed using range partitioning across 4 nodes. Subsample runs only on a few nodes causing skew, independent of the chunking scheme chosen.



For parallel selection, block-cyclic has less than 9% skew for all regular chunk sizes. For parallel dicing, it also effectively distributes the workload across nodes. Finally, for parallel join block-cyclic can avoid data shuffling and offers the best performance for the local join phase. Irregular chunks can smooth out skew for some operations such as selections, but they hurt join performance both in the local join and data shuffling phases. We prefer two-level chunking strategy REG-REG to IREG-REG mainly due to the simplicity and leveraging block-cyclic partitioning strategy. Also IREG-REG suffers from shuffling phase overhead in parallel join.

2.5.3 Performance of Overlap Storage in ArrayStore

We present the performance of ArrayStore's overlap processing strategy. We compare four options: ArrayStore's multi-layered overlap implemented on top of the two-level storage manager, ArrayStore's materialized multi-layer overlap, single-layer overlap, and no-overlap.

In all experiments, we use (REG-REG,2048-262144) for the 3D arrays and (REG-REG,4096-2097152) for the 6D arrays. In ArrayStore, we assume that the user knows how much overlap his queries need (*e.g.*, canopy threshold T1) and he creates sufficiently large materialized overlap views to satisfy most queries within storage space constraints. The width of overlap layers is tunable, but we find its effect to be less significant. Hence, we expect that a single view with thin layers should suffice for most arrays. In our experiments, we materialize 20 thin layers of overlap data for each chunk, which cover a total of 0.5 and 0.2 of each dimension length in 3D and 6D, respectively. The

Partitioning Strategy	Type	Shuffling
Same chunking strategy, chunks are co-located, no shuffling.		
Block-Cyclic	(REG-2048,REG-2048)	(00.0%,0)
Round-Robin	(REG-2048,REG-2048)	(00.0%,0)
Range (same dim)	(REG-2048,REG-2048)	(00.0%,0)
Different chunking strategies for two arrays, shuffling percentage.		
Round-robin	(REG-2048,REG-65536)	(87.5%,1498)
Random	(REG-2048,REG-65536)	(87.6%,1416)
Block-Cyclic	(REG-2048,REG-65536)	(87.5%,1326)
Range (same dim)	(REG-2048,REG-65536)	(00.0%,0)
Range (different dim)	(REG-2048,REG-65536)	(87.5%,1313)
IREG-REG chunks, shuffling required.		
Random	(TYPE1,TYPE2)	(62.0%,895)
Round-Robin	(TYPE1,TYPE2)	(73.0%,836)
Range (same dim)	(TYPE1,TYPE2)	(11.0%,210)
Block-Cyclic	(TYPE1,TYPE2)	N/A

Table 2.3: Parallel join (shuffling phase) for different types of chunk partitioning strategies across 8 nodes. TYPE1=(IREG-REG,2048-262144) in S43 and TYPE2= (IREG-REG,2048-262144) in S92. Each value in "Shuffling" column is a pair of (Cost,Time(sec.)).

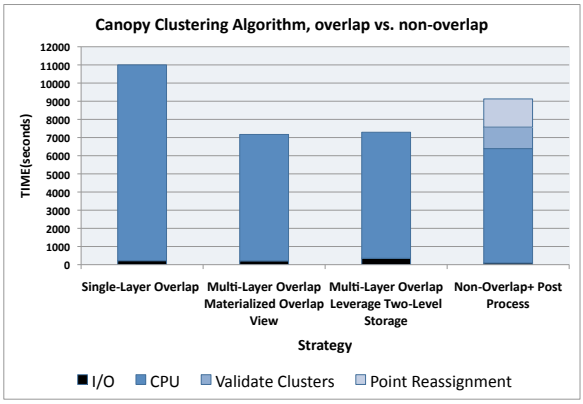
Technique:	Random	Round-robin	Range	Block-cyclic
Avg	1.24	1.08	1.18	1.06
Max - Min	1.56-1.16	1.08-1.08	1.18-1.18	1.06-1.06

Table 2.4: "Local join phase" with regular chunks partitioned across 8 nodes. The values in the table are the ratios of total runtime between the slowest and fastest parallel nodes. The table shows the Avg, Min, and Max ratios across 10 experiments. Block-cyclic has the least skew, (6%) compared to other techniques.

choice of 20 layers is arbitrary. The single-layer overlap is the concatenation of these 20 layers. Experiments are on a single node.

Figure 2.9 presents the performance results for the canopy clustering algorithm. $T1$ is set to 0.0125 (20% of the dimension length). We set $T2 = (0.75)T1$. Note that in the no-overlap case, a post processing phase is required to combine locally found canopies into global ones [56]. As the figure shows, both multi-layer overlap strategies outperform no-overlap and single-layer overlap by 25% and 35% respectively. The performance of single-layer overlap varies significantly depending on the overlap size chosen. In this experiment, the single-layer overlap is large to emphasize the drawback of inaccurate settings for this approach. In contrast, with multi-layered overlap, we can choose fine grained overlap layers (using views or small-size tiles), and get high-performance without tuning the total overlap size. Additionally, different applications can use different overlap

Figure 2.9 Canopy Clustering algorithm with or without overlap on the 3D dataset. Single-layer overlap does not perform well because of a large maximum overlap-size chosen. Both multi-layer overlap techniques outperforms no-overlap by 25%.

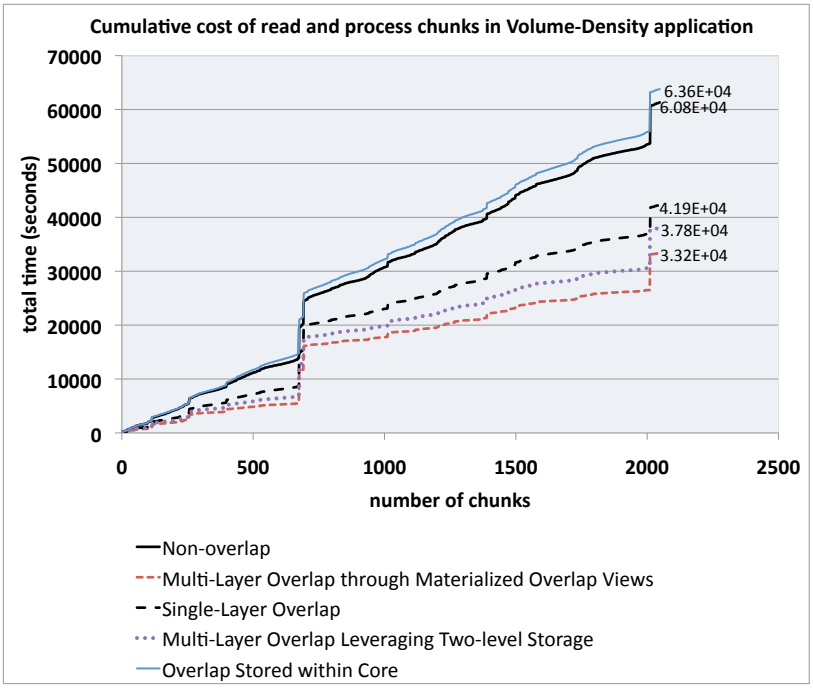


sizes without hurting each other's performance, which is not the case for the single-layer overlap approach. We ran the canopy application on the 6D dataset with the same $T1$ and $T2$ settings as in the 3D experiment and observed 16% improvement in total runtime for multi-layer overlap using materialized view compared with no-overlap. When leveraging the two-level storage, the improvement was 11%, mainly because of I/O overhead due to loading entire chunks.

Figure 2.10 shows the results from the volume-density application described in Section 2.4 on the 3D dataset. As shown in the figure, the multi-layer overlap strategy through materialized overlap views outperforms no-overlap by a factor of almost two! Indeed, in order to compute the volume-density of the points close to the boundary, without overlap, we may need to load and process up to $3^N - 1$ neighboring chunks, where N is the number of dimensions (*e.g.*, 26 chunks for the 3D dataset). In contrast, the multi-layer strategy loads and processes only thin layers of overlap data as needed. We observe the same trend on the 6D dataset

In this experiment, materialized views also outperform multi-layer using the two-level storage because views can load overlap layers thinner than tiles. Similarly, single-layer overlap loses because it does not have this flexibility in choosing the overlap granularity to load and process. For completeness, in this experiment, we also show the performance when overlap data is stored directly inside chunks as suggested in related work [87, 91]. The performance in this case is even worse than no-overlap. The reason is that the no-overlap case loads and processes neighboring chunks only when

Figure 2.10 Volume-density application on 3D dataset with and without overlap. Multi-layer overlap outperforms no-overlap by almost $2X$.



required, but when overlap data is stored inside the chunk, we are forced to process unnecessary overlap data.

Multi-layer overlap can outperform single-layer overlap even when overlap size is perfectly tuned. Indeed, in the volume-density application, we find that 90% of chunks only need 3 out of the 20 layers of overlap, but the maximum number of layers required is 10. This large difference between the average and maximum amount of overlap required is the key reason why multi-layer overlap can outperform single-layer overlap even for a single application (even if we ignore varying overlap needs of different operators).

2.6 Conclusion

We presented the design, implementation, and evaluation of ArrayStore, a storage manager for complex, parallel array processing. For efficient processing, ArrayStore partitions an array into chunks and we showed that a two-level chunking strategy with regular chunks and regular tiles

(REG-REG) leads to the best and most consistent performance for a varied set of operations both on a single node and in a shared-nothing cluster. ArrayStore also enables operators to access data from adjacent array fragments during parallel processing. We presented two new techniques to support this need: one leverages ArrayStore's two-level storage layout and the other one uses additional materialized views. Both techniques significantly outperform approaches that do not provide overlap or provide only a pre-defined single overlap layer. The overall performance gain was up to 2X on real queries and real data from two science domains.

ArrayStore's design focuses on the workload from Section 2.2. It does not consider array updates (Chapter 3) nor iterative operations (Chapter 4). It also does not consider operations that examine input cells across the array to compute the value of an output cell. Such operations do not benefit from overlap and some of them may be difficult to code with a chunk-based API. Finally, we did not study the impact of indexing data inside chunks, which could further accelerate some operations. All these considerations are interesting future work.

Chapter 3

TIMEARR: STORAGE MANAGER WITH EFFICIENT SUPPORT FOR VERSIONING

In many fields of science, researchers often work with multiple versions of the same array data. For example, LSST images in astronomy are 2D arrays of pixel intensities. LSST will undertake repeated exposures over time (Example 1.1.1), then the images form a 3D array of pixel intensities with an additional *version* dimension. Climate models use arrays to describe 3D regions of the atmosphere. They also simulate the behavior of these regions over *time*.

An important requirement that scientists have for parallel array engines such as SciDB is the ability to create, archive, and explore different *versions* of their arrays [102]. Hence, a no-overwrite storage manager with efficient support for querying old versions of an array is a critical component of an array database management system (DBMS).

A no-overwrite array-based engine must support different types of queries over a versioned array: It must support standard queries that retrieve specific array versions, queries that retrieve subarrays at specific versions, and queries that track the history in the form of a series of subarrays across multiple versions. At the same time, all these operations must be performed as efficiently as possible to enable fast data exploration and analysis.

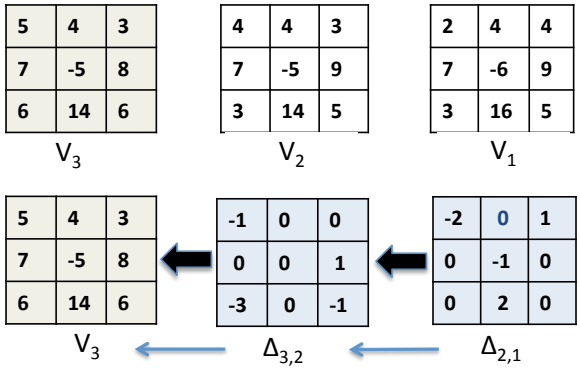
In this chapter, we present TimeArr [98], a new storage manager for array DBMSs that provides a no-overwrite, versioned array storage model together with both precise and approximate time-travel queries over these versioned arrays.

3.1 Challenges, and Contributions

TimeArr storage manager makes the following contributions:

(1) A backward delta versioning system specialized for arrays. At the heart of the TimeArr storage manager, is a new storage model for efficiently representing and querying a versioned array. First, because scientific datasets can grow to be large, the storage model compresses the data using

Figure 3.1 Illustration of a chain of backward delta versions for a 3x3 array. The most recent version V_3 is materialized. Earlier versions are stored in the form of arrays of cell-value differences.



a *backward delta* encoding method: the most recent version of the array is fully materialized and earlier versions only contain differences in cell values between consecutive versions as illustrated in Figure 3.1. $\Delta_{i,i-1}$ represents differences in cell values between version V_i and V_{i-1} of an array. The backward delta technique is known to be an efficient compression method. To illustrate the efficiency of this method for scientific arrays, we store 61 versions of the Global Forecast System (GFS) dataset [67] in TimeArr using four methods. The naïve materialization of all versions takes 65.6MB of space on disk. Storing only the values of cells that change between consecutive versions reduces disk-space utilization to 14MB. Storing differences in cell values between each version V_i and the original version V_z achieves almost no compression compared to storing only materialized versions and takes about 62.7MB. Finally, the backward delta method stores all versions using only 3.5MB, a 19X improvement over the full materialization. Of course, this compression comes at the cost of slower version retrieval. Hence, an important question is how to achieve fast array query processing with this method. Query processing times are also the main reason for always materializing the most recent version of the array, which should be most frequently accessed by applications.

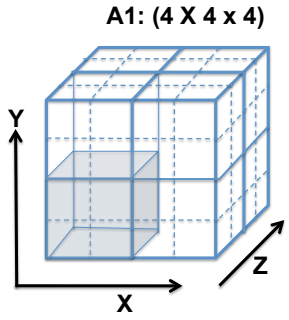
Unlike most other applications of the backward deltas method (*e.g.*, in backup storage [71] or temporal databases [52]), our storage layout is specialized for arrays. The specialization enables TimeArr to achieve both high compression ratios and high query performance. The approach uses three key ideas. First, it applies the notion of array tiling [14, 25, 29, 87, 99] to efficiently limit the changes that must be processed when retrieving old versions of a sub-array. This approach

significantly speeds up query processing. Second, it uses a variety of compressed bitmasks [87] to encode the regions of an array that change from one version to the next and to identify which subset of changes need to be processed to satisfy a user query. This approach both enables better data compression and speeds-up query processing. Third, our storage model uses variable-length delta encoding across tiles, which helps adapt the compression-level to different magnitudes of changes in different regions of an array and yields better compression ratios for the array data. In addition to these three basic methods, to further speed-up query processing over commonly accessed parts of an array, TimeArr lazily adds connections, called *skip links*, between certain non-consecutive versions of an array. TimeArr's *skip links* are similar to regular backward delta versions except that they contain differences in cell values between two non-consecutive versions. TimeArr utilizes skip links similarly to a skip list data structure [63] with the important difference that TimeArr creates links based on version *content* and not version numbers. Additionally, TimeArr creates skip links lazily during version retrieval to reduce the overhead of maintaining this data structure. Finally, it maintains skip links at the granularity of tiles to increase their efficiency. As a result, regions of the array that are fetched more often create more skip links which reduces their version retrieval time. We present TimeArr's detailed storage model in Section 3.3.

(2) Approximate and customizable array-exploration queries. It is well-known that, when first exploring data, users need a quick query turn-around time and are willing to tolerate some inaccuracy to achieve faster time-to-result [21, 38, 80]. To speed-up the exploration of a versioned array, we leverage this observation and introduce the idea of querying *approximate* versions of an array. In our approach, the user specifies both the degree of approximation tolerable and how that approximation should be computed. Hence, TimeArr's approximate exploration is *highly customizable* and *carefully controlled* by the user. The system efficiently answers approximate queries over a versioned array by aggressively leveraging tiling and skip links and also by maintaining short summary statistics that capture the overall changes between different subarrays at different versions. We present the details of approximate version querying and customization in Section 3.4.

(3) Prototype implementation and evaluation with real datasets. We implement the above techniques as a C++ prototype storage manager called TimeArr on a branch of the SciDB array processing engine [87]. We evaluate TimeArr on a real dense array containing 61 snapshots from the global

Figure 3.2 The 4x4x4 array A1 is divided into eight 2x2x2 chunks.



forecast system (GFS) model [67] and a real sparse array containing 9 snapshots from an astronomy universe simulation [51]. For precise queries, without using skip links, TimeArr outperforms the current SciDB version-storage technique [87, 92] (which is also based on backward deltas) by a factor of 1.6X to 6.6X in terms of query processing times and up to 40% in terms of version creation time. Skip links further improve query performance by 75%. Furthermore, when a user retrieves only a small fraction of an array, our approach based on a combination of bitmasks and virtual tiling can cut query times by an order of magnitude. For approximate queries, we show that query times are halved when a user is willing to see data off by at most one array version.

The goal of TimeArr is to efficiently support queries for array regions and versions. We do not study additional indexing techniques over array contents.

3.2 TimeArr Overview

TimeArr is a new storage manager for array database systems. While TimeArr could be integrated with various array systems [28, 87], our design and implementation are based on the SciDB engine [87]. In this section, we present an overview of TimeArr's approach and also TimeArr's core API.

In most array database engines including SciDB, each array is partitioned into chunks, which are small subarrays as illustrated in Figure 3.2. Array chunking is a well-known method for alleviating dimension dependency [94]. Each chunk maps onto a unit of disk IO (either a disk block or larger). TimeArr assumes a chunked array layout. Furthermore, we assume that chunks are regular. That is, each chunk covers the same space in terms of array coordinates. This layout has been shown to

Array Updates
Create(ArrayType Q, ArrayName A)
Append(ArrayName A, ArrayContent C_i)
Precise Queries
Select(ArrayName A, Predicate p, VersionNb j, VersionNb k)
Approximate Queries
Select(ArrayName A, Predicate p, VersionNb j, VersionNb k, ErrorBound B_1, ErrorBound B_2, Granularity g, StatisticID s)

Table 3.1: TimeArr Versioned Array API.

deliver high performance across a wide range of array operations for both dense and sparse arrays (Chapter 2). In SciDB, array chunks are further stored using a column-based representation [87]. TimeArr builds on this column-based, chunked storage layout.

TimeArr supports the operations shown in Table 3.1. The `Create` operation creates an initial, empty array of type Q and named A. The array type includes the specification of the array dimensions, the type of each array cell, and how the array should be both chunked and tiled. This operation only creates array metadata in the SciDB catalog.

The `Append` operation appends a new version to array A. The payload of the append operation, C_i, is a new snapshot of the array content at the new version i. Version numbers are incremented automatically.

When an initial array version is created, its data is broken up into chunks as per the chunking specification in the `ArrayType`. Each chunk is stored in a separate file on disk. For example, the array from Figure 3.2 is stored in eight separate files, one per chunk. Subsequent calls to `Append` add new versions to the array. The new version of each chunk is added to the corresponding file where the earlier version of that chunk is stored. We refer to a file that contains a materialized chunk together with its series of appended versions as a *segment*.

To maintain high performance in the face of a growing number of versions, TimeArr is configured with a maximum segment size F. If a segment grows beyond threshold F for some chunk, a new segment is created for that chunk. Each segment (or file) contains one materialized version of a chunk, which is the most recent version stored in that segment. All prior versions in the same segment are compressed using the backward-delta-based approach described in Section 3.3. Such periodic materialization is a well-known technique adopted in many systems including BigTable [13]. The

selection of the threshold value F depends on various parameters such as chunk size and version content. We do not address the problem of optimizing the value of F in this chapter.

The `Select` operation returns the content of a subarray of A that satisfies predicate p at versions $v \in [V_j, V_k]$. We refer to this operation as *array history selection*. To retrieve data for a single version, the last argument can be omitted. To retrieve the data for the entire array, the predicate p can be omitted. p is a predicate over array dimensions. For example, in the array from Figure 3.2, we could select the first chunk with predicate $x \in [1, 2] \wedge y \in [1, 2] \wedge z \in [1, 2]$. We further present TimeArr's storage model and history selection query implementation in Section 3.3.

TimeArr also supports an approximate variant of array history selection to speed-up early array exploration. As shown in Table 3.1, this variant takes four extra arguments as input. The first one, B_1, is an error bound: if a user requests a single array version, V_j, B_1 serves to specify the maximum tolerable loss in accuracy. The selection of a specific array version thus returns the subarray of A at version V_j that satisfies p. The content returned, $c'_j(p)$, satisfies the error condition: `Difference`$(c'_j(p), c_j(p)) < B_1$, where $c_j(p)$ is the precise version of the corresponding subarray. The computation of the `Difference` function is configurable as we show in Section 3.4. In fact, a user can specify several methods for computing this difference and use different methods in different queries. The `StatisticID` argument to the function specifies which of these methods to use. If not specified, TimeArr uses the *default* `StatisticID`. TimeArr computes version differences at two granularities of tiles or chunks. The user specifies the granularity with the `Granularity` parameter. We further discuss the semantics and computation of these differences in Section 3.4.

When multiple versions are requested, an extra parameter B_2 must also be specified. The `Select` operation then returns the most recent requested version, V_j, within error bound B_1 as above. It also returns a sequence of versions V such that $\forall\ V_u \in V, V_u \in (V_j, V_k] \wedge$ `Difference`$(c'_u(p), c_u(p)) < B_1$. Additionally, for each pair, (V_u, V_w) of consecutive returned versions (*i.e.*, no version in between V_u and V_w is returned), we have `Difference`$(c_u(p), c_w(p)) > B_2$. This operation thus returns the first selected version using the same method as above. It then returns subsequent versions such that each new version's content remains within distance B_1 of the corresponding precise version. Additionally, the query skips over similar versions, returning only the next version that differs by at least B_2. The granularity (tile or chunk) is the same as for B_1. We further discuss this approximate

history extraction in Section 3.4.

3.3 Version Storage and Retrieval

In this section, we present TimeArr's approach to storing and retrieving array version data.

3.3.1 Version Storage

As indicated earlier, TimeArr stores array versions using a backward delta approach: When a new version of a given chunk is appended, TimeArr iterates over both the new version, call it V_j, and the most recent previous version, call it V_{j-1}, of the chunk. It subtracts the cell values in the new chunk from the corresponding cell values in the older chunk. These differences in cell values are called *delta values*. More formally: $d_{j(j-1)k} \leftarrow Subtract(c_{(j-1)k}, c_{jk})$ where d is the delta value and c_{jk} is the k'th cell in the array at version j, assuming that cells are traversed in some order such as the row-major order. The group of delta values for a chunk forms a *delta chunk*. We call the array that wraps all delta chunks the *delta array*. Figure 3.1 illustrates a materialized array version and two delta arrays.

While the basic idea of storing array versions using backward deltas is not new [92], the details of the version data structures that TimeArr uses are different from prior work. In particular, TimeArr's version storage layout uses four key ideas: (1) it partitions chunks into tiles to limit the amount of work when rebuilding an old version of a subset of an array or when answering an approximate query; (2) it uses bitmasks to quickly identify the tiles or cells that changed between two versions; (3) it uses variable-length delta-encoding to capture changes with as few bytes as possible; it also uses run-length encoding (RLE) to compress its bitmasks; (4) it lazily creates skip links to boost the `Select` query performance over time. We now present these four key techniques.

Figure 3.3 shows the internal representation that TimeArr uses to store one segment on disk. Each segment contains one materialized version of a chunk and zero or more delta chunks. The materialized version in the segment could be stored using either a sparse or dense representation, with or without compression, etc. [14, 15, 25, 29, 87, 92, 99]. In this paper, we treat the most recent version as a black box.

TimeArr represents each delta chunk with a structure that we call `VersionDelta`. To speed-up

range-selection and approximate queries, TimeArr divides a delta chunk into a series of *virtual delta tiles*. Each tile is a subarray within the delta chunk. TimeArr represents virtual delta tiles with a structure that we call `TileDelta`. In the rest of the paper, $\Delta_{i,i-1}$ represents the delta values between version V_i and V_{i-1} of either an array, a chunk, or a tile depending on context.

A `VersionDelta` contains a header that summarizes the changes in the version and a payload that holds the actual changed values. The `VersionDelta` header contains a bitmask with one bit per tile (`VDBitmask` in the figure). `VDBitmask` identifies the tiles that have been modified in the new version of the array. Such tiles have their bit set to `true` in the `VDBitmask`. This approach has successfully been applied in the past to compressing array contents [87]. We apply it here for compactly storing changes between array versions.

Tiles that contain changes are stored in a set of `TileDelta` data structures. A `TileDelta` contains the details of changes in one tile. Because `TileDeltas` have variable sizes, TimeArr uses a standard slot-based approach to locate them on disk: for each tile that includes changes, a slot points to the location of the corresponding `TileDelta` on disk (`TileSlotsMap` in the figure). Prior work studied the tuning of chunk/tile shape, size, and layout on disk for a given workload and for regular chunking [74, 88]. In TimeArr, the virtual tile size determines the finest granularity with which the system can do history and approximation queries. Hence, smaller tiles enable finer-grained operations. On the other hand, larger tiles decrease the metadata overhead and preserve the locality of the data (logically close delta values in the array are physically stored together). However, we do not address the problem of tuning the size of virtual tiles in this chapter.

The details of a `TileDelta` structure are illustrated in Figure 3.4. Similar to the `VersionDelta` structure, each `TileDelta` contains a bitmask with one bit per cell (`TileBitMask` in the figure). The `TileBitMask` is a bit vector that indicates which cells in the tile contain any changes. A `TileDelta` also contains a payload that holds the actual delta values. A conceptual view of a TileDelta is shown in Figure 3.5. The first delta value in the list corresponds to the first 1 in the bitmask, the second delta value corresponds to the second 1, and so on. Because we use regular tiling, where each tile covers the same number of cells in each direction as other tiles, mapping from the bitmask bits to the cell coordinates happens efficiently in near constant time.

Following the `TileDelta` header, we store the actual cell updates in the form of backward deltas.

Figure 3.3 Representation of a single array chunk with multiple versions.

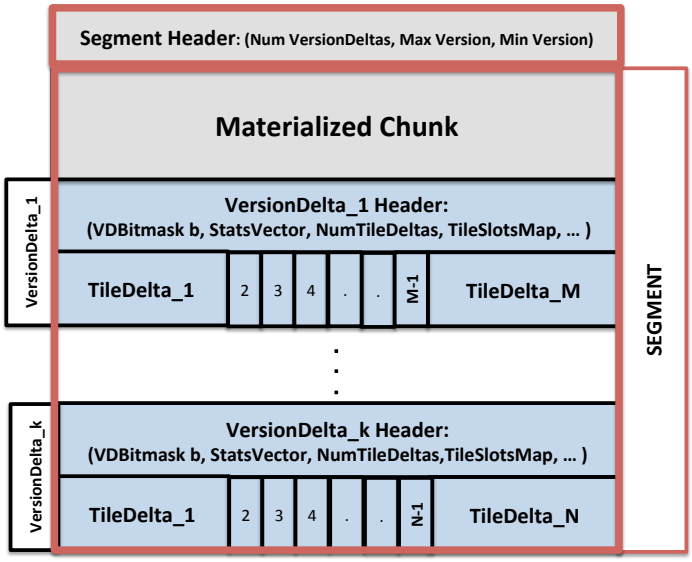
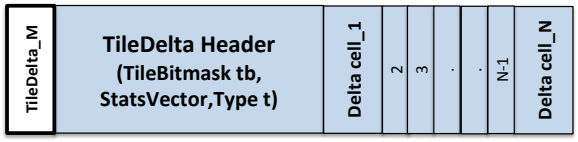


Figure 3.4 TileDelta Layout

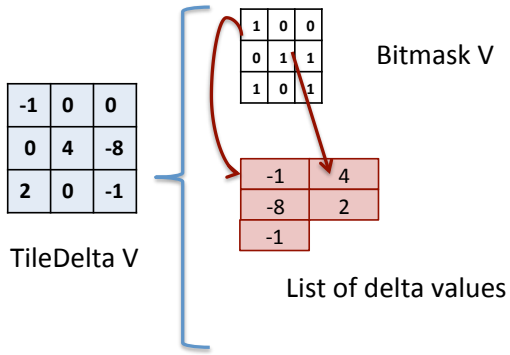


Depending on the magnitude of the changes, we can use a different number of bytes to store the delta values. TimeArr chooses the number of bytes to use to store delta values at the granularity of tiles. The `TileDelta` header includes some information about which encoding is used for delta values (Type *t* in the figure).

To save space, TimeArr uses run-length encoding (RLE) to compact all bitmasks. For example, bitmask `1100111000` is RLE encoded as <1,2> <0,2> <1,3> <0,3>. Values of 1 in the bitmask correspond to cells that were updated.

In addition to bitmasks, `VersionDelta` and `TileDelta` headers also include summary vectors, called `StatsVectors`. We describe the `StatsVectors` in Section 3.4, when we discuss customization and approximation.

Figure 3.5 Internal structure of a TileDelta `v` in TimeArr. The bitmask is represented as a 2D array only for illustration purposes.



3.3.2 Skip Links

Initially, the data in a segment corresponds to a fully materialized version of a chunk and a series of consecutive delta chunks. If V_r is the materialized version in the segment then any older version, V_k, of the chunk in the segment is rebuilt as follows: $V_k = V_r + \sum_{i=k+1}^{r}(\Delta_{i,i-1})$ where the "+" operator applies all delta values in one version of the chunk by invoking an `Add` function for each cell:

$c_{(j-1)k} \leftarrow Add(c_{jk}, d_{j(j-1)k})$

where c are cell values, d is a delta value, k represents the k'th cell, and j and $j-1$ represent two consecutive versions.

The version retrieval time thus grows linearly with the number of versions in a segment. One can use skip lists [63] to maintain the retrieval time for any version of a chunk below $\log(|V|)$ where $|V|$ is the number of versions in a segment. That is, a segment should contain the VersionDelta for consecutive versions but it should also contain additional VersionDeltas for each pair of versions 2^i versions apart. Extra VersionDeltas, however, would significantly increase storage costs. Additionally, as we discussed in Section 3.1, delta chunks for non-consecutive versions that are far apart do not provide much compression compared to materializing the actual array version. Finally, skip lists would significantly increase the time to append a new version due to the creation of multiple extra delta chunks.

To avoid these limitations yet benefit from "shortcut links", TimeArr uses what we call *skip links*,

inspired by the skip list technique, to cut the version retrieval costs by *skipping* over multiple versions in one step. To maximize the benefits of these links, TimeArr defines them at the granularity of tiles. The fundamental differences between a skip list and TimeArr's skip links are that (1) skip links *replace* some of the consecutive delta tiles, ($\Delta_{i+1,i}$), with non-consecutive ones, $\Delta_{j,i\ j>i+1}$, and (2) skip links are established only between *similar* versions; that is, only when $\Delta_{j,i}$ is backward delta encoded more compactly than $\Delta_{i+1,i}$:

$$\texttt{sizeof}(\Delta_{j,i}) < \alpha \times \texttt{sizeof}(\Delta_{i+1,i}) \tag{3.1}$$

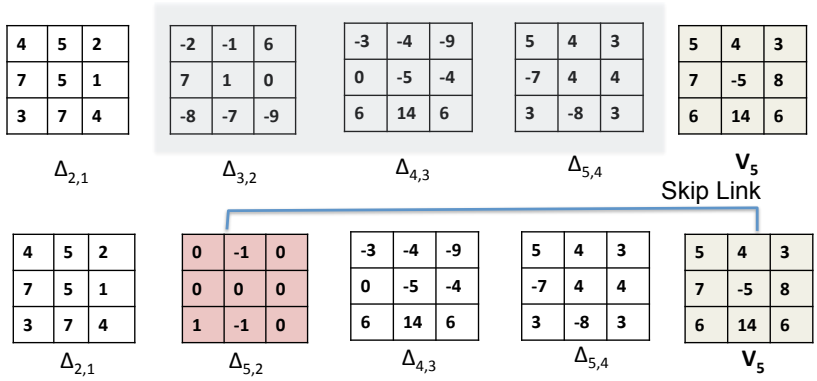
where `sizeof` returns the size of an object in bytes and $\alpha \in [0, 1]$ is a tunable parameter that ensures skip links are created between similar tiles rather than between arbitrary ones. We use $\alpha = 0.9$ in our experiments, which we find to suffice to filter out spurious skip links. An abstract example of skip links is shown in Figure 3.6.

To decide which tile versions to consider for replacement, TimeArr could enumerate all possible $\Delta_{j,i}$ combinations and verify the condition in Equation 3.1. This approach, however, would be computationally expensive because of the large number of version combinations. Instead, we propose to consider only the *linear* sequence of links. That is, given a most recent version V_r, TimeArr only considers adding skip links of the form $\Delta_{r,i\ \forall i<r}$. This approach is significantly less expensive computationally because it considers fewer options but also because it can compute these options incrementally. Indeed, TimeArr reuses the computation spent on a previous candidate skip link $\Delta_{r,i}$ to recursively construct the new candidate skip link $\Delta_{r,i-1}$. At the same time, we hypothesise (and experimentally demonstrate in Section 3.5) that this approach retains the most useful links, since TimeArr always starts from V_r when fetching older versions.

An important design decision for TimeArr is *when* to create skip links. One approach is to exhaustively consider all linear skip links every time a new version is appended to a chunk. A less expensive variant is to compute skip links only every T new versions appended, where $T > 1$. We study the overhead and gain of different values of T in Section 3.5 for version insertion and retrieval.

A third approach is to identify skip links lazily when executing selection queries that retrieve old

Figure 3.6 $\Delta_{5,2}$ is a skip link from V_5 to V_2. $sizeof(\Delta_{5,2}) < \alpha \times sizeof(\Delta_{3,2})$. So $\Delta_{3,2}$ is replaced with $\Delta_{5,2}$ in the chain of backward deltas. Note that $V_2 = V_5 + \Delta_{5,2}$ and $V_1 = V_5 + \Delta_{5,2} + \Delta_{2,1}$.



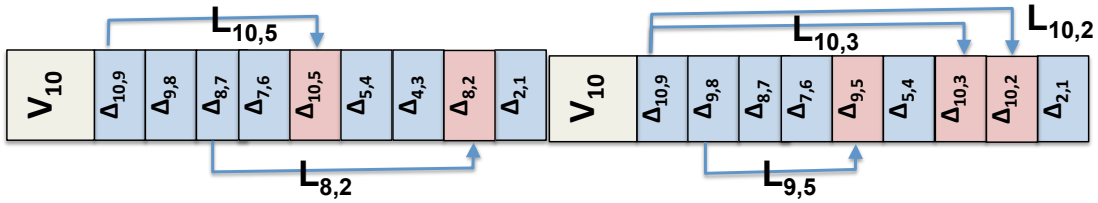
versions of sub-arrays. The approach works as follows: Consider a segment with materialized version V_r and the goal is to retrieve version V_i. To test potential linear skip links, TimeArr reconstructs V_i as $V_i = V_r + \Delta_{r,i}$ and it computes $\Delta_{r,i}$ incrementally by computing each intermediate linear skip link, $\Delta_{r,k\ \forall k\ i \leq k < r}$, where V_r is the materialized version in the segment. This approach thus significantly reduces the overhead of finding skip links at the expense of not being able to use this optimization the first time that an old version is retrieved. To further limit overheads, while retrieving old versions, TimeArr keeps track of deltas $\Delta_{r,i}$ that have already been explored as potential skip links. For example, if TimeArr issues two consecutive selection queries to retrieve V_i, only the first one involves the exploration of possible skip links.

Algorithm 4 describes the tile-based skip link creation procedure. In the algorithm, after $\Delta_{i+1,i}$ is replaced with skip link $\Delta_{j,i}$ for tile t (Line 9), TimeArr puts a *lock* on all the deltas $\Delta_{k+1,k}$ $i < k < j$ at tile t. Delta versions that are locked are not eligible to be replaced with any other skip links (which is one reason why spurious links should be avoided by tuning the α parameter). Locks are at the granularity of tiles and are not revertible. This constraint is reflected in Algorithm 4 at Line 8 and line 10. The reason TimeArr locks the deltas is to avoid *overlapping skip links* as illustrated in Figure 3.7(a). If skip links overlap, TimeArr can reach a dead-end if it does not apply the correct combination of skip links during version retrieval. Locking certain tiles prevents this complication and simplifies the version retrieval algorithm.

Algorithm 4 Skip Links Creation Procedure for One Tile

Input: Materialized Version V_r, Delta versions $\Delta_{i+1,i}$, Target Version Index k.
Output: V_k.
$i \leftarrow r - 1$
$\Delta_{r,x} \leftarrow \phi$
while $i \geq k$ **do**
    $\Delta_{r,x} \leftarrow \Delta_{r,x} + \Delta_{i+1,i}$
    if $\Delta_{r,x} \leq (\alpha \times \Delta_{i+1,i})$ **then**
        if $\Delta_{i+1,i}$ is not *locked* **then**
            Replace $\Delta_{i+1,i}$ with $\Delta_{r,x}$
            Lock all the $\Delta_{n+1,n}$ $i < n < r$
        end if
    end if
    $i = i - 1$
end while
$V_k \leftarrow V_r + \Delta_{r,x}$

Figure 3.7 Valid *v.s.* Invalid states of skip links. TimeArr must avoid overlapping skip links. The *lock* mechanism prevents Invalid state (a) by prohibiting $L_{10,5}$.



(a) Invalid State: V_1 and V_2 are accessible only if TimeArr applies $L_{8,2}$ and not $L_{10,5}$.

(b) Valid State: All versions are accessible through any combination of links.

3.3.3 Query Processing

To support a selection query that retrieves a specific version of an array chunk, TimeArr first selects the set of files –with one file per array chunk– that contain the desired version. Within each file, TimeArr starts from the most recent materialized chunk version and applies all the changes backwards until it rebuilds the version of interest.

If the selection query includes a range predicate, TimeArr leverages its virtual tiles to identify and process only changes that fall within the region of interest.

Each delta tile $\Delta_{j,j-1}$ keeps track of a constant number of skip links $\Delta_{j,i}$ $_{j-1>i}$ that TimeArr can leverage at version V_j to skip directly to an older version V_i. Each $\Delta_{j,j-1}$ stores the number of

versions that $\Delta_{j,i}$ skips as $L_{ji} = (j - i - 1)$ as shown in Figure 3.7. In our experiments keeping track of a small number of L_{ji}'s, $L = 3$, sufficed to hold all skip links for $\alpha < 0.9$. Right before applying $\Delta_{j,j-1}$, TimeArr checks for potential skip links to leverage. TimeArr selects a link that skips the most versions while still landing before or at the desired version. For example in Figure 3.7(b), in order to retrieve version V_1, TimeArr chooses $L_{10,2} = 7$ at V_{10} and skips 7 delta tiles until it reaches $\Delta_{10,2}$ which means $V_1 = V_{10} + \Delta_{10,2} + \Delta_{2,1}$. These choices are performed separately for each tile.

3.4 Approximate Queries

In this section, we present TimeArr's approach to efficiently supporting approximate queries.

3.4.1 Distance between Versions

We recall from Section 3.2 that, when a user requests the approximate content $c'_j(p)$ of the subarray satisfying predicate p at version number j, the user specifies the maximum tolerable error in the form of an error bound B_1. The system guarantees that the data returned will satisfy the condition `Difference`$(c'_j(p), c_j(p)) < B_1$. The difference between two subarrays can be computed at the granularity of tiles or chunks as requested by the user. The semantics are as follows:

$$\texttt{Difference}(c'_j(p), c_j(p)) < B_1 \;\; iff \tag{3.2}$$
$$\forall \text{tiles or chunks} \;\; c'_{jk} \in c'_j \;\; \texttt{Distance}(c'_{jk}, c_{jk}) < B_1$$

where $c_j(p)$ is the exact content of the subarray at version j and $c_{jk}(p)$ is the exact content of tile or chunk k in that subarray. The computation includes tiles or chunks that partially overlap the subarray $c_j(p)$.

Similarly, the `Difference` between subarrays is equal to B_1 if the `Distance` between all pairs of tiles or chunks is equal to B_1. If the `Difference` is neither less than B_1 nor equal to B_1, then it is considered to be greater than B_1.

`Distance` functions in TimeArr are implemented in a manner analogous to aggregation functions in OLAP data cubes [34] or parallel aggregations [107]. The distance between two tiles is computed by aggregating the delta values of their cells as shown in Listing 1: the `Distance` function takes two

Listing 1 Distance Function at the Granularity of Tiles

```
// A1 and A2 are two versions of the same tile
double Distance (Subarray A1, Subarray A2)
  Instantiate Statistics object s.
  s.initialize()
  Iterate over all pairs of matching cells (c1,c2)
  where c1 in A1 and c2 in A2 in lock step:
      delta = s.subtract(c1,c2)
      s.process(delta)
  return s.finalize()
```

Listing 2 Distributive Distance Function at the Granularity of Chunks

```
// A1 and A2 are two versions of the same chunk
double Distance (Subarray A1, Subarray A2)
  Instantiate Statistics object s.
  s.initialize()
  Iterate over all pairs of matching tiles t1 and t2 where
  t1 in A1 and t2 in  A2 in lock step:
      delta = Distance(t1,t2)
      s.merge(delta)
  return s.finalize()
```

versions of the same tile as input ($A1$ and $A2$). It iterates over the two versions and computes the delta value for each pair of cells. The *subtract* method used here is the same as the one introduced in Section 3.3. It then accumulates these differences using a standard aggregation method.

If the difference computation is at the granularity of chunks, to avoid tedious re-computations, TimeArr requires that the `Distance` function be distributive as defined by Gray *et al.* [34]: max(), count(), and sum() are all distributive. That is, to compute the `Distance` of two chunks, TimeArr aggregates the `Distance` of the underlying tiles as shown in Listing 2.

The user can redefine the subtract and aggregate operations involved in these distance computations as we describe shortly.

TimeArr also requires the `Distance` function to be a *metric* and thus to satisfy the triangle inequality:

Listing 3 Statistics Interface

```
interface Statistics
  CellValue add (CellValue, CellValue)
  CellValue subtract(CellValue, CellValue)

  void initialize()
  process(CellValue c)
  merge(Statistics s2)
  double finalize()
```

$$Distance(A1, A3) \leq Distance(A1, A2) + Distance(A2, A3) \tag{3.3}$$

where A_i is a subarray. An example of a metric is a `Distance` function that computes the maximum delta value for all cells in the array.

At the core of these `Distance` functions is the `Statistics` object, which defines how the delta values are computed and aggregated. To implement a new `Distance` function, a user only needs to provide a new class that implements the `Statistics` interface as shown in Listing 3.

The `add` and `subtract` methods operate on delta values as described in Section 3.3. `CellValue` can be any numeric atomic type including integer and real.

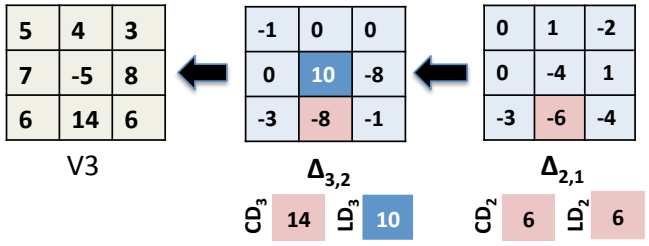
TimeArr allows users to provide multiple classes that implement the Statistics interface. TimeArr also provides a *default* `Distance` function using a *default* `Statistics` class that computes a value difference for `subtract` and a value sum for `add`. It also maintains the *absolute* maximum delta value across versions as the aggregate distance returned by finalize.

Next, we present how TimeArr uses these `Distance` functions to answer approximate queries.

3.4.2 Approximate Version Selection

Given a segment with a fully materialized chunk version V_r and VersionDeltas for earlier versions V_{r-1} down to V_z (the original version in the segment), the goal is to return some desired version V_j within an error bound B_1.

Figure 3.8 A 3x3 array with 3 versions. `V3` is materialized and `V2` and `V1` are backward delta encoded. CumDiff (CD) and LocDiff (LD) are calculated for the two $\Delta_{3,2}$ and $\Delta_{2,1}$. Highlighted cells are the ones to contribute to the CD and LD calculations, which use the default distance (maximum absolute difference between any two cells). The VersionDelta for $\Delta_{3,2}$ contains CumDiff_3 and LocDiff_3. Similarly, the VersionDelta for $\Delta_{2,1}$ contains CumDiff_2 and LocDiff_2.



In the absence of approximation, TimeArr will start with V_r and it will apply delta chunks in sequence (using skip-links when possible) until it gets back to version V_j. With approximation specified at the granularity of tiles, for each tile separately, we want to stop the delta application process as soon as we reach a version V'_j that satisfies the error condition: $\texttt{Distance}(c_{j'k}, c_{jk}) < B_1$, where c_{jk} is the content of tile k at version j and $c_{j'k}$ is the content of that same tile at version j'. If the approximation is specified at the granularity of chunks, TimeArr returns a set of tiles in the approximate chunk $c_{j'k}$ that all have the same version j' and it checks the error condition at the granularity of the whole chunk.

The key question is how to efficiently verify these error conditions? It is impractical to compute the `Distance` function between all versions of each tile in a chunk. Instead, TimeArr computes only two distances for each version V_u: $\texttt{Distance}(\texttt{c}_{\texttt{uk}}, \texttt{c}_{\texttt{u}-1,\texttt{k}})$ and $\texttt{Distance}(\texttt{c}_{\texttt{uk}}, \texttt{c}_{\texttt{zk}})$. We call the former distance the $\texttt{LocDiff}_{uk}$ or *Local Difference at version u and tile index k* because it is a difference between consecutive versions . We call the latter distance the $\texttt{CumDiff}_{uk}$ or *Cumulative Difference at version u and tile index k* because it is the distance to the oldest version in the chunk.

For each new version V_u appended to a chunk, TimeArr computes $\texttt{CumDiff}_{uk}$ and $\texttt{LocDiff}_{uk}$ at the granularity of tiles and stores the results in the `TileDelta StatsVector` for version V_{u-1} (since version V_u will be materialized). Figure 3.8 illustrates the `CumDiff` and `LocDiff` computation for a small array. Finally, TimeArr merges these `CumDiff` and `LocDiff` values for all tiles in a chunk and stores the chunk-level `CumDiff` and `LocDiff` in the `VersionDelta StatsVector`.

Hence, the `StatsVector` is a vector of pairs `(CumDiff,LocDiff)`, with one pair for the system's default `Statistics` object and extra pairs for all the user-defined `Statistics` objects. In the API in Table 3.1, the arguments that refer to statistics are indexes into the StatsVector.

To verify that the error condition is satisfied, TimeArr leverages the fact that `Distance` is a metric and verifies two conditions. First, since $\mathrm{CumDiff}_{j'k}$ is defined as `Distance`($c_{j'k}$,c_{zk}), and the distance function is a metric, we have:

$$\texttt{Distance}(c_{j'k}, c_{jk}) \leq \mathrm{CumDiff}_{j'k} + \mathrm{CumDiff}_{jk} \tag{3.4}$$

Therefore:

$$\text{IF } \mathrm{CumDiff}_{j'k} + \mathrm{CumDiff}_{jk} \leq B_1 \Rightarrow \texttt{Distance}(c_{j'k}, c_{jk}) \leq B_1 \tag{3.5}$$

Second, TimeArr performs a similar check using LocDiffs:

$$IF \sum_{u=j'}^{j} \mathrm{LocDiff}_{uk} \leq B_1 \Rightarrow \texttt{Distance}(c_{j'k}, c_{jk}) \leq B_1 \tag{3.6}$$

If either condition holds, $c_{j'k}$ is an approximate version of c'_{jk} that satisfies the error threshold B_1, which avoids further processing of tile k until version V_j.

Algorithm 5 shows how TimeArr utilizes `CumDiff` and `LocDiff` in Equation 3.5 and 3.6 to answer approximate selection queries of the form: "Select version V_j of array A with predicate p and ErrorBound B_1". For simplicity, the algorithm is only described for the error computation at the tile granularity, but it follows a similar description for the chunk granularity. For each tile, Algorithm 5 finds the version $V_{j'}$ that satisfies either of the inequalities in Equation 3.5 or 3.6. Then it reconstructs and updates C'_j as the approximate version content. It repeats the process for all the tiles separately.

Algorithm 5 uses two bitmasks `ChunkRangeBitmask` and `TileRangeBitmask` that keep track of the chunks and tiles that require to be processed further toward version j. Whenever the `CumDiff` or the `LocDiff` of a given tile satisfies the inequality in Equation 3.5 or 3.6, respectively, the corresponding bit value in `TileRangeBitMask` is set to 0, which avoids further triggers of the `ApplyDelta()` function for the same tile. The `ApplyDelta(`c_{jk}`,`C'_j`)` executes the `Add()` function on

Algorithm 5 Approximate Selection Queries

Input: ArrayName A, Predicate p, End VersionNumber j, ErrorBound B_1.
Output: $C_j^{'}$, approximate content of A at V_j.
$C_j^{'} \leftarrow V_r$, $i \leftarrow r$ //current VersionNumber i, materialized VersionNumber r.
ChunkRangeBitmask bit set for chunks with cells that satisfy p.
TileRangeBitMask bit set for tiles with cells that satisfy p.
while $i \geq j$ **do**
  for all Chunks C in A **do**
    $c \leftarrow$ chunk index C
    if ChunkRangeBitMask.getBit(c) == $false$ **then**
      continue.
    end if
    for all Tiles T in c **do**
      $t \leftarrow$ tile index T
      if TileRangeBitMask.getBit(t) == false **then**
        continue.
      end if
      if $\mathrm{CumDiff}_{\mathrm{it}} + \mathrm{CumDiff}_{\mathrm{jt}} \leq B_1$ or $\sum_{u=j}^{i} \mathrm{LocDiff}_{ut} \leq B_1$ **then**
        TileRangeBitMask.unsetBit(t).
      end if
      ApplyDelta($C_j^{'}$,T)
    end for
    if TileRangeBitMask IS ALL ZERO **then**
      ChunkRangeBitMask.unsetBit(c).
    end if
  end for
  if ChunkRangeBitMask IS ALL ZERO **then**
    $break$.
  end if
  $i = i - 1$
end while

all the corresponding pairs of cells in c_{jk} and $C_j^{'}$.

We include $\texttt{CumDiff}_u$ together with $\texttt{LocDiff}_u$ for the approximate version selection computation because it significantly improves the bound. However, one challenge with this approach lies in the efficient computation of the $\texttt{CumDiff}_u$ values. $\texttt{CumDiff}_u$ corresponds to $\texttt{Distance}(V_u, V_z)$ where V_u is the most recent version and V_z is the original version in the segment. In order to calculate $\texttt{CumDiff}_u$, we need to compute $\texttt{Distance}(V_u, V_z)$ at the granularity of tiles and chunks, which means that we need to have a helper `VersionDelta` that keeps track of delta values corresponding to $\Delta_{u,z}$. We name this auxiliary `VersionDelta` aux. At version V_u insertion time, in addition to the regular computation of $\Delta_{u,u-1}$, TimeArr applies $(\Delta_{u,z} + \Delta_{u,u-1})$ to update delta values in the aux

VersionDelta. Unlike $\texttt{CumDiff}_u$, $\texttt{LocDiff}_u$ values are easy to compute during version insertion since they aggregate delta values between consecutive array versions.

3.4.3 Approximate History Selection

LocDiff also serves to skip over similar versions during approximate history selection. These are versions for which $\texttt{Difference}(V_{u+1}, V_u) \leq B_2$, which is directly captured by the LocDiff values.

Algorithm 6 shows the details of how TimeArr extracts approximate history at the granularity of tiles (algorithm for chunk granularity is very similar). The algorithm proceeds in two phases. In the first phase, TimeArr extracts the header information using the statisticsID s specified by the user. From the header information, TimeArr extracts all the LocDiff values at the granularity of either tiles of chunks as requested by the user. The algorithm then runs a standard SciDB query to identify all versions V_u that differ by more than B_2 from their successor V_{u+1}. The query issued on line 6 in Algorithm 6 captures the maximum variation between adjacent versions and it checks if the difference is high enough to satisfy the lower-bound constraint B_2. TimeArr outputs the version numbers of these versions into a variable named *res*. Recall that TimeArr checks B2 at the same granularity as B1.

The second phase retrieves the actual version contents using the version numbers and a bulk approximate selection query that scans all past versions and returns the desired ones with the required degree of precision B_1.

3.5 Evaluation

In this section, we evaluate TimeArr's performance on two real datasets and two synthetic datasets. All experiments are performed on dual quad-core 2.66GHz Intel/AMD OpteronPentium-based machines with 16GB of RAM running RHEL5. We use the following datasets.

Astronomy Universe Simulation dataset (Astro). The first dataset is the output of an astrophysical simulation (see Chapter 7 for a detailed description of this dataset). We use 9 snapshots from this dataset where each snapshot is 1.6 GBs in size and represents the universe as a set of particles in a 3D space. To represent the data as an array with integer dimensions, we create a $(500 \times 500 \times 500)$ array and project the array content. Following SciDB's column-based array representation, we perform all experiments on the array containing the data for the mass attribute of the particles. We divide this

Algorithm 6 Approximate History Queries

Input: ArrayName A, Predicate p, Start VersionNumber k, End VersionNumber j, ErrorBound B_1, ErrorBound B_2, StatisticID s.
Output: A sequence of contents C containing all matching version contents $C_j^{'}$.

PHASE ONE: Header Information Extraction.

Extract header info using StatisticID s for the tiles that match p from V_k to V_j.

Store result in array A_{head}. // Schema: $A_{head}\{CumDiff, LocDiff\}$[v][t]
// Extract all versions that meet the B_2 bound (v is VersionNumber, t is TileIndex):
res = SELECT v FROM A_{head} WHERE EXISTS (SELECT t FROM A_{head} WHERE $A_{head}[v][t].LocDiff \geq B_2$)

PHASE TWO: Bulk Approximate Version Selection.
for all i in res **do**
    $C_i^{'} \leftarrow$Select(A_p, p, i,B_1,s)
    C.add($C_i^{'}$)
end for

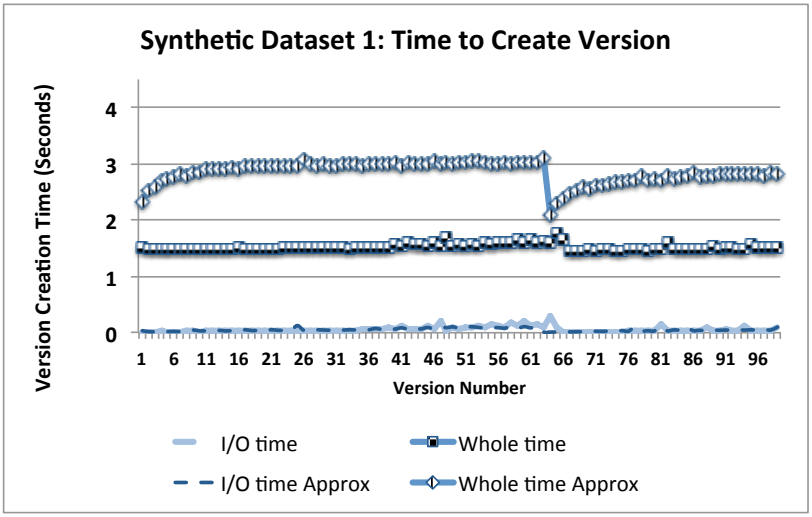
array into 8 chunks, each containing one eighth of the logical size of the array and each having 1000 virtual tiles.

Global Forecast System Model dataset (GFS). The second dataset is the output of a flow cytometer from oceanography from the National Oceanic and Atmospheric Administration (NOAA) [67] that is also described in Chapter 7. We use this dataset for a total of 61 versions, each about 1MB in size. Each grid is a (720×360) two dimensional array (one array in one chunk) and we consider 100 virtual tiles for this single chunk.

Gaussian Distribution (Synthetic dataset 1). The synthetic dataset comprises a single, dense two-dimensional array chunk with 1000×1000 cells. The chunk is divided into one hundred 100×100 tiles unless mentioned otherwise. We create synthetic versions by randomly updating the array. The probability that a cell will be updated follows a normal distribution centered at coordinate $[500][500]$. For a normal distribution, 99.8% of all values fall within 3 standard deviations of the mean. Hence we pick sigma to be $\frac{1}{6}$ of the dimension length. Each update consists of the addition of a marginal value ($<$127) . Each snapshot is 8 MB in size.

Uniform Distribution (Synthetic dataset 2). The synthetic dataset follows the same description as the Gaussian Distribution except the probability that a cell will be updated follows a uniform

Figure 3.9 Time to create 100 versions of a two-dimensional array with normally distributed updates. A new segment is initialized at version 65 in the non-approximate setting and version 62 when approximations are enabled. Each new version adds a constant overhead. I/O overhead is insignificant.



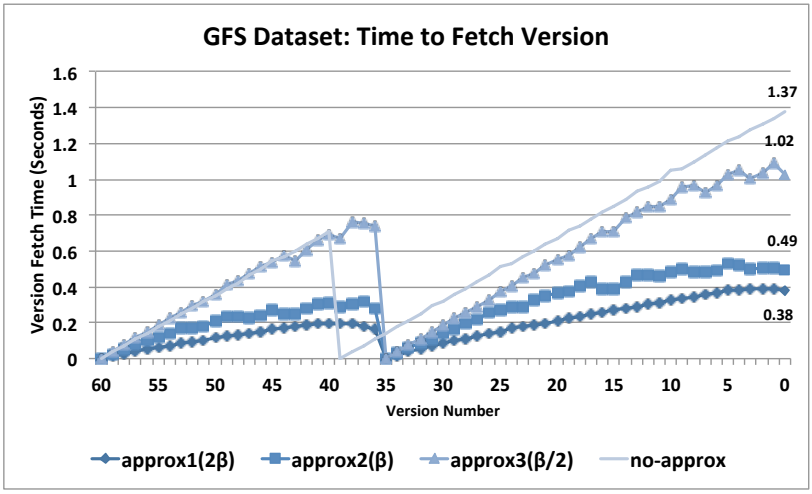
distribution.

3.5.1 Basic Version Creation and Retrieval Performance

We first evaluate the performance when appending new versions to an array. Figure 3.9 shows the time to create 100 new versions for the first synthetic dataset. Each new version adds a constant overhead (1.5 seconds) in the non-approximate setting. The overhead is constant primarily because the total number of updates is approximately the same for each version. As we show later, the version creation time grows almost linearly with the number of updated cells per version. The overhead of creating a new version is higher with approximation enabled. The extra overhead comes primarily from updating the aux `VersionDelta` in addition to the main `VersionDelta`, doubling processing times. The I/O times in both cases are insignificant. The CPU cost of computing delta values dominates the runtime. In this experiment, we arbitrarily set the segment size to 48 MBs. When a new segment is created the version creation time with approximation is close to the non-approximate setting primarily because the aux VersionDelta starts-off empty and is thus quick to update. We observe the same trend with the real datasets.

Next, we study the query processing time to fetch each version either precisely or approximately

Figure 3.10 Time to fetch each version in the GFS dataset. β, 2β, and $\frac{\beta}{2}$ refer to the error bounds, where β is the average maximum change observed in two adjacent versions. I/O times are insignificant and not shown. For the GFS dataset the maximum segment size is 12 MBs. The segments reaches its full size at version number 38 and 36 in the non-approximate and approximate settings respectively.



in synthetic and real datasets. For the experiments with approximation, we consider an error bound β equal to the average maximum change observed between any two adjacent versions. With such an error bound, the user may see values that are in aggregate off by at most one array version. As Figure 3.10 shows, the cost of retrieving a version precisely decreases linearly with the version number. With high approximation (error bound β), for the GFS dataset, query times decrease by factors between 25% and 300%. Similarly, we observe that retrieving any version in the synthetic dataset (not shown in the figure) takes half the time or less compared with retrieving the exact version. Even with a small approximation (error bound $\beta/2$), performance gains are above 35%. We observe similar trends for the `Astro` dataset.

Overall, TimeArr's approach to approximate query processing thus adds overhead during version insertion. This overhead, however, is paid only once. At the same time, approximation enables the system to cut query times significantly when users can tolerate approximate results. These savings are repetitive. Interestingly, the performance gains of approximation increase as we query older versions while the version creation overhead remains constant.

We also study the effect of the number of updates on the version creation time. We calculate the

total time to create 50 versions from the `synthetic dataset 2` with different numbers of updates ranging from one thousand to one million updates per version. As expected version creation time grows almost linearly with the number of updates per version. Going from one thousand to one million updates always added 7 to 8 seconds to the total version creation time. We run a similar experiment keeping the number of updates constant, but increasing the updated values. We observe no significant increase in the version creation time (although the size of the version in terms of bytes changed rapidly).

We also evaluate the benefit of using virtual tiles and variable-length delta encoding on the storage space at version creation time and we observed up to 70% space savings compare to the case with no-tile settings and no variable-length delta encoding. The space savings, however, do not come for free. The finest tile settings in the experiment had up to 25% version creation time overhead compare to the no-tile settings.

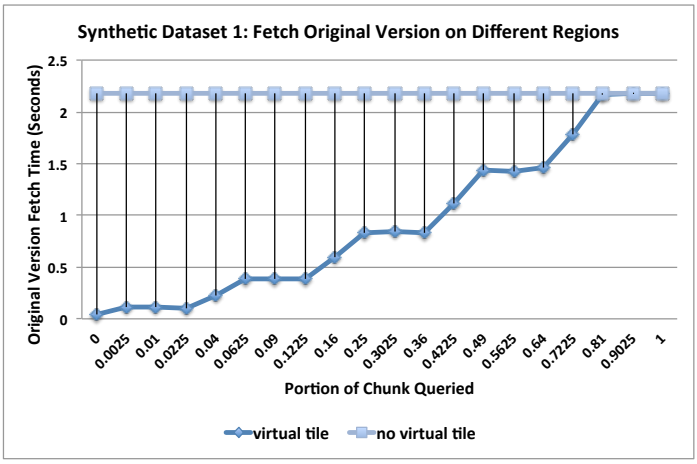
We now evaluate the benefits of using virtual tiles to speed-up historical queries over subsets of a chunk (*i.e.*, range selection queries over array coordinates). Figure 3.11 shows the performance of the following query: `Return the original version of the rectangular subarray [`C_1`;`C_2`]`, where C_1 and C_2 are the upper-left and lower-right corners of a region. In Figure 3.11(a), we use a single chunk with 100 virtual tiles and Synthetic dataset 1. The rectangular regions have the same center as the chunk and range from one tile to the whole chunk. The performance gains that we achieve using virtual tiles depend on the granularity of the tiles and the size of the fetched region. In the tile setting in this experiment, we fetch a single tile 50 times faster than the setting with no virtual tiles used. Even with the window sizes that retrieve as much as 25% of the chunk, TimeArr runs significantly faster than the setting without virtual tiles. Figure 3.11(b) shows the performance of the range selection query on the astronomy dataset. Similar to what we observed in the synthetic dataset, the benefits of using virtual tiles to speed-up historical queries over subsets of an array are significant. The trend is the same for the GFS dataset as well.

3.5.2 Version Retrieval with Links Support

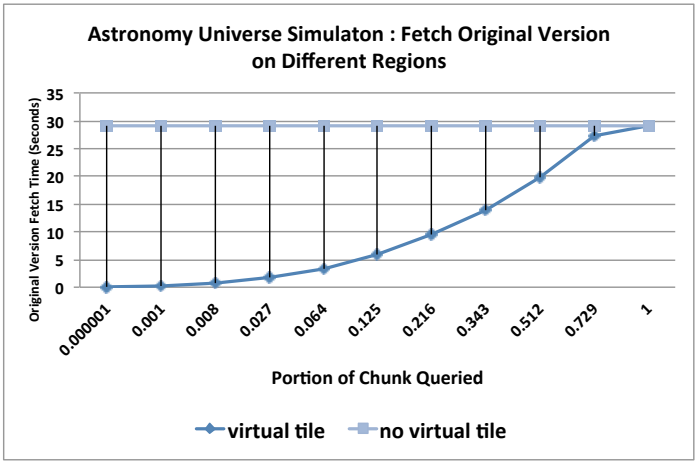
The advantage of the skip link technique is highlighted on datasets such as the *Global Forecast System Model* (GFS) where similar data patterns are repeated at different versions. Figure 3.12 shows

Figure 3.11 Time to fetch the original version where the region window changes from one tile to the whole chunk. (a) Synthetic dataset with 20 versions. (b) Astronomy dataset with 9 versions.

(a)

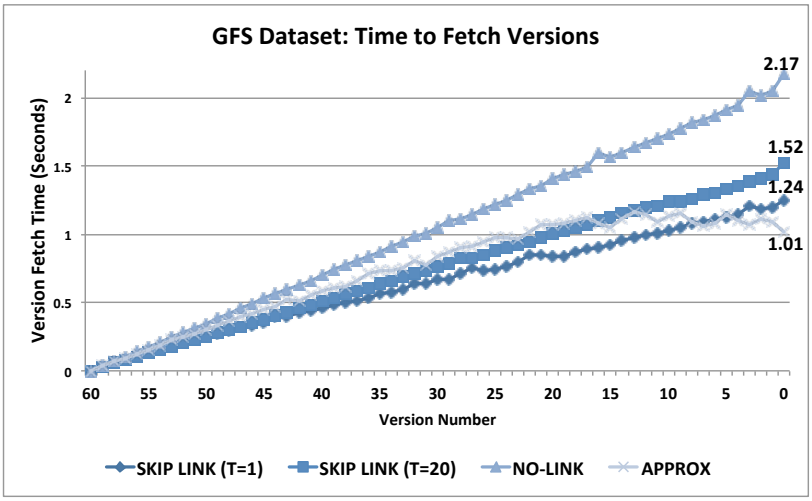


(b)



the time to fetch 60 versions of the GFS dataset. The segment size is chosen such that all the versions reside in one segment. In this experiment, TimeArr periodically computes skip links after each T versions appended. As illustrated in Figure 3.12 the performance gain to fetch the oldest version with skip links is 42% for $T = 20$ and 75% for $T = 1$ compared to the no-link case. However, the skip link computation incurs overhead at version insertion time. Table 3.2 summarizes the overhead for different values of T. Although exhaustive computation of skip links ($T = 1$) improves the performance in Figure 3.12, it incurs significant overhead when inserting new versions. Finding the optimal interval T is left for future work. Instead, TimeArr uses *lazy* computation of skip links whose performance is shown in Figure 3.13. The query workloads, `Q-NORM` and `Q-UNIFORM` are as follows:

Figure 3.12 Time to fetch each version in a single-chunk array with 60 versions. The result with skip link is competitive with approximate result with β error bound.

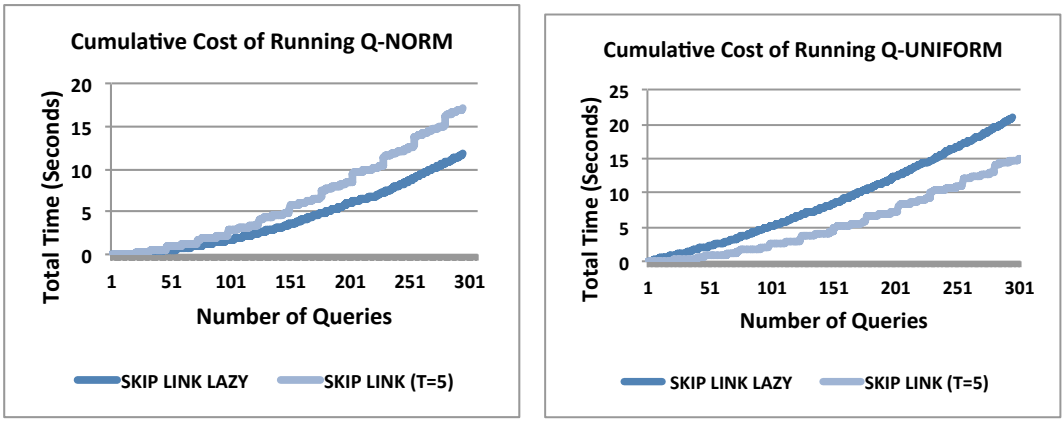


TimeArr appends 61 versions from the GFS dataset in total and between each append operation, we issue 5 original-version retrieval queries (305 queries in total). Each original-version retrieval query only fetches a few tiles from the array. The tiles to be fetched are selected randomly based on either normal distribution (`Q-NORM`) or uniform distribution (`Q-UNIFORM`). Figure 3.13(a) shows the advantage of the *lazy* link computation with the `Q-NORM` workload. `Q-NORM` simulates a workload with a *hot spot* region; *i.e.*, a number of tiles are fetched many times while other tiles are fetched only once. Figure 3.13(a) shows that lazy computation of skips links is better than the skip-link computation at version insertion time with interval $T = 5$. However, this is not true when TimeArr runs the `Q-UNIFORM` workload (Figure 3.13(b)), because the skip-link computation overhead for a specific tile at version fetch time is not paid off later. In the `Q-UNIFORM` workload, many tiles are only fetched once. In Figure 3.13, lazy computation of skip links during version retrieval incurs approximately 2 seconds of overhead in total (not shown in the figure). The algorithm to decide when to compute skip links lazily during version retrieval, when to compute them after certain intervals at version insertion time, and possibly the combination of these two approaches are left for future work.

	NoLink (Lazy)	Link (T=20)	Link (T=5)	Link (T=1)
Add Versions (sec)	13.16	12.00	13.48	14.56
Create Links (sec)	0	3.68	11.11	47.13

Table 3.2: GFS dataset: Skip links overhead at version insertion time. TimeArr computes skip links each T consecutive versions appended.

Figure 3.13 Cumulative query runtime of workloads `Q-NORM` and `Q-UNIFORM`. Skip links are computed either lazily at version retrieval or at version insertion time after each $T = 5$ versions appended.



(a) (b)

3.5.3 Comparison with SciDB

The current SciDB version storage also uses backwards deltas [92]. Unlike TimeArr, however, it represents each `VersionDelta` using two chunks, one with a sparse and the other with a dense representation. Each cell-value in the `VersionDelta` is either in the sparse or dense chunk.

We compare TimeArr to SciDB's current storage manager using the `synthetic dataset 2`. There is thus a total of 10^6 cells in a single-chunk array. We create four synthetic streams of versions: *mass* updates, *medium* updates, *rare* updates, and *very rare* updates that correspond to 10^6,10^5,10^4, and 10^3 updates between each array version respectively. The approximation feature is turned off in all the experiments. Table 3.3 shows the results. TimeArr outperforms SciDB in all four cases. It achieves 40% version creation time savings for medium and mass updates. Table 3.3 also shows that version creation time variation in SciDB is much larger than TimeArr in the case of medium and mass updates. In TimeArr the overhead of adding a new version is constant while this is not the case

updates		mass	medium	rare	very rare
TimeArr(sec)	AVG	1.16	0.60	0.50	0.48
TimeArr(sec)	STD	0.08	0.01	0.04	0.05
SciDB(sec)	AVG	1.80	1.01	0.67	0.55
SciDB(sec)	STD	0.83	0.45	0.85	0.60

Table 3.3: Average time to create one version after appending 50 versions on a two-dimensional array with uniformly distributed updates. TimeArr outperforms SciDB in all four cases.

in SciDB.

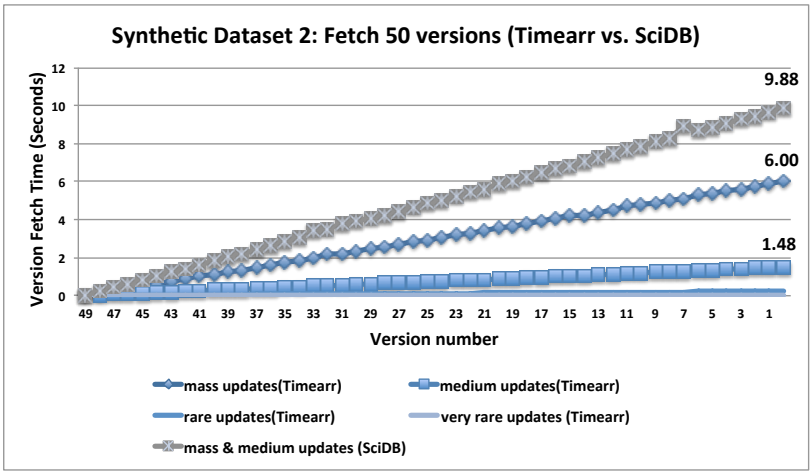
Figure 3.14 shows the query processing performance of both approaches when fetching the whole chunk at a specific, precise version. The chunk has 100 tiles. TimeArr achieves a 1.6X to 6.6X performance gain in terms of query processing compared with SciDB for mass and medium updates. For rare updates, improvement is marginal (it is not shown in the figure). TimeArr's performance gains compared to SciDB come from the fact that SciDB stores delta values in one dense and one sparse chunk for compactness. When fetching a version, SciDB first needs to combine delta values from both representations, which incurs significant overhead. Also, TimeArr uses bitmask techniques to locate changes efficiently, while SciDB needs to iterate over the whole dense delta chunk. Overall, our design decision to have a single storage representation for delta chunks is a key factor for TimeArr's query time performance.

We also studied the advantage of using virtual tiles in TimeArr compared to the current implementation of SciDB. We did a similar experiment as the one in Figure 3.11. We observed two orders of magnitude improvement in TimeArr for regions covering only a few tiles (The trend is similar to Figure 3.11).

3.5.4 Approximate History Query

We now demonstrate the benefits of approximate history query using the GFS dataset. We execute the following example query: `Select(`A_{GFS}`,true,`V_1`,`V_{61}`,54,260)` where A_{in} is the input array. This query asks for all 61 versions of the dataset such that each version is approximately returned with the error bound $B_1 = 54$ and only versions that differ by at least error bound $B_2 = 260$ are returned. 260 is approximately half of the maximum change observed in two adjacent versions. Figure 3.15 shows the result of this query. TimeArr quickly identifies that only 9 versions differ by

Figure 3.14 Time to fetch each version from 1 to 50 on a two-dimensional array with uniformly distributed updates. TimeArr is about 1.6X to 6.6X better than SciDB for mass and medium updates. "Rare updates" and "very rare updates" lines overlap for both systems. Only TimeArr is shown.

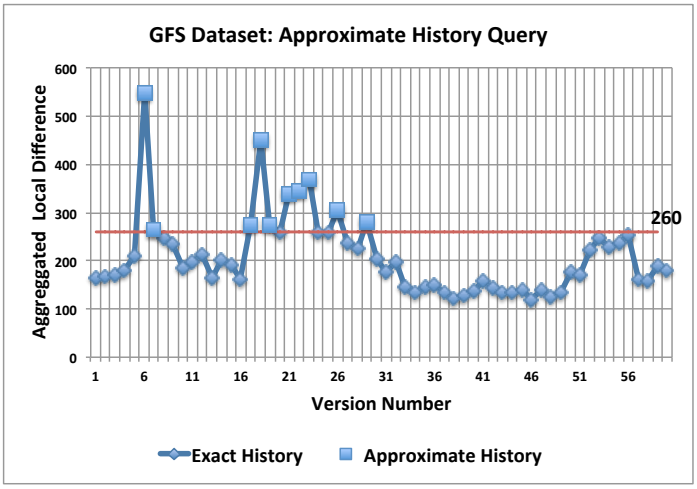


more than the specified threshold and it only requests to approximately fetch these 9 versions. In contrast, with exact history TimeArr has to fetch all the versions. The approximate query runs in less than 7.5 seconds and the equivalent precise query takes 37 seconds to complete, which is a 5X performance difference.

3.6 Conclusion

TimeArr is a new storage manager for an array database. Its key contribution is to efficiently store and retrieve versions of an entire array or some sub-array. TimeArr also introduces the idea of approximate exploration of an array's history. To achieve high performance, TimeArr relies on several techniques including virtual tiles, bitmask compression of changes, variable-length delta representations, and skip links. TimeArr enables users to customize their exploration by specifying both the maximum degree of approximation tolerable and how it should be computed. Experiments with a prototype implementation on two real datasets demonstrate the performance of TimeArr's approach.

Figure 3.15 Approximate history query returns only 9 versions out of 61 with a maximum degree of changes from the previous version greater than 260, while in exact history , TimeArr has to go over all the versions. The performance gain is almost 5X.



Chapter 4

ARRAYLOOP: SCIDB WITH SUPPORT FOR ITERATIVE COMPUTATION

Many data analysis tasks today require iterative processing [30]: machine learning, model fitting, pattern discovery, flow simulations, cluster extraction, and more. As a result, most modern Big Data management and analytics systems (*e.g.*, [53, 108]) support iterative processing as a first-class citizen and offer a variety of optimizations for these types of computations: caching [10], asynchronous processing [53], prioritized processing [59, 109], etc.

The need for efficient iterative computation extends to analysis executed on multi-dimensional scientific arrays. In this chapter, we present the design and implementation of ArrayLoop [62], an extension of SciDB that adds native support for array iterations. ArrayLoop comprises a model for iterative processing in a parallel array engine and followed by three optimizations to improve the performance of these types of computations: incremental processing, mini-iteration overlap processing, and multi-resolution processing.

4.1 Requirements, Challenges, and Contributions

Iterative data analysis on multi-dimensional scientific arrays is ubiquitous. For example, astronomers typically apply an iterative outlier-removal algorithm to telescope images as one of the first data processing steps. Once the telescope images have been cleaned, the next processing step is to extract sources (*i.e.*, stars, galaxies, and other celestial structures) from these images. The source extraction algorithm is most easily written as an iterative process as well. As a third example, the simple task of clustering data in a multi-dimensional array also requires iterating until convergence to the final set of clusters. We describe these three applications in more detail in Section 4.2.

While it is possible to implement iterative array computations by repeatedly invoking array queries from a script, this approach is highly inefficient (as we show in Figure 4.11(a)). The reason is that iterations do a lot of repeated work that can be avoided if the computation keeps track of the states across iterations. Instead of invoking iterative array queries from a script, a large-scale array

management systems such as SciDB should support iterative computations as first-class citizens in the same way other modern data management systems do for relational or graph data.

In this chapter, we address the problem of how to efficiently execute iterative computations over array data. We focus both on single-machine and parallel array processing in a shared-nothing cluster as both are common deployments today. We fist develop a model that captures the essence of a large class of iterative array computations. Using this model, we develop three optimizations that significantly speed-up iterative array computations. The first optimization, which focuses on making iterations incremental, is based on analogous optimizations in relational and graph systems, but we show how it can be pushed all the way to the storage manager in the case of an array system. The following two optimizations, which relate to efficient parallel array iterations and multi-resolution computations, are specific to arrays.

1) Incremental iterative processing: In many iterative applications, the result of the computation changes only partly from one iteration to the next. As such, implementations that recompute the entire result every time are known to be inefficient. The optimization, called *incremental iterative processing* [30], involves processing only the part of the data that changes across iterations. When this optimization is applicable, it has been shown to significantly improve performance in relational and graph systems [30, 59]. This optimization also applies to array iterations. While it is possible to manually write a set of queries that process the data incrementally, doing so is tedious, error-prone, and can miss optimization opportunities. To address this challenge, we develop an approach that enables users to specify their computation at a logical, high-level. The system automatically generates an incremental version of the computation when it is possible. Additionally, we optimize the way in which the system handles all partial results and merges them with the full result at each iteration. We show that several of these optimizations can be pushed all the way down to the storage manager in an array system.

2) Overlap iterative processing: In many array operations, including, for example, cluster finding and source detection, some operations in the body of the loop update the value of certain array cells by using the values of other neighboring array cells. These neighborhoods are often bounded in size. These applications can effectively be processed in parallel if the system partitions an array but also replicates a small amount of overlap cells. In the case of iterative processing, the key challenge lies

in keeping these overlap cells up-to-date. We develop an approach that efficiently updates overlap cells through select bulk-operations.

A subset of applications that leverage overlap data also have the property that overlap cells can be updated only every few iterations. Examples of such applications are those that try to find structures in the array data. They can find structures locally, and need to exchange information only periodically to stitch these local structures into larger ones. We develop an approach that leverages this property to reduce the overhead of synchronizing overlap data. We call this optimization, mini-iterations.

3) Multi-resolution iterative processing: Finally, in many applications the raw data lives in a continuous space (3D universe, 2D ocean, N-D space of continuous variables) and arrays capture discretized approximations of the real data. Different data resolutions are thus possible and scientifically meaningful to analyze. In fact, it is common for scientists to look at the data at different levels of detail. In many applications over such data, it is often efficient to first process the low-resolution versions of the data and use the result to speed-up the processing of finer-resolution versions of the data if requested by the user. Our final optimization automates this approach.

We implement the iterative model and all three optimizations as extensions to the open-source SciDB engine and we demonstrate their effectiveness on experiments with 1 TB of publically-available synthetic LSST images [83]. Experiments show that *Incremental iterative processing* can boost performance by a factor of 4-6X compared to a non-incremental iterative computation for applications that support it. *Iterative overlap processing* together with *mini-iteration processing* can improve performance by 31% compared with SciDB's current implementation of overlap processing in the context of parallel iterative computations that can leverage overlap data. Finally, *multi-resolution optimization* can cut runtimes in half if an application can leverage this technique. Interestingly, these three optimizations are complementary and their benefits can be compounded.

4.2 Motivating Applications

We start by presenting three array-oriented, iterative applications. We use these applications as examples throughout the chapter and also in the evaluation.

Example 4.2.1. Sigma-clipping and co-addition of LSST images (SigmaClip): The Large Synoptic Survey Telescope (LSST [54]) is a large-scale, multi-organization initiative to build a new

telescope and use it to continuously survey the visible sky. The `LSST` will generate tens of TB of telescope images every night. Before the telescope produces its first images, astronomers are testing their data analysis pipelines, storage techniques, and data exploration using realistic but simulated images.

When analyzing telescope images, some sources (a "source" can be a galaxy, a star, etc.) are too faint to be detected in one image but can be detected by stacking multiple images from the same location on the sky. The pixel value (`flux` value) summation over all images is called image *co-addition*. Figure 4.1 shows a *single* image and the corresponding *co-added* image. Before the co-addition is applied, astronomers often run a "sigma-clipping" noise-reduction algorithm. The analysis in this case has two steps: (1) outlier filtering with "sigma-clipping" and then (2) image co-addition. Listing 4 shows the pseudocode for both steps. Sigma-clipping consists in grouping all pixels by their (x,y) coordinates. For each location, the algorithm computes the mean and standard deviation of the flux. It then sets to null (or zero) all cell values that lie k standard deviations away from the mean. The algorithm iterates by re-computing the mean and standard deviation. The cleaning process terminates once no new cell values are filtered out. We refer to this application as **SigmaClip**. □

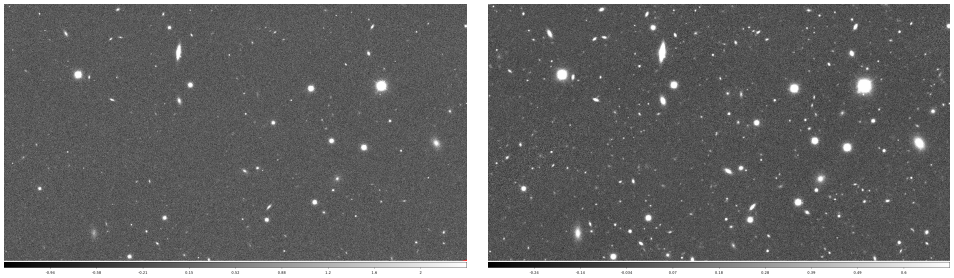
Example 4.2.2. Iterative source detection algorithm (SourceDetect): Once telescope images have been cleaned and co-added, the next step is typically to extract the actual sources from the images.

A simple pseudocode of source detection is shown in Listing 5. Each non-empty cell is initialized with a unique label and is considered to be a different object. At each iteration, each cell resets its label to the minimum label value across its neighbors. Two cells are neighbors if they are adjacent. This procedure continues until the algorithm converges and no more cell value changes. We refer to this application as **SourceDetect**. □

Example 4.2.3. K-means clustering algorithm (KMeans): In many domains, clustering algorithms are commonly used to identify patterns in data. Their use extends to array data. We consider in particular K-means clustering on a 2D array [45]. K-means clustering works as follows: It assigns each cell randomly to one of the k clusters. It computes the centroid of each cluster. It iterates by re-assigning each cell to its nearest cluster. We refer to this application as **KMeans**. □

These applications illustrate two important properties of iterative computations over arrays.

Figure 4.1 Illustrative comparison of one *single* image and its corresponding *co-added* image. There are many faint objects that show up in the co-added image but not in the single image



(a) Single Image (b) Co-added image

Listing 4 Pseudocode for SigmaClip application

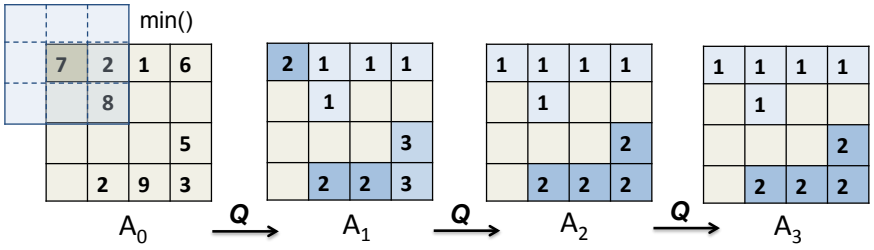
```
Input: Array A with pixels from all the x-y images over time.
//Part 1: Iterative sigma-clipping
While(some pixel changes in A)
 For each (x,y) location
  Compute mean/stddev of all pixel values at (x,y).
  Filter any pixel value that is k
  standard deviations away from the mean
//Part 2: Image co-addition
Sum all non-null pixel values grouped by x-y
```

First, the goal of an iterative computation is to take an array from an initial state to a final state by iteratively refining its content. The `SigmaClip` application, for example, starts with an initial 3D array containing 2D images taken at different times. Each iteration changes the cell values in this array. The iteration terminates when no cell changes across two iterations. Second, the value of each cell at the next iteration is determined by a subset of array cells at the current iteration that can be mathematically described. For `SigmaClip` those are "*all pixel values at the same (x,y) location*". Interestingly, unlike the `SigmaClip` application, where each group of cells at the same (x, y) location influences *many* cell-values at the next iteration, in the `SourceDetect` algorithm any given cell (x, y) is influenced by a *unique* group of cells, which are its adjacent neighbors. These groups of cells partially overlap with each other, which complicates parallel processing as we discuss in Section 4.5.

Listing 5 Pseudocode for SourceDetect application

```
Input: Co-added Array A with uniquely labeled pixels from
       all the x-y images.
Input: int r, the adjacency threshold.
While(some pixel changes in A)
 For each (x,y) location
  Compute the minimum label of all pixel values (x',y')
  with x-r <= x'<= x+r  and  y-r <= y'<= y+r.
  Update (x,y) with the minimum label.
```

Figure 4.2 Iterative array A and its state at each iteration for "iterative source detection" application. $\{Q^{f^\pi,\delta^\pi}_{cells(A)} : \forall c_{i,j} \in cells(A)\ i \in I_1\ \&\ j \in I_2\ \}$ where $I_1 = I_2 = \{1,2,3,4\}$ are the set of dimension values, f^π applies $min()$ aggregate on each group of cells, δ^π simply stores the aggregated value in each cell $c_{i,j}$, and $\pi : (x,y) \rightarrow [x \pm 1][y \pm 1]$. At each iteration, sliding window scans through all the cells. The convergence occurs at iteration 3 when there is no change in labeling between A_2 and A_3



4.3 Iterative Array-Processing Model

We start with a formal definition of an array similar to Furtado and Baumann [32]: Given a discrete coordinate set $S = S_1 \times \ldots \times S_d$, where each $S_i, i \in [1,d]$ is a finite totally ordered discrete set, an array is defined by a d-dimensional domain $D = [I_1, \ldots, I_d]$, where each I_i is a subinterval of the corresponding S_i. Each combination of dimension values in D defines a *cell*. All cells in a given array A have the same type T. $cells(A)$ is the set of all the cells in array A and function $V : cells(A) \rightarrow T$ maps each cell in array A to its corresponding tuple with type T. In the rest of the paper, we refer to the dimension x in array A as $A[x]$ and to each attribute y in the array A as $A.y$.

In SciDB, users operate on arrays by issuing declarative queries using either the Array Query Language (AQL) or the Array Functional Language (AFL). AQL and AFL queries are translated into query plans in the form of trees of array operators. Each operator O takes one or more arrays as input and outputs an array: $O : A \rightarrow A$ or $O : A \times A \rightarrow A$.

In an iterative computation, the goal is to start with an initial array A and transform it through

a series of operations in an iterative fashion until a termination condition is satisfied. The iterative computation on A typically involves other arrays, including arrays that capture various intermediate results (*e.g.*, arrays containing the average and standard deviation for each (x, y) location in the `SigmaClip` application) and arrays with constant values (*e.g.*, a connectivity matrix in a graph application).

One can use the basic array model to express iterative computations. The body of the loop can simply take the form of a series of AQL or AFL queries. Similarly, the termination condition can be an AQL or AFL query.

To enable optimizations, however, we extend the basic array model with constructs that capture in greater details how iterative applications process arrays. We start with some definitions.

Definition 4.3.1. *We call an array iterative if its cell-values are updated during the course of an iterative computation. The array starts with an initial state A_0. As the iterative computation progresses, the array goes through a set of states A_1, A_2, A_3, . . ., until a final state A_N. Note that all A_i have the same schema. In other words, the shape of an iterative array does not change.*

Figure 4.2 shows a (4×4) iterative array that represents a tiny telescope image in the `SourceDetect` application. In the initial state, A_0, each pixel with a flux value above a threshold is assigned a unique value. As the iterative computation progresses, adjacent pixels are re-labeled as they are found to belong to the same source. In the final state A_3, each set of pixels with the same label corresponds to one detected source.

Iterative applications typically define a termination condition that examines the cell-values of the iterative array. In many applications, this condition can be expressed as follows:

Definition 4.3.2. *We say that an iterative array A has converged, whenever $T(A_i, A_{i+1}) \leq \epsilon$ for some aggregate function T. T is the termination condition. ϵ is a user-specified constant.*

In Figure 4.2, convergence occurs at iteration 3 when $\epsilon = 0$ and the termination condition T is the count of differences between A_i and A_{i+1}.

An iterative array computation takes an iterative array, A, and applies to it a computation Q until convergence:

$$A_0 \xrightarrow{Q} A_1 \xrightarrow{Q} \ldots \xrightarrow{Q} A_i \xrightarrow{Q} A_{i+1} \tag{4.1}$$

where Q is a set of valid AQL or AFL queries. At each step, Q can either update the entire array or only some subset of the array. We capture the distinction with the notion of *major* and *minor* iteration steps:

Definition 4.3.3. *A state transition, $A_i \xrightarrow{Q} A_{i+1}$, is a major step if the function Q operates on all the cells in A at the same time. Otherwise it is a minor step.*

The array state $A_{i,j}$ represents the state of the iterative array after i major steps followed by j minor steps. We are interested in modeling computations where each major step can be decomposed into a set of *associative* and *commutative* minor steps that can be evaluated in parallel. That is, a Major step Q_i can be expressed as a set of minor steps q_i such that for any permutation of these steps σ, $Q_i = q_{i,\sigma_1} \cdot q_{i,\sigma_2} \cdots q_{i,\sigma_{n-1}} \cdot q_{i,\sigma_n}$.

The iterative array computation in Equation 4.2 includes $(i+1)$ major steps. The first line illustrates the transition of iterative array A in major steps and the second line illustrates the possible minor steps between two major steps i and $i+1$. Termination condition check always occurs between two states of an iterative array after a major step.

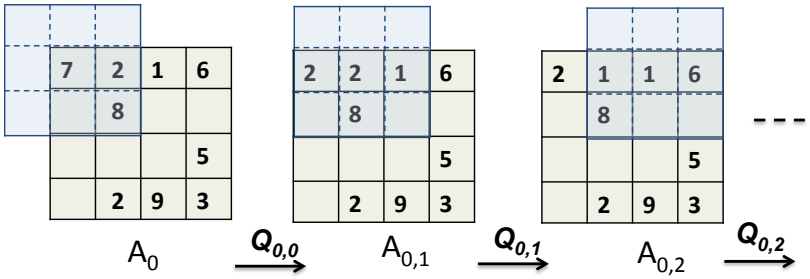
$$A_0 \xrightarrow{Q_1} A_1 \xrightarrow{Q_2} \ldots \overbrace{A_{i-1} \xrightarrow{Q_i} A_i \xrightarrow{Q_{i+1}} A_{i+1}}^{A_i \xrightarrow{q_{i,1} \cdot q_{i,2} \cdots q_{i,j-1} \cdot q_{i,j}} A_{i+1}} \tag{4.2}$$

Figure 4.2 shows an iterative array computation with only major steps involved, while Figure 4.3 presents the same application but executed with minor steps.

We further observe from the example applications in Section 4.2 that the functions Q often follow a similar pattern.

First, the value of each cell in iterative array A_{i+1} that is updated by Q only depends on values in *nearby cells* in array A_i. We capture this spatial constraint with a function π that specifies the mapping from output cells back onto input cells:

Figure 4.3 Iterative array A and its state after three minor steps, each of the form: $Q_{i,j} = Q_{c_{i,j}}^{f^\pi,\delta^\pi}$ where $c_{i,j}$ is the cell at $A[i][j]$, f^π applies $min()$ aggregate, δ simply stores the aggregate result as the new value in cell $c_{i,j}$, and $\pi : (x, y) \rightarrow [x \pm 1][y \pm 1]$



Definition 4.3.4. *π is an assignment function defined as $\pi : cells(A) \rightarrow \mathcal{P}(cells(A))$, where $cells(A)$ is the set of all the cells in array A and $\mathcal{P}()$ is the powerset function.*

Figure 4.4 illustrates two examples of assignment functions.

Definition 4.3.5. *f^π is an aggregate function defined as $f^\pi : cells(A) \rightarrow \tau$. f^π groups the cells of the array A according to assignment function π, with one group of cells per cell in the array A. It then computes the aggregate functions separately for each group. The aggregate result is stored in tuple τ.*

Finally, Q updates the output array with the computed aggregate values:

Definition 4.3.6. *$\delta^\pi : (cells(A), f^\pi) \rightarrow cells(A)$ is a cell-update function. It updates each cell of the array A with the corresponding tuple τ computed by f^π and the current value of the cell itself.*

These three pieces together define the iterative array computation $Q_C^{f^\pi,\delta^\pi}$ as follows:

Definition 4.3.7. *Iterative array computation $Q_C^{f^\pi,\delta^\pi}$ on the subset of cells C where $C \in \mathcal{P}(cells(A))$ generates subset of cells $C' \in \mathcal{P}(cells(A))$ such that $\forall c \in C$ and $c' \in C'$ $c' = \delta^\pi(c, f^\pi(c))$ where c and c' are two corresponding cells in those subsets.*

An example iterative array transformation is presented in Figure 4.2.

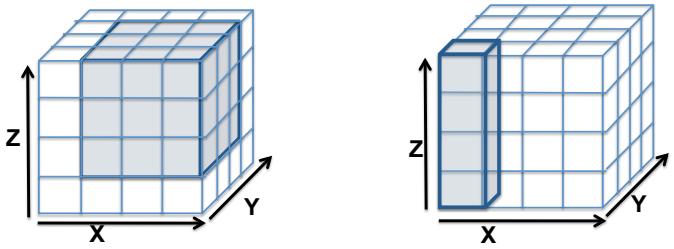
The following example from the `SourceDetect` application helps us to clarify the intuition behind the definition of *iterative array computation*.

Example 4.3.1. Consider a simplified version of the `SourceDetect` application described in Section 4.2. The goal is to detect all the clusters c in the array A where each cell $p_1 = (x_1, y_1)$ in cluster c has at least one neighbor $p_2 = (x_2, y_2)$ $|x_1 - x_2| \leq 1$ $|y_1 - y_2| \leq 1$ in the same cluster, if it is not a single-cell cluster. One can implement SourceDetect with a simple iterative computation. Consider π to be the 3X3 window around a cell. We slide the window through the array cells in major order. At each *minor* step, at each cell $c_{i,j}$ at the center of the window, we apply an iterative array computation $Q_{i,j} = Q_{c_{i,j}}^{f^\pi, \delta^\pi}$ where f^π applies $min()$ aggregate of the 3x3 window, π, and δ^π is a cell-update function that simply stores the result of the $min()$ aggregate into the cell $c_{i,j}$. Figure 4.3 illustrates three steps of this computation. Notice that the output of iterative array computation $Q_{0,0}$ becomes the input for $Q_{0,1}$ and so on. Another strategy is to have many windows grouped and applied together. In other words, instead of applying iterative array computation per cell, we apply $Q_C^{f^\pi, \delta^\pi}$ on a group of cells $C \in \mathcal{P}\left(cells(A)\right)$ at one *major* step. Note that when using minor steps, the output of each minor step serves as input to the next step. In contrast, when using major steps, the iterative array computations see the original array state at the beginning of that iteration. Figure 4.2 shows the iterative array computation of the latter strategy. The former strategy has less expensive steps than the latter strategy, but it requires more steps to converge. □

π assignment function In many iterative array algorithms, the value of a cell c_i in the next iteration is influenced either by *nearby* cells or by cells that share the same cell-value with c_i for some attributes. Based on this observation, our system allows the user to express the $\mathcal{P}\left(cells(A)\right)$ in Definition 4.3.4 either as a *window* or as an attribute. An example window assignment in a 3D array is $\pi : (x, y, z) \rightarrow [x \pm k_1][y \pm k_2][z \pm k_3]$, where (x, y, z) is a cell coordinate and the window w is defined as follow:

$$
\begin{aligned}
&w : [x \pm k_1][y \pm k_2][y \pm k_3] = \{(x', y', z') \in cells(A) : \\
&x - k_1 \leq x' \leq x + k_1 \ \& \\
&y - k_2 \leq y' \leq y + k_2 \ \& \\
&z - k_3 \leq z' \leq z + k_3\}
\end{aligned}
$$

Figure 4.4 Two examples of window assignment functions: (a) $\pi_1 : (x, y, z) \rightarrow [x \pm 1][y \pm 1][z \pm 1]$, the associated window is highlighted for the cell at $(2, 1, 2)$. (b) $\pi_2 : (x, y, z) \rightarrow [x][y]$, the associated window is highlighted for all the cells at (x, y, z) with $z = 0$.



(a) An example window for `SourceDetect` application

(b) An example window for `SigmaClip` application

where each k_i is a constant. Two examples of assignment functions are illustrated in Figure 4.4. Note that in the Example 4.4(a), assignment function π_1 is a one-to-one function and in 4.4(b), π_2 is not a one-to-one mapping. An example *attribute* assignment function is the K-means clustering application described in Example 4.2.3: $\pi : (x, y) \rightarrow label$ where all the cells with the same label are grouped together.

ArrayLoop asks the user to encapsulate all the elements of the model in a *FixPoint* operator:

$$FixPoint(A, \pi, f, \delta, T, \epsilon) \tag{4.3}$$

With our model so far, the user specifies the logic of the iterative algorithm without worrying about the way it is going to be executed. Similar to relational databases that support logical and physical independence, our model can be implemented and executed on top of various array execution engines independent of their execution strategy (refer to Example 4.3.1). In the rest of the paper, we describe how the queries specified in our model are rewritten and efficiently run in the SciDB array engine. The execution strategy in SciDB only uses major steps. Mini steps implementation, i.e. asynchronous execution, is left for future study.

To execute an iterative array computation in SciDB, a naïve approach (shown in Algorithm 7) is to simply iteratively invoke array queries from a high-level scripting language such as SciDB-Py [90], a python interface for SciDB. This approach, however, prevents or at least complicates the

Algorithm 7 Iterative algorithm, naïve approach

```
1: while A_i ≠ A_{i+1} do
2:     run AQL Q_1
3:     ...
4:     run AQL Q_k
5: end while
```

automated optimizations of iterative computations. Instead, our approach is to extend SciDB-Py with a python $FixPoint()$ operator and an optimizer module that we name ArrayLoop. The user encapsulates its iterative algorithm in our $FixPoint()$ python operator and then the ArrayLoop optimizer triggers a set of query re-writing tasks in order to leverage a series of optimizations that we develop: *incremental iterative processing*, *overlap iterative processing*, and *multi-resolution iterative processing*. ArrayLoop acts as a pre-processing module before executing the iterative query in SciDB. Currently the majority of the ArrayLoop implementation is outside the core SciDB engine. As future work, we are planning to push the ArrayLoop python prototype into the core SciDB engine. In the rest of this section we describe each of the three optimizations in more detail.

4.4 Incremental Iterations

In a wide range of iterative algorithms, the output at each iteration differs only partly from the output at the previous iteration. Performance can thus significantly improve if the system computes, at each iteration, only the part of the output that changes rather than re-computing the entire result every time. This optimization called *incremental iterative processing* [30] is well-known, *e.g.* in semi-naive datalog evaluation, and has been shown to significantly improve performance in relational and graph systems. ArrayLoop tries to build in support for incremental processing when the application supports it. The `SigmaClip` application described in Section 4.2.1 is an example application that can benefit from incremental iterative processing. Figure 4.6 shows multiple snapshots of running the sigma-clipping algorithm with incremental iterative processing on a subset of the `lsst` dataset. Green-colored points are the ones with changed values across two consecutive iterations. As the iterative computation proceeds, the number of green-colored points drops dramatically and consequently the amount of required computation at that step.

`sigma-clipping()` and `incr-sigma-clipping()` modules in Algorithm 8 show the

Algorithm 8 `SigmaClip` application followed by image co-addition

1. **function SIGMA-CLIPPING**(A,k) ▷ Naïve sigma-clip
2. Input: Iterative Array A <float d>[x,y,t]
3. Input: k a constant parameter.
4. **while** (some pixels $A[x, y, t]$ are filtered) **do**
5. $T[x, y]$ = select $avg(d)$ as μ, $stdv(d)$ as σ from A group by x, y
6. $S[x, y, t]$ = select * from T join A on $T.x$ = $A.x$ *and* $T.y$=$A.y$
7. $A[x, y, t]$ = select d from S where $\mu - k \times \sigma \leq d \leq \mu + k \times \sigma$
8. **end while**
9. **end function**

10. **function INCR-SIGMA-CLIPPING**(A,k) ▷ Incremental sigma-clip
11. Input: Array A <float d>[x,y,t].
12. Input: k: a constant parameter.
13. Local: Array C <int c,float s,float s^2>[x,y].
14. Local: $Collect \leftarrow \phi$ ▷ Collects all the filtered points.
15. Local: $Remain \leftarrow A$ ▷ Keeps track of remaining points.
16. $\Delta A \leftarrow A$
17. **while** (ΔA is not empty) **do**
18. $T_1[x, y] \leftarrow$ select count(d) as c, sum(d) as s, sum(d^2) as s^2 from ΔA group by x, y
19. **if** (first iteration) **then**
20. $C \leftarrow T_1[x, y]$
21. **else**
22. ΔC[x,y] $\leftarrow$ select $C.c - T_1.c$ as c , $C.s - T_1.s$ as s , $C.s^2 - T_1.s^2$ as s^2 from C join T_1 on $T_1.x$ = $C.x$ & $T_1.y$=$C.y$
23. **end if**
24. $T[x, y] \leftarrow$ select $\frac{C.s}{C.c}$ AS μ, $\sqrt[2]{\frac{C.s^2}{C.c} - (\frac{C.s}{C.c})^2}$ AS σ from ΔC
25. $S[x, y, t] \leftarrow$ select $A.d$, $T.\mu$, $T.\sigma$ from T join $Remain$ on $T.x$ = $A.x$ *and* $T.y$=$A.y$
26. $\Delta A \leftarrow$ select d from S where $d \leq \mu - k \times \sigma$ *or* $d \geq \mu + k \times \sigma$
27. $Remain \leftarrow \pi_d(S)$-$\Delta A$ ▷ Updates Remain.
28. $Collect \leftarrow \Delta A$ ▷ Adds the filtered points to Collect.
29. **end while**
30. $A \leftarrow A$-$Collect$ ▷ Produces the final state for A.
31. **end function**

co-addition phase:
32. $R[x, y] \leftarrow$ select sum($A.d$) as $coadd$ from A group by x, y

original implementation and the manually-written incremental version of the implementation, respectively. In the `sigma-clipping()` module, the `avg()` and `stdv()` aggregate operators are computed over the whole input at each iteration, which is inefficient. In `incr-sigma-clipping()`, the user rewrites the `avg()` and `stdv()` aggregate operators in terms of two other aggregate operators `count()` and `sum()` (Algorithm 8, Lines 18 and 24). The user also needs to carefully merge the current partial aggregates with the aggregate result of the previous iteration (Algorithm 8, Line 22). As shown in Algorithm 8, writing an efficient incremental implementation is not a trivial task. It is painful for users if they need to rewrite their algorithms to compute these increments

Algorithm 9 ArrayLoop version of the `SigmaClip` application followed by image co-addition

1. **function** ARRAYLOOP-SIGMA-CLIPPING(A,k) ▷ `SigmaClip` algorithm with FixPoint operator provided by the user.
2. Input: Iterative Array A <float d>[x,y,t],
3. Input: k a constant parameter.
4. $\pi : [x][y][z] \rightarrow [x][y]$.
5. δ : "$A.d \geq \mu - k \times \sigma$ and $A.d \leq \mu + k \times \sigma : A?\phi$"
6. $f : \{avg() \ as \ \mu, stdv() \ as \ \sigma\}$
7. $FixPoint(A, \pi, f, \delta, count(), 0)$
8. **end function**

9. **function** ARRAYLOOP-INCR-SIGMA-CLIPPING(A,k) ▷ ArrayLoop incremental rewriting of the `SigmaClip`.
10. Input: Iterative Array A <float d>[x,y,t],
11. Input: k: a constant parameter.
12. Local: Iterative Array C <int c,float s,float s^2>[x,y],
13. $\Delta A^- \leftarrow A$
14. **while** (ΔA^- is not empty) **do**
15. $T_1[x, y] \leftarrow$ select count(d) as c, sum(d) as s, sum(d^2) as s^2 from ΔA^- group by x, y
16. **if** (first iteration) **then**
17. $C \leftarrow T_1[x, y]$
18. **else**
19. merge(C,T_1,$C.c - T_1.c$)
20. merge(C,T_1,$C.s - T_1.s$)
21. merge(C,T_1,$C.s^2 - T_1.s^2$)
22. **end if**
23. $T[x, y] \leftarrow$ select $\frac{T.s}{T.c}$ AS μ, $\sqrt[2]{\frac{T.s^2}{T.c} - (\frac{T.s}{T.c})^2}$ AS σ FROM $\Delta^+ C$
24. merge($A, T, T.\mu - k \times T.\sigma \leq A.d \leq T.\mu + k \times T.\sigma : A?\phi$)
25. **end while**
26. **end function**

co-addition phase:

27. $R[x, y] \leftarrow$ select sum($A.d$) as $coadd$ from A group by x, y

and manage them during the computation. Ideally, the user wants to define the semantics of the algorithm and the system should automatically generate an optimized, incremental implementation. Additionally, as we show in the evaluation, if the system is aware of the incremental processing, it can further optimize the implementation by pushing certain optimizations all the way to the storage layer.

4.4.1 Query Rewrite for Incremental Processing

In ArrayLoop, we show how the incremental processing optimization can be applied to arrays. As it is shown in Algorithm 9, with ArrayLoop, the user provides a `FixPoint` operator in `ArrayLoop-sigma-clipping` function. ArrayLoop automatically expands and rewrites the operation into an incremental implementation if the application supports it, as shown in the `ArrayLoop-incr-sigma-clipping` function. Currently the decision to use incremental itera-

tive processing is left to the user. Given the `FixPoint` operator, ArrayLoop performs two tasks: (1) it automatically rewrites aggregate functions, if possible, into incremental ones and (2) it efficiently computes the last state of the iterative array using the updated cells at each iteration. The automatic rewrite is enabled by the precise model for iterative computations in the form of the three functions π, f, and δ. Given this precise specification of the loop body, ArrayLoop rewrites the computation using a set of rules that specify how to replace aggregates with their incremental counter-parts when possible. To efficiently compute incremental state updates, we introduce a special *merge* operator. We now describe both components of the approach.

(1) Automatic aggregate rewrite: ArrayLoop triggers the *incremental iterative processing* optimization if any aggregate function in the `FixPoint` operator is flagged as incremental. The Data cube paper [34] defines an aggregate function $F()$ as algebraic if there is an M-tuple valued function $G()$ and a function $H()$ such that: $F(\{X_{i,j}\}) = H(\{G(\{X_{i,j}\}|i = 1, \ldots, I\})|j = 1, \ldots, J\})$. ArrayLoop stores a triple $(agg, \{G_1, \ldots, G_k\}, H)$ for any $algebraic$ function in the system and rewrites the aggregate query in terms of $G()$ and $H()$ functions during the query rewriting phase. For example, ArrayLoop records the triple `(avg(),{sum(),count()},sum/count)` and rewrites the *algebraic* average function `avg()` using the combination of `sum()` and `count()` to leverage incremental iterative processing.

(2) Incremental state management: ArrayLoop provides an efficient method for managing array state and incremental array updates during the course of an iterative computation. We observe that, during incremental processing, a common operation is to *merge* the data in two arrays, which do not necessarily have the same number of dimensions. In our example application, merging happens when the partial aggregates are combined with the aggregate result of the previous iteration, line 22 in `incr-sigma-clipping()` function. This operation merges together two 2D arrays where the merge logic is inferred from the incremental aggregate function f. Such merging also happens when the results of the aggregate function are used to update the iterative array, lines 25 and 26 in `incr-sigma-clipping()` function. In this case, the application merges the data in a 2D array with the data in a 3D array by *sliding* or *extruding* the 2D array through the 3D array. The δ cell-update function defines the logic of the merge operation in this case. The π assignment function pairs-up cells from the intermediate aggregation array and the iterative array that merge together.

In the manual implementation, shown in the `incr-sigma-clipping()` function, the user implements the merge logic manually using join and filter queries, which is inefficient. [1] To remove this inefficiency, given the `FixPoint` operator, ArrayLoop automatically generates queries with explicit merge points that leverage a new merge operator that we add to SciDB: `merge(Array source, Array extrusion, Expression exp)`.

The new *merge* operator is unique in a sense that it not only specifies the merge logic between two cells via a mathematical expression, `exp`, but it also automatically figures out which pairs of cells from the two arrays merge together by examining their common dimensions. ArrayLoop merges two cells from the *source* array and *extrusion* array if they share the same dimension-values in those dimensions that match in dimension-name. One cell in the `extrusion` array can thus merge with many cells in the `source` array. Figure 4.7 illustrates the merge operator for queries in lines 25 and 26 in Algorithm 8. As the figure shows, the `extrusion` array slides through the `source` array as the merging proceeds.

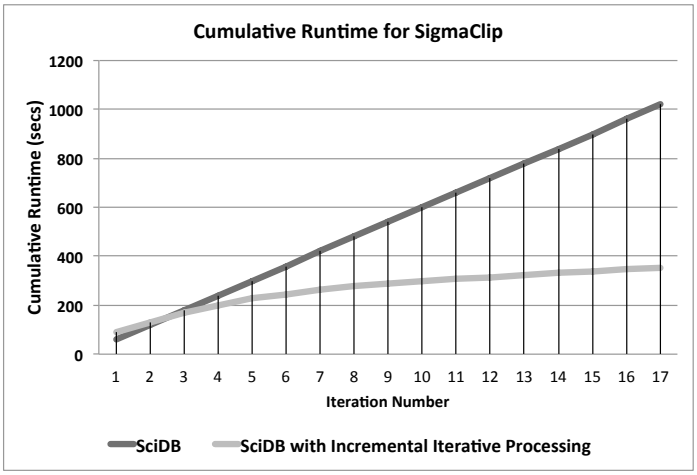
4.4.2 Pushing Incremental Computation into the Storage Manager

Our key observation is that increments between iterations translate into updates to array cells and can thus be captured with two auxiliary arrays: a *positive delta array* and a *negative delta array*. At each iteration, the positive delta array ΔA^+ records the *new* values of updated cells and the negative delta array ΔA^- keeps track of the *old* values of updated cells. Delta arrays can automatically be computed by the system directly at the storage manager level.

As a further optimization, we extend the SciDB storage manager to manage simple merge operations such as addition/subtraction found in Lines 19, 20, and 21 of Algorithm 9. In fact, queries in the `incr-sigma-clipping()` function at Lines 27, 28, and 30 (queries with red box frames) can all be pushed into the storage manager. ArrayLoop uses naming conventions as a hint to the storage manager about the semantics of the merge operation. For example $A_{(-)} \leftarrow B$, asks the storage manager to subtract array B from array A and store the result of the $(A - B)$ operation as the new version of array A. In case array A is iterative, the new values and the old values of updated

[1] From an engineering point of view, the new *merge* operator, unlike a join, can also leverage vectorization where instead of merging one pair of matching cells at a time, ArrayLoop merges group of matching cells together, potentially improving query runtime, especially when the number of dimensions in the two input arrays is different.

Figure 4.5 Cumulative runtime of the `SigmaClip` application with constant $k = 2$ on a subset of LSST images with and without incremental iterative processing optimization on the first 17 iterations.



cells are stored in Δ^+A and Δ^-A, respectively.

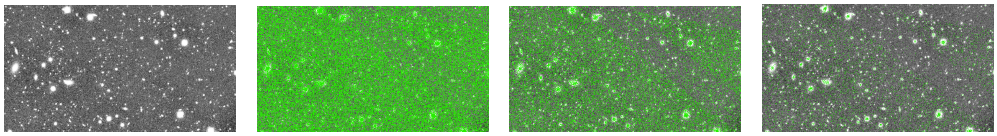
To achieve high performance, the storage manager keeps chunks of the result array A together on disk with the corresponding chunks from the auxiliary ΔA^+ and ΔA^- arrays. Furthermore, we extend the `Scan()` and `Store()` operators to read and write partial arrays ΔA^+ and ΔA^-, respectively. With those optimizations, the user does not need to explicitly write a user-defined `diff()` function or, as shown in the `incr-sigma-clipping()` example, a sequence of `join()` and `filter()` queries in order to extract delta arrays from the output of the last iteration.

A query runtime comparison of the `SigmaClip` application with all the incremental iterative processing optimizations and no optimization is illustrated in Figure 4.5. Unlike the naïve version whose cumulate runtime increases linearly as the iteration proceeds, the runtime increase drops significantly in the incremental version and the cumulative runtime of the graph nearly flatten at iteration 17.

4.5 Iterative Overlap Processing

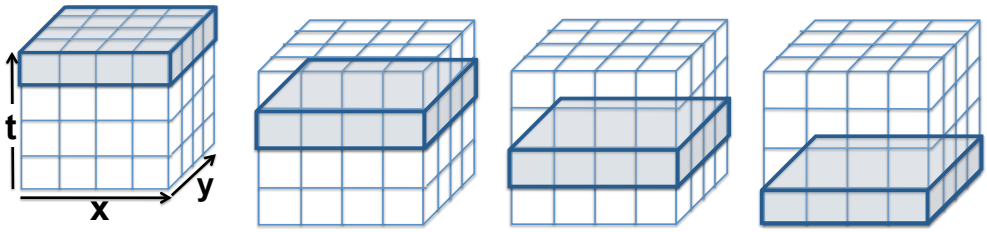
To process a query over a large-scale array in parallel, SciDB (and other engines) break arrays into sub-arrays called chunks, distribute chunks to different compute nodes (each node receives multiple chunks), and process chunks in parallel at these nodes. For many operations, such as filter for example, one can process chunks independently of each other and can union the result. This

Figure 4.6 Snapshots from the first 3 iterations of the `SigmaClip` application with incremental optimization on the LSST dataset. Green-colored points are the ones that change across iterations. As the iterative computation proceeds, the number of green-colored points drops dramatically.



(a) Original Image (b) Iteration-1 (c) Iteration-2 (d) Iteration-3

Figure 4.7 merge$(A, T, T.\mu - k \times T.\sigma \leq A.d \leq T.\mu + k \times T.\sigma : A?\phi)$ in `SigmaClip` application. This is the core filtering step where the outliers are removed. The 3D source array A <float d>[x,y,t] and the 2D extrusion array (highlighted) T <float μ,float σ>[x,y] share the first two dimensions. (a), (b), (c), and (d) show how the cells in the extrusion array slide into the source array at runtime.



(a) step 1 (b) step 2 (c) step 3 (d) step 4

simple strategy, however, does not work for many scientific array operations. Frequently, the value of each output array cell is based on a neighborhood of input array cells. Data clustering is one example. Clusters can be arbitrarily large and can go across array chunk boundaries. A common approach to computing such operations in parallel is to perform them in two steps: a local, parallel step followed by an aggregate-type post-processing step [48, 49, 56] that merges partial results into a final output. For the clustering example, the first step finds clusters in each chunk. The second step combines clusters that cross chunk boundaries [48]. Such a post-processing phase, however, can add significant overhead. To avoid a post-processing phase, some have suggested to extract, for each array chunk, an overlap area ϵ from neighboring chunks, store the overlap together with the original chunk [87, 91], and provide both the core data and overlap data to the operator during processing [99]. This technique is called *overlap processing*. An example of the overlapped chunks are depicted in Figure 4.8.

4.5.1 Efficient Overlap Processing

Overlap processing can be especially helpful for iterative computations that need to perform an operation multiple times until a termination condition. A technique similar to *overlap processing* has been used in iterative graph-processing systems. Distributed GraphLab [53] proposed *ghost nodes* that are replicas of vertices across graph partitions. The idea is to provide each vertex in the graph with direct memory access to all its neighboring vertices. The challenge is then to efficiently keep the cached vertices up to date. Similarly, ArrayLoop leverages overlapping techniques to support iterative parallel array processing, where cells at boundary of chunk partitions are replicated and must be kept up to date.

Array applications that can benefit from overlap processing techniques are those that update the value of certain array cells by using the values of neighboring array cells. The `SourceDetect` application described in Section 4.2.2 is an example application that can benefit form overlap processing. Other example applications include "oceanography particle tracking", which follow a set of particles as they move in a 2D or 3D grid. A velocity vector is associated with each cell in the grid and the goal is to find a set of trajectories, one for each particle in the array. Particles cannot move more than a certain maximum distance (depending on the maximum velocity of particles) at each step. These applications can be effectively processed in parallel by leveraging overlap processing techniques.

The challenge, however, is to keep replicated overlap cells up-to-date as their values change across iterations. To efficiently update overlap array cells, we leverage SciDB's bulk data shuffling operators as follows: SciDB's operator framework implements a `bool requiresRepart()` function that helps the optimizer to decide whether the input array requires repartitioning before the operator actually executes. The partitioning strategy is determined by the operator semantics. For example, `WindowAggregate` operator [89] in SciDB requires repartitioning with overlap in case the input array is not already partitioned in that manner. We extend the SciDB operator interface such that ArrayLoop can dynamically set the returned value of the operator's `requiresRepart()` function. To update overlap data, ArrayLoop sets the `requiresRepart()` return value to true. ArrayLoop has the flexiblity to set the value to true either at each iteration or every few iterations. In case an operator in SciDB is guided by ArrayLoop to request repartitioning, the SciDB optimizer

injects the `Scatter/Gather` [89] operators to shuffle the data in the input iterative array before the operator executes. With this approach, ArrayLoop leverages the rich set of array operators available in SciDB to keep the overlap data up to date. One benefit of this approach is that array operators in SciDB runs at the chunk level. They receive data chunks as inputs and produce a data chunk as output. Therefore, the `Scatter/Gather` operation re-shuffles overlap data one chunk at a time. Chunk-based data shuffling is faster compared with the method that shuffles overlap data one cell at a time. The downside of using SciDB's `Scatter/Gather` general operators is the relative higher cost of data shuffling when only few overlap cells have changed.

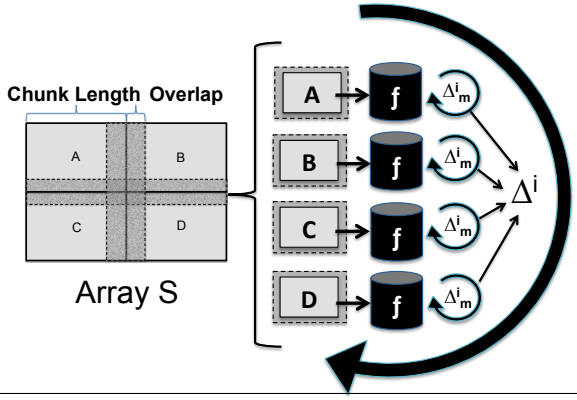
4.5.2 Mini-Iteration Processing

Keeping overlapping cells updated at each iteration requires reading data from disk, shuffling it across the network, and writing it to disk. These are all expensive operations. Any reduction in the number of such data synchronization steps can yield significant performance improvements.

We observe that a large subset of iterative applications have the property that overlap cells can be updated only every few iterations. These are applications, for example, that try to find structures in the array data, *e.g.* `SourceDetect` application. These applications can find structures locally and eventually need to exchange information to stitch these local structures into larger ones. For those applications, we do the following additional optimization: We run the algorithm for multiple iterations without updating the replicas of overlap cells. The application iterates over chunks locally and independently of other chunks. Every few iterations, we update overlap cells, and continue with another set of local iterations. The key idea behind this approach is to avoid data movement across array chunks unless we are sure there are enough changes to justify the cost.

We call each series of local iterations without overlap cell synchronization a *mini iteration*. Figure 4.8 illustrates the schematic of the mini iteration optimization. A similar idea has already been exercised in other data management systems with iteration support. For example, in distributed GraphLab [53], a vertex may choose to schedule its neighbors only when it has made a substantial change to its local data. We borrow this optimization and apply it to arrays to measure the efficiency of that optimization in this new setting. Unlike GraphLab, ArrayLoop makes the overlap data shuffling a global decision for the entire array not the individual cells, which leverages SciDB's

Figure 4.8 The schematic picture for mini iteration optimization. Δ_m^i represents local changes at iteration i at mini-iteration m and Δ^i is the global changes at iteration i.



simpler scheduler and also amortizes the cost by synchronizing all overlap cells in the same operation.

An alternative approach is for the scheduler to delegate the decision to shuffle overlap data to individual chunks, rather than making the decision array-global as we do in this paper or cell-local as in GraphLab. We leave this extra optimization for future work.

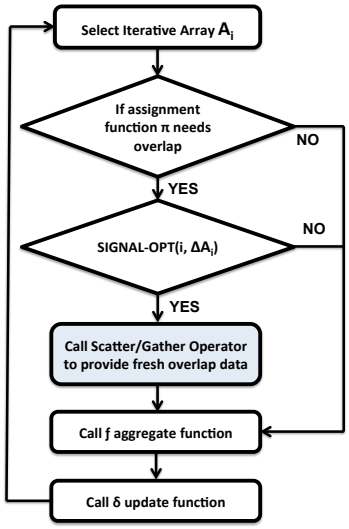
ArrayLoop includes a system-configurable function `SIGNAL-OPT()` that takes as input an iteration number and a delta iterative array which represents the changes in the last iteration. This function is called at the beginning of each iteration. The output of this function defines if the overlap data at the current iteration needs to be shuffled. A control flow diagram of this procedure is described in Figure 4.9. There exists an optimization opportunity to exploit: Do we exchange overlap cells every iteration? or do we wait until local convergence? or something in between these two extremes? We further examine those optimization questions in Section 4.7.

4.6 Multi-Resolution Optimization

In many scientific applications, raw data lives in a continuous space (3D universe, 2D ocean, N-D space of continuous variables). Scientists often perform continuous measurements over the raw data and then store a discretized approximation of the real data in arrays. In these scenarios, different levels of granularity for arrays are possible and scientifically meaningful to analyze. In fact, it is common for scientists to look at the data at different levels of detail.

As discussed earlier, many algorithms search for structure in array data. One example is the

Figure 4.9 Control flow diagram for mini-iteration-based processing in ArrayLoop.



extraction of celestial objects from telescope images, snow cover regions from satellite images, or clusters from an N-D dataset. In these algorithms, it is often efficient to first identify the outlines of the structures on a low-resolution array, and then refine the details on high-resolution arrays. We call this array-specific optimization *multi-resolution* optimization. This multi-resolution optimization is a form of prioritized processing. By first processing a low-resolution approximation of the data, we focus on identifying and approximating the overall shape of the structures. Further processing of higher-resolution arrays helps extract the more detailed outlines of these structures.

In the rest of this section we describe how ArrayLoop automates this optimization in SciDB. We use the `KMeans` application described in Section 4.2.3 and the `SourceDetect` application described in Section 4.2.2 as our illustrative examples.

To initiate the *multi-resolution* optimization, ArrayLoop initially generates a series of versions, $A^i, A^{i+1}, \ldots, A^j$, of the original iterative array A. Each version has a different resolution. A^i is the original array. It has the highest resolution. A^j is the lowest-resolution array. Figure 4.10 illustrates three pixelated versions of an `lsst` image represented as iterative array A^0 in the context of the `SourceDetect` application. The coarser-grained, pixelated versions are generated by applying a sequence of `grid` followed by `filter` operations represented together as $grid_p$`()`, where p is the predicate of the `filter` operator. The size and the aggregate function in the `grid` operator are

application-specific and are specified by the user. The `SourceDetect` application has a grid-size of (2×2) and an aggregate function `count` with a filter predicate that only passes grid blocks without empty cells (in this scenario all the grid blocks with count=4). This ensures that cells that are identified to be in the same cluster in a coarsened version of the array, remain together in finer grained versions of the array as well. In other words, the output of the iterative algorithm on the pixelated version array A^j should be a *valid* intermediate step for A^{j-1}. ArrayLoop runs the iterative function Q on the sequence of pixelated arrays in order of increasing resolution. The output of the iterative algorithm after convergence at pixelated version A^i is transformed into a finer-resolution version using an `xgrid` operator (inverse of a grid operator). It is then merged with A^{i-1}, the next immediate finer-grained version of the iterative array. We represent both operations as $xgrid_m$`()`. The `xgrid` operator [89] produces a result array by scaling up its input array. Within each dimension, the `xgrid` operator duplicates each cell a specified number of times before moving to the next cell. The following equations illustrate the ordered list of operators called by ArrayLoop during *multi-resolution* optimizations:

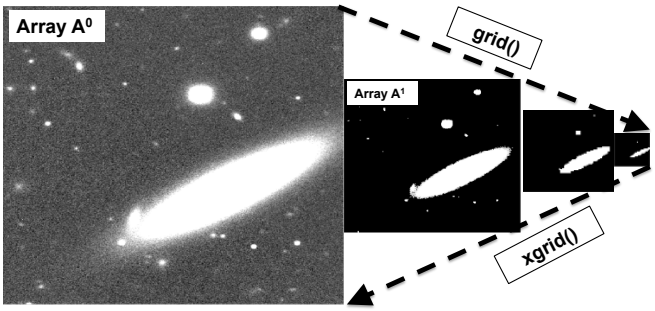
$$
\begin{aligned}
& A^0 \xrightarrow{grid_p()} \ldots A^i \xrightarrow{grid_p()} A^{i+1} \xrightarrow{grid_p()} \ldots A^j \\
& A^j \xrightarrow{Q} A^{*j} \xrightarrow{xgrid_m(A^{*j})} A_x^{j-1} \\
& \ldots \\
& A_x^1 \xrightarrow{Q} A^{*1} \xrightarrow{xgrid_m(A^{*1})} A_x^0 \\
& A_x^0 \xrightarrow{Q} A^{*0}
\end{aligned}
\tag{4.4}
$$

where A^{*i} is the output of the iterative algorithm Q on pixelated array A^i, and A^{j-1} is replaced with A_x^{j-1} as the new input for the iterative computation at pixelated version $(j-1)$.

By carefully merging the approximate results with the input array at the next finer-grained level, ArrayLoop skips a significant amount of computation.

The K-means clustering algorithm on points in a continuous space is another example application that benefits from this optimization. The `KMeans` application can use an arbitrary grid size. It also uses `count` as the aggregate function with a filter predicate that passes grid blocks that have at

Figure 4.10 Illustration of the multi-resolution optimization for the `SourceDetect` application. There is a sequence of three grid operations initiated from the original `lsst` image A^0: $A^0 \xrightarrow{grid_p(A^0,2,2)} A^1 \xrightarrow{grid_p(A^1,2,2)} A^2 \xrightarrow{grid_p(A^2,2,2)} A^3$. The more pixelated versions only retain the main structure of the image.



least one non-empty cell. It is easy to observe that in case of K-means clustering, A_x^{j-1} is a *valid* labeling for the next pixelated array A^{j-1}. Basically, K-means clustering on A^j produces a better set of centroids for the k-means algorithm on A^{j-1} than a random set of centroids.

The advantage of applying the *multi-resolution* optimization goes beyond better query runtime performance. This optimization can also help when the original iterative array changes, which is described as the following additional optimization:

Input Change Optimization: If ArrayLoop materializes the outputs A^{*i} for all the pixelated versions of the original array A, then there is an interesting optimization in case the original iterative array A is modified. Unlike the Naiad system [59] that materializes the entire state at each iteration to skip some computation in case of change in the input data, ArrayLoop takes a different strategy. When changes in the input occur, ArrayLoop re-generates the pixelated arrays A^is in Equation 4.4, but only runs the iterative algorithm Q for those arrays A^is that have also changed in response to the input array change. If A^i did not change for some i, ArrayLoop skips the computation $A^k \xrightarrow{Q} A^{*k}\ \forall k \geq i$ and uses the materialized result A^{*i} from the previous run to produce A_x^{i-1}. The intuition is that, if there are only a few changes in the input array, it is likely that changes are not carried over to all the pixelated versions of the array and our system reuses some results of the previous run for the current computation as well.

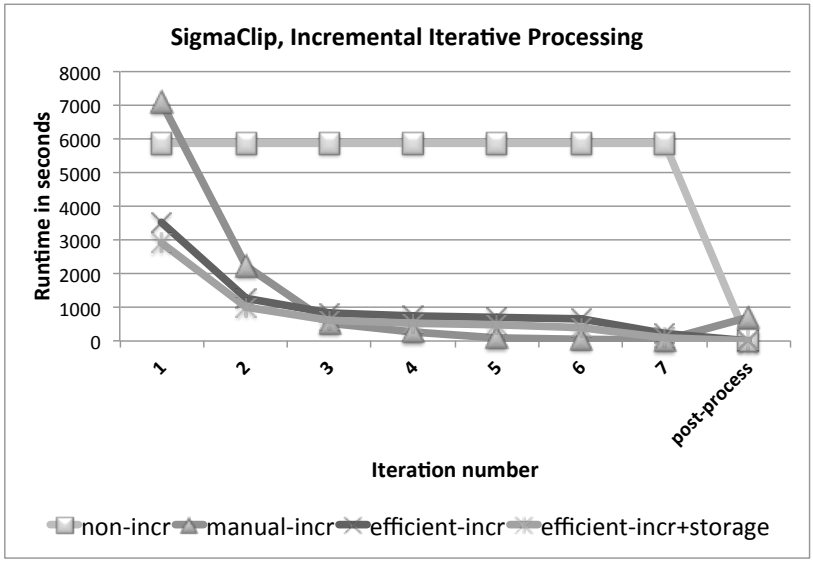
4.7 Evaluation

In this section, we demonstrate the effectiveness of ArrayLoop's native iterative processing capabilities including the three optimizations on experiments with 1TB of `LSST` images [83]. We describe this dataset in detail in Chapter 7. Because the LSST will only start to produce data in 2019, astronomers are testing their analysis pipelines with synthetic images that simulate what the survey will produce. We use one such synthetic dataset. The images take the form of one large 3D array (2D images accumulated over time) with almost 44 billion non-empty cells. The experiments are executed on a 20-machine cluster. (Intel(R) Xeon(R) CPU E5-2430L @ 2.00GHz) with 64GB of memory and Ubuntu 13.04 as the operating system. We report performance for two real-scientific applications `SigmaClip` and `SourceDetect` described in Sections 4.2.1 and 4.2.2, respectively. `SigmaClip` runs on the large 3D array and `SourceDetect` runs on the co-added 2D version of the whole dataset.

4.7.1 Performance for Incremental Iterative Processing

We first demonstrate the effectiveness of our approach to bringing incremental processing to the iterative array model in the context of the `SigmaClip` application. Figure 4.11(a) shows the total runtime of the algorithm with different execution strategies. As shown, the `non-incremental` "sigma-clipping" algorithm performs almost four times worse than any other approach. The `manual-incr` approach is the `incr-sigma-clipping` function from Section 4.4, which is the manually-written incremental version of the "sigma-clipping" algorithm. This approach keeps track of all the points that are still candidates to be removed at the next iteration and discards the rest. By doing so, it touches the minimum number of cells from the input dataset at each iteration. Although `manual-incr` performs better than other approaches at later stages of the iterative computation, it incurs significant overhead during the first few iterations due to the extra data points tracking (Lines 25 to 28 in `incr-sigma-clipping()` function). `manual-incr` also requires a `post-processing` phase at the end of the iterative computation to return the final result. `efficient-incr` and `efficient-incr+storage` are the two strategies used by ArrayLoop. `efficient-incr` represents ArrayLoop's query rewrite for incremental state management that also leverages our `merge` operator. `efficient-incr+storage` further includes the storage manager extensions.

Figure 4.11 `SigmaClip` application: incremental strategy *v.s.* non-incremental. Constant $k = 3$ in all the algorithms.



(a) `SigmaClip` application with different strategies. `manual-incr` refers to the `incr-sigma-clipping` function in Section 4.4. `efficient-incr` and `efficient-incr+storage` refer to ArrayLoop versions of the `SigmaClip` computation with and without additional storage optimizations, respectively.

non-incr	manual-incr	efficient-incr	efficient-incr+storage
40957	10975	8007	6096

(b) Total runtime for `SigmaClip` for different strategies in seconds.

Figure 4.11(b) shows the total runtime in each case. ArrayLoop efficient versions of the algorithm are competitive with the manually written variant. They even outperform the manual version in this application. All the incremental approaches beat the non-incremental one by a factor of $4 - 6X$. Interestingly, our approach to push some incremental computations to the storage manager improves `efficient-incr` by an extra 25%.

4.7.2 Performance of Overlap Iterative Processing

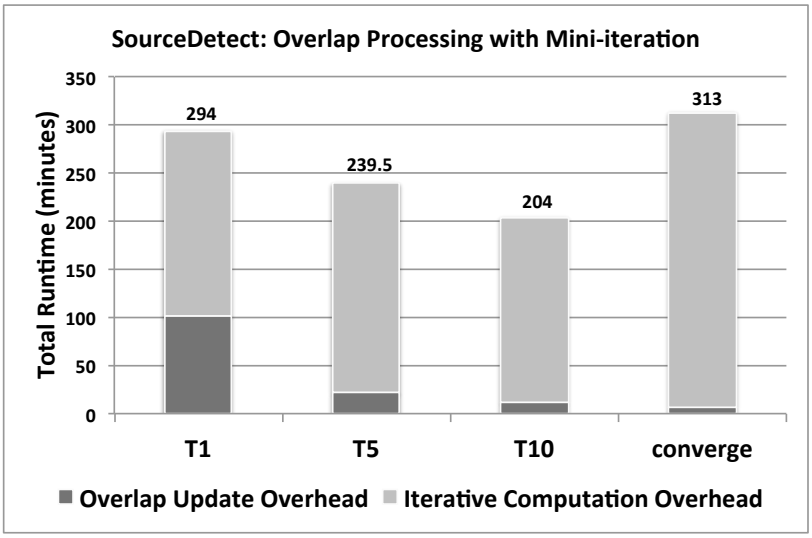
In Section 4.5, we describe overlap processing as a technique to support *parallel* array processing. In the case of an iterative computation, the challenge is to keep the overlap data up-to-date as the iteration progresses. The solution is to efficiently shuffle overlap data at each iteration. An optimization applicable to many applications is to perform `mini-iteration` processing, where

the shuffling happens only periodically. Figure 4.12(a) shows the effectiveness of this optimization in the context of the `SourceDetect` application, which requires overlap processing. `T1` refers to the policy where ArrayLoop shuffles overlap data at each iteration, or no `mini-Iteration` processing. As expected this approach incurs considerable data shuffling overhead, although it converges faster in the `SourceDetect` application (Figure 4.12(b)). At the other extreme, we configure ArrayLoop to only shuffle overlap data after local convergence occurs in all the chunks. Interestingly, this approach performs worse than $T1$. Although this approach does a minimum number of data shuffling, it suffers from the long tail of mini-iterations (Figure 4.12(b): 94 mini-iterations). $T5$ and $T10$ are two other approaches, where ArrayLoop shuffles data with some constant interval. We find that $T10$, which shuffles data every ten iterations, is a good choice in this application. The optimal interval is likely to be application-specific and tuning that value automatically is beyond the scope of this paper. The other interesting approach is to instruct ArrayLoop to initiate overlap data shuffling when the number (or magnitude) of changes between mini-iterations is below some threshold. We simply pick a constant number to determine the overlap data shuffling interval in the context of the `SourceDetect` application. More sophisticated approaches are left for future study.

4.7.3 Performance of Multi-Resolution Optimization

The `multi-resolution` optimization is a form of prioritized processing. By first processing a low-resolution approximation of the data, we focus on identifying the overall shape of the structures. Further processing of higher-resolution (larger) arrays then extracts the more detailed outlines of these structures. Figure 4.13(a) shows the benefits of this approach in the context of the `SourceDetect` application. We generate four lower-resolution versions of the source array A0 by sequentially calling the `grid()` operator with the grid-size of (2×2). We operate on these multi-resolution versions exactly as described in Equation 4.4. The performance results are compared to those of `T10` from Figure 4.12(a) as we pick the same overlap-processing policy to operate on each multi-resolution array. Interestingly, the `multi-resolution` optimization cuts runtimes nearly in half. Note that most of the saving comes from the fact that the algorithm converges much faster in $A0$ compared to its counterpart $T10$ (Figure 4.13(b)) thanks to the previous runs over arrays $A1$ through $A4$, where most of the cell-points are already labeled with their final cluster values.

Figure 4.12 `SourceDetect` application: Iterative overlap processing with mini-iteration optimization.



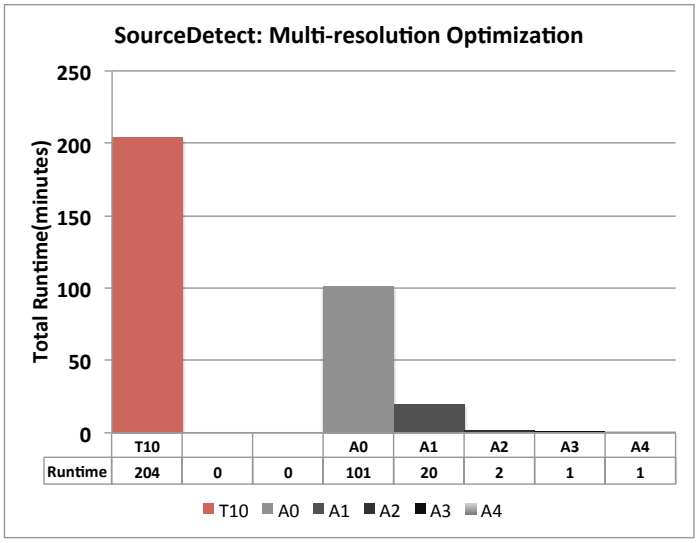
(a) `SourceDetect` application: T1, T5, and T10 refer to policies where ArrayLoop shuffles overlap data every iteration, every 5 iterations, and every 10 iterations, respectively. `converge` is the strategy where ArrayLoop shuffles data only after local convergence occurs.

	T1	T5	T10	converge
mini#	51	57	60	94
major#	51	11	6	3

(b) Number of major and mini iterations. Major# is the number of times that overlap data is reshuffled and Mini# is the total number of iterations.

In Section 4.6, we described a potential optimization in case of input data changes in the original array. As an initial evaluation of the potential of this approach, we modify the input data by dropping one image from the large, 3D array. This change is consistent with the LSST use-case, where a new set of images will be appended to the array every night. We observe that the new co-added image only differs in a small number of points from the original one. Additionally, these changes do not affect the pixelated array A1. This gives us the opportunity to re-compute the `SourceDetect` application not from the beginning, but from the pixelated version A1. Although the performance gain is not major in this scenario, it demonstrates the opportunity for further novel optimizations that we leave for future work.

Figure 4.13 `SourceDetect` application: Multi-resolution Optimization.



(a) `SourceDetect` application: T10 refers to the strategy where Array-Loop shuffles overlap data every 10 iterations. A0, A1, A2, A3, and A4 are five versions of the same array with different resolutions, where A0 is the same resolution as the original array and A4 is the most pixelated version. The grid-size is (2×2).

	T10	A0	A1	A2	A3	A4
iter#	60	35	12	12	6	10

(b) Iteration# that converges at each resolution.

4.8 Conclusion

In this chapter, we developed a model for iterative processing in a parallel array engine. We then presented three optimizations to improve the performance of these types of computations: incremental processing, mini-iteration overlap processing, and multi-resolution processing. Experiments with a 1TB scientific dataset show that our optimizations can cut runtimes by 4-6X for incremental processing, 31% for overlap processing with mini-iterations, and almost 2X for the multi-resolution optimization. Interestingly, the optimizations are complementary and can be applied at the same time, cutting runtimes to a small fraction of the performance without our approach.

Chapter 5

ASCOTDB: DATA ANALYSIS AND EXPLORATION PLATFORM FOR ASTRONOMERS

In previous chapters, we tackled some challenging problems in the context of building an array-based system. We introduced ArrayStore (Chapter 2) which provides efficient storage management techniques to store an array on disk. We built TimeArr (Chapter 3), a second storage manager with efficient support for updates and data versioning. We also prototyped ArrayLoop (Chapter 4), an array-query executor and extended storage manager with support for efficient iterative computations. In this chapter, we present AscotDB a data analysis system and service that we built on top of SciDB to enable efficient analysis of telescope images. AscotDB is an example application that leverages a parallel array-based system, including several of the iteration-related optimizations from Chapter 4.

As we described in the introduction (Chapter 1.1.1), astronomy, like other scientific fields, is currently moving to a new realm of research driven by large datasets. A famous example is the Large Synoptic Survey Telescope (LSST), which will accumulate a large database of telescope images of the visible sky. In order to bring a truly transformative science from the LSST dataset, astronomers need powerful engines with the ability to directly analyze the *pixelated raw* images. The system must enable *interactive and exploratory* computation and visualization of the data, which are essential first steps that inform further, more in-depth analysis.. Now the question is: *Can we provide the scientific community with such a transformative tool?* The AscotDB [62, 106] system has emerged in answer to this question.

AscotDB is a new tool for the analysis of telescope image data. While based on astronomy as its key application-domain, AscotDB primitives are general enough to be applicable to other scientific fields. AscotDB integrates several pieces of technology: the SciDB [87] engine for data storage and processing, Python for easy programmatic access (the programmatic access is not a contribution of this thesis), and The AStronomy COllaborative Toolkit (Ascot) [58] for graphical data exploration. None of these technologies solves the LSST large-scale data analytics and data management problems

on its own. More importantly, their naïve combination is also insufficient.

Ascot [58][1] developed in the Astronomy community, is a collection of Web-based gadgets that facilitate collaboration between astronomers. These gadgets are assembled into a dashboard and communicate using a node.js server. Through the use of a customizable dashboard interface, users can easily visualize, manipulate, and share large data sets from many different sources. Ascot, however, only enables the display of individual images and the analysis of pre-processed *catalog* data. It does not support the large-scale analysis of pixel data.

AscotDB provides a compelling and powerful environment for exploration, visualization, and collaborative analysis of large telescope image datasets. AscotDB further contributes several techniques: (1) It demonstrates a scientifically useful integration of Ascot and SciDB. (2) It proves the success of integration by demonstrating [62] the data exploration and analysis enabled by this integrated tool on a terabyte-sized dataset. (3) Finally, It demonstrates the necessity of extending SciDB with native support for efficient iterative computations (Chapter 4). Other contributions of AscotDB that are not part of this thesis include: (1) A more intuitive Python language bindings through a new Python package called SciDB-Py[2]. (2) A Python middleware that enables efficient storage and manipulation of spherical data [106].

In the rest of this chapter, we present an overview of the AsctoDB system, its architecture, its components, and how user interacts with the AscotDB front-end.

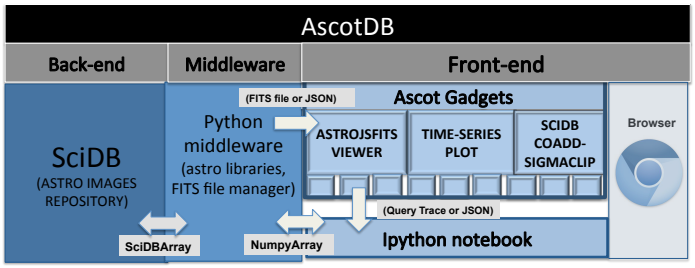
5.1 AscotDB Overview

The basic operations astronomers perform on images fall into two categories: detection and measurement. Detection is the process of identifying the location of individual sources in an image, where the sources might be stationary objects such as individual stars and galaxies, or moving/transient objects such as asteroids or supernovae. Measurement involves computing well-calibrated statistics from the light of the individual objects: for example, computing the total optical flux from a star through a detailed model of its response across the CCD pixels. AscotDB provides the user with two modes of interaction with the data: (1) Visual interaction with Ascot gadgets

[1] `http://ascot.github.io/`

[2] `http://jakevdp.github.io/SciDB-Py`

Figure 5.1 AscotDB architecture: SciDB as back-end, python middleware, Ascot and IPython as front-ends.



that we describe in Section 5.2.1 – these visual interactions are an important component of the detection process; and (2) an IPython interface for programmatic interaction for both detection and measurement. The latter is not a contribution of this thesis and we refer the reader to the AscotDB overview paper for details [106]. The overall architecture of AscotDB and the high level interaction between components are illustrated in Figure 5.1.

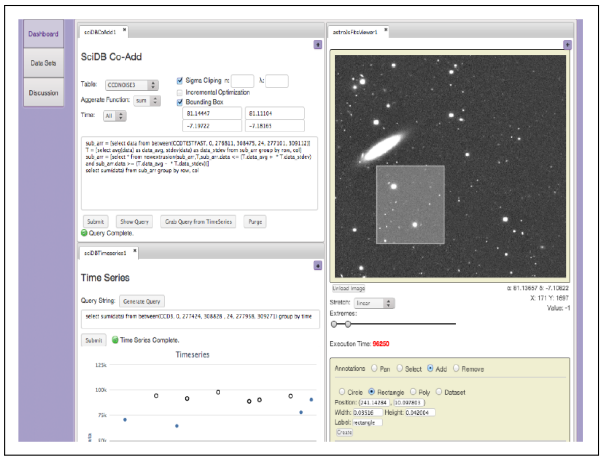
5.2 AscotDB Front-end and User Interactions

AscotDB provides the user with two modes of interaction with the data: (1) Visual interaction; and (2) Programmatic interaction. These two modes are illustrated in Figure 5.2.

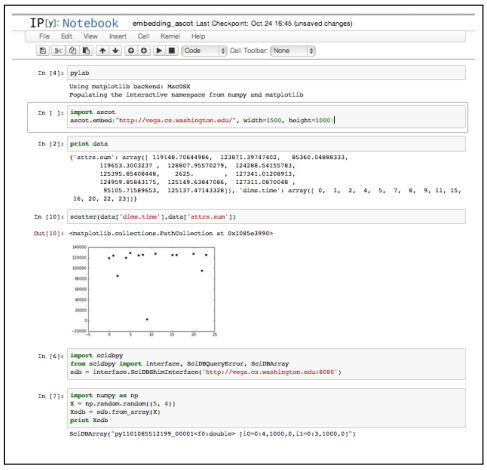
5.2.1 Graphical and Programmatic Front-End Support

Figure 5.2(a) shows a screenshot of AscotDB's graphical interface. AscotDB retains important Ascot features including its extensibility through the addition of new gadgets and its sharing capabilities across users. AscotDB, however, radically transforms Ascot's data analysis capabilities. Originally, Ascot enabled users to view a *single* telescope image at a time and overlay on the image catalog data extracted from a back-end relational DBMS. In contrast, AscotDB enables users to manipulate raw pixel data. For example, users can *stack* images of the sky that fall within a region, *clean them* using an iterative process, re-run a source detection algorithm, annotate interesting sources, and generate visual summaries (e.g., light curves) before initiating a measurement process. To support these novel operations, AscotDB stores the telescope image data inside SciDB and translates operations on the interface into queries over SciDB's arrays. Because scientists are technically savvy users and because graphical interfaces can inaccurately capture a user's intent, AscotDB always shows the queries that

Figure 5.2 Two modes of interaction with AscotDB: visual interface and IPython interface. The visual interface is embedded in IPython, but here is shown separately for purposes of illustration.



(a) Graphical front-end of AscotDB with a pixel image, a time-series chart, and a SciDB interface



(b) Programmatical front-end of AscotDB.

it generates and enables a user to modify them before executing them.

A second important design decision in AscotDB's graphical front-end was to strike a balance between a tool specialized for one analysis task and a general-purpose system such as Tableau [103]. For AscotDB, we opted to develop one gadget per high-level activity (*e.g.*, data cleaning, image stacking, and time-series extraction) since the community typically performs a small set of such activities. AscotDB enables users to assemble these gadgets and thus activities in arbitrary combinations.

AscotDB also includes a Python interface for programmatic interaction which is not the contribution of this thesis. The Python interface to AscotDB is provided via the IPython notebook as illustrated in Figure 5.2(b). IPython is a set of tools designed to facilitate the entire life-cycle of a scientific project, from data exploration to publication. One component is a browser-based *notebook* with the support for editing and running Python code, as well as rich objects such as embedded plots, html objects, and mathematical expressions. Integrating the interactive, visual scientific computing environment of IPython interface within the AscotDB environment enables seamless sharing and processing of large astronomical datasets.

5.2.2 User Interaction with AscotDB

We now describe the AscotDB capabilities in more detail. There are two fundamental phases as the user interacts with AscotDB: the *Analysis phase* and the *Exploration phase*.

Analysis Phase In this phase, the user can run and observe the results of the "co-addition" (stacking images together to detect faint sources) and the "sigma-clipping" (iterative data cleaning algorithm) tasks (Example 1.1.1 for more details). The user selects the region of the sky that forms the input to the analysis. The user also has the choice to run the "sigma-clipping" algorithm in a naïve way or to enable incremental iterative processing optimization (Section 4.4) and observe the performance differences. The user can also tune parameters, such as the threshold to filter outliers in "sigma-clipping" algorithm, to observe its effect on the number of iterations until convergence. The sequence of tasks in this phase is illustrated in Figure 5.3(a).

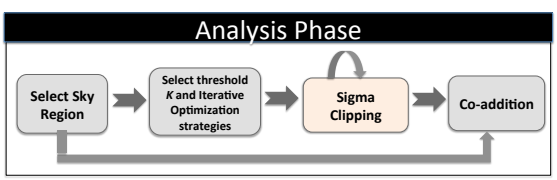
Exploration Phase AscotDB supports multiple exploratory tasks such as viewing images, querying catalogs, and plotting time-series data. This phase is illustrated in Figure 5.3(b). We describe the time-series capability in more detail: the user can select an arbitrary region on the sky and generate a time-series for that region as shown in Figure 5.2(a). The time-series plot that AscotDB supports shows the value of one of the attributes (*e.g.* flux value) over time. The value shown is an aggregate value computed over the entire selected region.

Each point in the time-series plot corresponds to one timestep in our original pixel images. AscotDB gives the option to the user to select *interesting* points in the time series, filter out the rest, and redo the analysis (*i.e.*, sigma-clipping and co-addition) on a subset of pixel images corresponding to the interesting points in the time series. As Figure 5.4 illustrates, the output of the exploration phase is fed back as refined input to the next analysis phase.

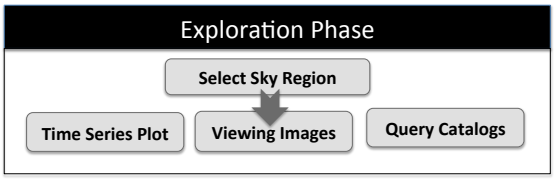
5.3 AscotDB Middleware Python Support and SciDB backend

AscotDB is a layered system. It builds on a Python middleware and SciDB back-end to enable both exploratory and deep analysis of the data. We now describe these two layers in more details.

Figure 5.3 "Analysis" and "Exploration" phases

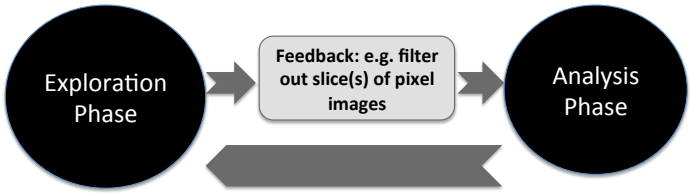


(a) Analysis phase: The user selects a region on the sky, tunes parameters, and runs the "sigma-clipping" algorithm followed by image "co-addition". The user also has the option to perform "co-addition" on the original images directly.



(b) Exploration phase: AscotDB supports multiple exploratory tasks such as viewing images, querying catalogs, and plotting time-series data.

Figure 5.4 The user interacts with AscotDB by alternating between exploration and analysis phases.



5.3.1 Middleware Python

Python has seen significant uptake among astronomers and other domain scientists because it is open-source, cross-platform, and easy for non-programmers to learn. The scientific Python ecosystem, built around the core tools of NumPy & SciPy [69], Matplotlib [42], and IPython [78] (including the web-based interface of the *IPython notebook*), provides a complete environment for the analysis and visualization of data at a small to medium scale. Due to its advantages, a wide selection of current and future astronomical surveys are now building their data analysis pipelines using Python. Of particular relevance for this work, the LSST project plans to use Python as their primary data pipeline interface. In this environment, it is increasingly important for data analysis tools to provide Python-hooks for any computing infrastructure. Python alone, however, provides limited data management capabilities and is non-trivial to use on LSST-sized datasets.

AscotDB includes a Python middleware, illustrated in Figure 5.1, that wraps around the SciDB backend and provides a foundation for a host of more specialized packages implementing algorithms used in a wide variety of scientific fields. Current implementation of Python middleware hosts

libraries to transform SciDB query result into FITS[3] file format and essential specialized Python libraries to transform sky coordinates to the underlying SciDB array coordinates and vice versa.

The graphical front-end is layered on top of this python middleware and leverages both the FITS file and spherical coordinates capabilities. The Python front-end through SciDB-py also uses these middlware capabilities.

5.3.2 SciDB Back-end

AscotDB stores the LSST image data inside SciDB. Several astronomy operations including source detection and data cleaning are iterative in nature and they are expensive to run in SciDB. To support these iterative tasks at interactive speeds (Figure 4.5), AscotDB uses the iterative array processing extensions in SciDB that we developed and we present in Chapter 4. The support includes the execution of such iterations in shared-nothing clusters and various optimizations such as incremental processing (Section 4.4). We presented an overview of SciDB in Section 1.2. For further information about SciDB, we refer the reader to the SciDB overview paper [87].

5.4 AscotDB Lessons Learned and Future Directions

Efficient data exploration, visualization, and analysis in the context of future large astronomical surveys will require a combination of advances in the areas of large-scale data storage, processing, and visualization. We have presented one set of approaches to this problem: AscotDB. While we demonstrate the capabilities of our solution in the context of the analysis of astronomy telescope images, we emphasize that AscotDB primitives are general enough to be applicable to other scientific fields.

One important requirement of AscotDB is the integration of both the graphical and python modes of interaction such that the user can go seamlessly back and forth between them. The main question to be answered is how to keep the working data in both modes synchronized. One possibility is to keep track of all the actions in the Ascot front-end in the form of SciDB queries and, after switching the mode to Python, have the system run the same queries in the same order in the background. This approach makes it easier to track the lineage of data and thus facilitates reproducibility. However,

[3] `http://fits.gsfc.nasa.gov/fits_standard.html`

a session where a user interacts with a graphical engine can be long, can contain a large number of queries, and can dwarf the remaining Python script. Another approach is to move minimal amounts of data between the two interfaces to keep the working data synchronized. We prefer synchronizing the two interfaces by capturing user queries because of this approach's lineage tracking and reproducibility properties. To address the problem of large query sessions, however, in the AscotDB system, an interesting area of future work is to experiment with a variety of techniques to automatically extract minimal query sets, which produce the results from a visual analysis session. The key aim is to minimize the number of queries but without re-ordering them nor combining them into more complex and difficult-to-understand queries. It is critical for the user to identify in the summary script all the key steps that he or she took during the visual exploration.

Chapter 6

RELATED WORK

There are multiple lines of research that are related to this thesis work. In this chapter, we cover the literature in the following areas: Array processing and systems, array data storage, array data versioning, and iterative processing.

6.1 Related Work on Array Processing and Systems

Many engines are being built today to support multidimensional arrays [3, 15, 20, 28, 33, 87, 110]. RasDaMan [28] is a multidimensional array DBMS implemented on top of a relational DBMS. Array chunks in RasDaMan are stored as BLOBs in the underlying database. The array processing in RasDaMan is through a set of operators that are integrated into the SQL language. A few optimization techniques in RasDaMan are loop fusion, dynamic compilation, and GPU support. Loop fusion optimization merges atomic operation nodes in the query plan tree into one. In our work, we developed several array storage and processing methods, and implemented them as extensions to SciDB, to provide efficient support for versioning and iterative computations, features that are not well-supported by RasDaman.

MonetDB [3, 43] is a columnar database with support for array processing through the RAM [105] and the SRAM [22] systems. RAM and SRAM do not run in parallel and they do not provide native support for arrays. Arrays in those systems are represented as relations and array operations are mapped onto relational algebra operators.

Many engines support parallel array processing [3, 15, 20, 28, 33, 87]. A well-studied approach for processing array operations in parallel is to divide the original array into multiple subarrays and run the operation independently on each subarray. If necessary, the results from local computations are post-processed (*a.k.a.*, "rolled-up" or "merged") to obtain the final output [3, 15, 20, 28, 48, 49, 87]. Additionally a few engines [87] also have the ability to run queries with overlap and potentially avoid the merge processing overhead. The prototypes that we developed in this thesis support parallel

array processing by both partitioning the array into smaller subarrays and also running queries with overlap.

SciDB [87] is an open-source database system with inherent support for multi-dimensional arrays. SciDB is a parallel, distributed data management system based on shared-nothing architecture. It provides parallelism by partitioning array data into multiple sub-arrays called chunk. It further parallelizes queries using overlap processing strategy (rather than post-processing merge). The application layer in SciDB has full support for both declarative and functional languages. There are bindings from popular languages such as Python and R to SciDB as well. Our work develops techniques that can be integrated into a system such as SciDB and several of our prototypes (ArrayLoop in Chapter 4 and TimeArr in Chapter 3) have been developed as extensions to SciDB.

Parallel programming languages such as UPC [104], Global Arrays [65], and Co-Array Fortran [68] faciliate the task of coding parallel array processing applications. Those parallel languages provide flexible access to remote array partitions providing the abstraction of a shared memory address space. They target applications where a given local chunk may need to access an arbitrary set of remote chunks. In contrast, "overlap" and "merge" execution techniques in array engines such as SciDB and RasDaMan provide more loosely coupled parallelism for applications where the computation of a given chunk is restricted to its adjacent chunks in the context of shared nothing architectures.

The Network Common Data Form (netCDF) [84, 85] and Hierarchical Data Format version 5 (HDF5) [37] are simple data models that provide a portable and efficient mechanism to store and access multidimensional data. They both simply organize data into regular arrays using traditional programming languages techniques. Neither is a database management system and thus is not not an alternative for array databases.

EXTASCID [16] is an extensible parallel system that provides native support both for arrays and key-value relations. The execution strategy in EXTASCID is through user defined aggregates (UDA) interface with easy reasoning for parallelism, while other array engines such as SciDB provide parallelism through array partitioning and overlap processing.

AstroShelf [64], similar to AscotDB, is a collaborative system that enables astrophysicists to investigate celestial objects using catalog data hosted at different sites. Astroshelf is a stream

processing system that matches events (either celestial events or derived annotations) to users who are likely to be interested in them. In contrast, AscotDB focuses on interactive, collaborative processing of archived data.

While SciDB does provide a low-level Python API for the execution of Array Query Language (AQL) and Array Functional Language (AFL) queries, the interface is too opaque for it to be useful to most astronomers. SciDB-Py [90][1] is an intuitive Python interface to SciDB. It is designed with an API familiar to users of the NumPy array computing library. The SciDB-py operations are executed entirely within the SciDB architecture through the automatically generated AFL queries. the actual AFL queries are transparent to the user, and the Python interpreter itself never sees the data. By providing such an intuitive, high-level wrapper around the efficient storage and operations available in SciDB, SciDB-py makes the power of SciDB accessible to the average scientific Python user.

Blaze [9], often billed as the "Next generation NumPy", aims to implement a very general array framework within Python. It will support a wide variety of table and array-like structures capable of handling arbitrary local and distributed memory layouts, type heterogeneity, axis labels, missing or masked values, and other commonly-requested features. SciDB is just one of a wide variety of potential backends for large-scale distributed array storage and computing through Blaze: in this light, SciDB-Py can be considered a precursor to the much more ambitious framework Blaze promises to implement. Our extensions in SciDB also provide SciDB AFL and AQL languages as query interface, so they are thoroughly compatible with Python interfaces to SciDB such as SciDB-py and potentially Blaze.

6.2 Related Work on Array Storage

Many existing array-processing systems [14, 25, 29, 87] use regular tiling for data storage. Others support user-defined irregular tiles [15]. In contrast to our work on ArrayStore (Chapter 2), none of these systems, however, studies the impact of different tiling strategies on query processing performance, although they do consider different tile layouts on disk [14] and across disks [14, 61] for range-selection queries.

MOLAP [31] systems store data in multidimensional arrays [31, 77]. They focus on aggregation

[1] `http://jakevdp.github.io/SciDB-Py`

queries and exploit data structures to efficiently compute rollups, while, in ArrayStore, we consider a broader set of operations. Today's business intelligence (BI) suites utilize closed source, proprietary MOLAP engine solutions such as Oracle Database OLAP Option [72], Cognos PowerPlay [18], and others to analyze large datasets. To the best of our knowledge, Palo [76] is the only open source memory-based MOLAP engine, which is specifically developed for spreadsheet data storage and analysis. However, Palo is not designed for large databases.

Furtado and Baumann [32] studied the performance of different tiling strategies in RasDaMan (including regular and irregular tiles). Their study, however, was limited to scans and different types of array dicing operations. Their conclusions are thus different from the ArrayStore since they find that arbitrary tiling tuned to a specific workload outperforms regular tiling. Reiner *et al.* [82] studied hierarchical storage support for large-scale multi-dimensional arrays in RasDaMan. Their approach is analogous to the two-level, IREG-REG, chunking strategy, described in ArrayStore. However, their study is constrained to range-selection queries.

Shimada *et al.* [94] propose a chunking scheme for sparse arrays, where multiple chunks are compressed and stored in a single physical page. This approach is analogous to the two-level, IREG-REG, storage system that we study in Chapter 2. Shimada *et al.*, also introduce "extended chunks", which are similar to IREG. Again, however, this earlier study was limited to range-selection queries.

Prior work studied the tuning of chunk shape, size, and layout on disk for a given workload and for regular chunking [74, 88]. This work is orthogonal to our work in Chapter 2, since we comparatively study regular *v.s.* irregular *v.s.* two-level chunking schemes and support for overlap data.

Seamons and Winslett [91] examine different storage management strategies for regularly-tiled arrays. In particular, they propose that data from multiple arrays be either stored separately or be interleaved on disk. This strategy is orthogonal to those we study in Chapter 2. They also consider storage strategies for overlap data mentioning both the option to store overlap data together with or separately from the core data. Their implementation and evaluation, however, only examine the co-located scenario, similarly to SciDB [87].

There exist many data structures for indexing multidimensional data including the R-Tree [35]

and its variants [1, 6], the KD-Tree [7], the KDB-Tree [86], the GammaSLK [55], the Pyramid technique [8], the Gamma strategy [73], RPST [81], and more. All these indexes organize a raw dataset into a multi-dimensional data structure to speed-up range-, containment-, and nearest-neighbor queries. In contrast, in ArrayStore, we study storage management techniques for more varied array operations.

6.3 Related Work on Array Data Versioning

There is a long line of research on temporal databases [46, 52, 75, 96]. Temporal databases have two notions of time: "valid time" and "transaction time". Many databases provide time-travel support along the transaction time dimension [41, 52, 71, 100]. However, none of these databases is specialized for time travel over array data nor approximate time-travel. In particular, Postgres [100] uses R-trees for version management. This technique is complementary to the approach that we propose in TimeArr (Chapter 3). Immortal DB [52] adds transaction time database support into a database engine. For this, Immortal DB stores versions data as a linked list, while we store versions as delta values. Our versioning system in TimeArr also heavily applies array-oriented techniques including bitmasks, virtual tiles, and skip links. Finally, TimeArr supports a new type of "approximate queries" in the context of scientific array database engines. Most array engines being built today, such as RasDaMan [28], are *not* designed as no-overwrite storage systems and consequently cannot naturally support versioning. NetCDF [84, 85] and HDF5 [37] are common data models that provide a portable and efficient mechanism to store and access multidimensional data which are extensively used by scientists, but they also do not support versioning explicitly.

MOLAP systems store data in multidimensional arrays [31, 77]. The MOLAP system in [47] supports versions to represent changes to the data sources that should be propagated to the data warehouse periodically. But the versioning system is designed to benefit the concurrency control mechanism in order to minimize contention between query and maintenance transactions.

Delta encoding is a popular technique in video and image compression. Video compression codecs like `MPEG-1` [17] apply several delta encoding techniques both within and between frames. Similar to TimeArr, they regularly materialize versions (frames) in a chain of delta frames, They also divide the frames into smaller chunks and compare each chunk to every possible region in a specified radius around its origin. Hence, their version insertion is expensive. Previous work [92] showed that

although video compression techniques efficiently compress arrays, the version import time is too expensive and consequently not appropriate for versioning in array systems.

Version Control Systems are an old topic in computer science. Versioning techniques such as forward and backward delta encoding and the use of multi-version B-trees have been implemented in various legacy systems. Git [12] is one of the conventional version-control systems and is believed to be faster and more disk efficient than other similar version-control systems. TimeArr borrows some ideas such as backward delta encoding from other version control systems such as Git, but we also use sophisticated array-oriented optimization techniques to efficiently encode the delta versions and to support approximate queries.

Lastly, the state of the art for versioning in array systems [92] uses a materialization matrix to efficiently find the best versions to materialize. TimeArr is similar to this recent prior work [92] in the sense that we also use backward delta versions and store and fetch consecutive deltas together. The ability of TimeArr to add skip links at the granularity of tiles, to approximately answer queries, and our use of virtual tiles to support versioning at fine granularity are the main advantage of our system compared to this prior work [92].

6.4 Related Work on Iterative Processing

Several systems have been developed that support iterative big data analytics [10, 30, 53, 93, 109]. Some have explicit iteration, others require an external driver to support iteration, but none of them provide native support for iterative computation in the context of parallel array processing.

Twister [27], Daytona [4], and HaLoop [10] extend MapReduce to preserve state across iterations and support for looping construct. HaLoop takes advantage of the task scheduler to increase local access to the static data. However, our system takes advantage of iterative *array* processing to increase local access to the dynamic data as well by applying overlap iterative processing.

PrIter [109] is a distributed framework for fast iterative computation. The key idea of PrIter is to prioritize iterations that ensure fast convergence. In particular, PrIter gives each data point a priority value to indicate the importance of the update and it enables selecting a subset of data rather than all the data to perform updates in each iteration. ArrayLoop also supports a form of prioritized processing through multi-resolution optimization. ArrayLoop initially finds course-grained outlines

of the structures on the more pixelated versions of the array, and then it refines the details on fine-grained versions.

REX [60] is a parallel shared-nothing query processing platform implemented in Java with a focus on supporting incremental iterative computations in which changes, in the form of deltas, are propagated from iteration to iteration. Similar to REX, ArrayLoop supports incremental iterative processing. However REX lacks other optimization techniques that we provided.

A handful of systems exist that support iterative computation with their focus on graph algorithms. Pregel [57] is a bulk synchronous message passing abstraction where vertices hold states and communicate with neighboring vertices. Unlike Pregel, ArrayLoop relieves the synchronization barrier overhead by including mini-iteration steps in the iterative query plan. Unlike ArrayLoop, Pregel does not prioritize iterative computation.

GraphLab [53] develops a programming model for iterative machine learning computations. The GraphLab abstraction consists of three main parts, the data graph, the dynamic asynchronous computation as update functions, and the globally sync operation. GraphLab has configurable consistency levels and update schedulers, making it a powerful, but with low level abstraction. Similar to our overlap iterative processing technique in ArrayLoop, GraphLab has similar notion of *ghost* nodes, however the granularity of computation is per node, while ArrayLoop supports overlap iterative processing per chunk. Our system also supports prioritization through the novel multi-resolution iterative processing.

Chapter 7

EVALUATION DATASETS

This chapter describes the datasets used in the evaluation of the array storage and processing techniques developed in this thesis.

Astronomy Universe Simulation dataset (Astro): The Astro dataset comprises nine snapshots from a large-scale astronomy simulation [51] for a total of 74GB of data. The simulation models the evolution of cosmic structure from about 100K years after the Big Bang to the present day. Each snapshot represents the universe as a set of particles in a 3D space, which naturally leads to the following schema: `Array Simulation {id,vx,vy,vz,mass,phi} [X,Y,Z]`, where X, Y, and Z are the array dimensions and `id, vx, vy, vz, mass, phi` are the attributes of each array cell. `id` is a signed 8 byte integer while all other attributes a 4 byte floats. Each snapshot is stored in a separate array. Since the universe is becoming increasingly structured over time, data in snapshot $S92$ is more skewed than in $S43$. In Figure 2.3, the largest regular chunk has 25X more data points than the smallest one. The ratio is only 7 in $S43$ for the same number of chunks.

Oceanography Flow Cytometer dataset (OceanFlow): The oceanography dataset is the output of a flow cytometer [2]. A flow cytometer measures scattered and fluoresced light from a stream of water particles. Similar microorganisms exhibit similar intensities of scattered light. In this dataset, the data takes the form of points in a 6-dimensional space, where each point represents a particle or organism in the water and the dimensions are the measured properties. We thus use the following schema for this dataset: `Array Cytometer {day, filenumber, row, pulseWidth, D1, D2} [FSCsmall, FSCperp, FSCbig, PE, CHLsmall, CHLbig]`, where all attributes are 2-byte unsigned integers. Each array is approximately 7 GB in size.

Global Forecast System Model dataset (GFS): "GFS" is a dataset from the National Oceanic and Atmospheric Administration (NOAA) [67]. This dataset is the output from 180 hours of simulation based on a global forecast system weather model and contains data for a grid of locations covering the whole earth. Each type of measurement such as snow cover or inches of rain is stored as a floating-point number in its own versioned grid. We use the output of a 180 hour weather forecast simulation sampled every 3 hours, for a total of 61 grids, each about 1MB in size. Each grid is a (720×360) two dimensional array (one array in one chunk).

Large Synoptic Survey Telescope (LSST): The Image Co-Addition benchmark[1] records raw astronomy data in the form of 2D images output by a telescope. The data in this benchmark comes from an LSST image simulator developed by the LSST team. It comprises 4725 images over 25 time steps. The LSST [54] telescope is going to visit sky each night. A visit is one exposure on the sky and is made up of 189 individual 2D ccd images stored in HDF format . The ccd images are grouped into 21 individual rafts (9 per raft). We load all the simulated ccd image into the SciDB and represent it as a 3D array with the following schema: `Array CCD <data:float NULL,mask:int16 NULL,var:float NULL> [row(int64),col(int64),time(int64)]`, where `data`, `mask`, and `var` are attributes in each cell and `row`,`col`, and `time` are three dimensions. Total number of points is approximately 44 billions and the total size is approximately one terabyte of data.

[1] http://myria.cs.washington.edu/repository/uw-cat.html

Chapter 8

CONCLUSION AND FUTURE DIRECTIONS

In this thesis work, we developed efficient query processing, versioning, and storage techniques for parallel array database systems. The contributions of this thesis are as follows:

Array Data Processing and Storage: In Chapter 2, we presented the design, implementation, and evaluation of ArrayStore, a storage manager for complex, parallel array processing. In ArrayStore, we studied the impact of different chunking strategies on query processing performance for a wide range of operations, including binary operators and user-defined functions. We showed that a two-level chunking strategy with regular chunks and regular tiles (REG-REG) leads to the best and most consistent performance for a varied set of operations both on a single node and in a shared-nothing cluster. We presented two new techniques to support efficient processing of overlap data (i.e., data in neighboring array chunks): one leverages ArrayStore's two-level storage layout and the other one uses additional materialized views. Both techniques significantly outperform approaches that do not provide overlap or provide only a pre-defined single overlap layer.

Array Data Versioning: In Chapter 3, we presented TimeArr, a new storage manager for array databases that provides a no-overwrite, versioned array storage model. TimeArr's key contribution is to efficiently store and retrieve versions of an entire array or some sub-array. TimeArr also introduces the idea of approximate exploration of an array's history. To achieve high performance, TimeArr relies on several techniques including virtual tiles, bitmask compression of changes, variable-length delta representations, and skip links.

Iterative Array Processing: In Chapter 4, we presented ArrayLoop, an extension of SciDB that adds native support for iterative computation. First, we developed a model for iterative processing in a parallel array engine. We then presented three optimizations to improve the performance

of these types of computations: incremental processing, mini-iteration overlap processing, and multi-resolution processing.

AscotDB: Finally, in Chapter 5, we presented AscotDB, a new, extensible data analysis system for the interactive analysis of telescope image data. We designed AscotDB as a layered system with a rich graphical interface as front-end that is built on top of a Python middleware and the SciDB parallel array engine as back-end.

We validated our contributions through extensive evaluations with real datasets described in Chapter 7. Several open research problems follow directly from the results presented in this thesis. First, the interested reader may ask how best to handle workloads that are a mix of array and non-array datasets and operations. One example are 2D satellite images with meta data associated with them. Should both the data and metadata be stored in the array engine? Is the overhead of adding artificial structure to the non-array meta data significant? or perhaps we should store satellite images (array data) in array engines and meta data (non-array data) in a relational engine. Second, it is very hard for the user of an array engine to figure out the right chunk size. Are there approaches that help the system to figure out the chunk size (semi) automatically? Third, we did not explore the nested array data representation. What are the challenges and difficulties of supporting this feature? How much more complexity it adds to the query execution layer? Fourth, the user has to provide a complete schema of an array, e.g. type of dimensions, type of attributes, starting index and ending index for each dimension, chunk size, overlap size, before doing any exploration or analytics on that data. Can we relax some of the conditions of this rigid schema to speed up the initialization phase?

Beyond the direct future work described above, several longer-term research directions would further facilitate large-scale array processing:

Support data visualization as first-class citizen: Current big data management tools often leave the data visualization task to the user. To visualize the result, the user either delegates the responsibility to a general-purpose system such as Tableau [103] or writes a customized application [62]. Array data analytics systems should integrate the visualization task into the query itself and propagate the query plan, which is augmented with the visualization query, down to the query processing layer on the server-side to leverage possible optimizations. For example, the system could leverage the fact

that the user visualizes the query result in an aggregated (pixelated) form [5], and could possibly push the aggregation down the query plan or even return an approximate but fast result. Furthermore, we regard the future data analytics pipeline as a cycle consisting of a sequence of exploration and analysis phases. As we discussed in Chapter 5, users need support for easily alternating between data exploration using a visual interface and deeper analysis using a programming interface (e.g., Python). Data visualization plays an important role during the exploration phase where users detect patterns or anomalies in results. Although initial efforts [5] are ongoing to support data visualization as a first-class citizen for array data management systems, none yet provides a seamless interaction between exploratory and analysis phases.

Support large-scale interactive stream analytics: Modern array data management systems should provide a compelling and powerful environment for the exploration, analysis, and visualization of data with the ability for interactive feedback from the client-side. Smart devices are making the world more connected and the number of interfaces for data analytics tools to collect data is rapidly increasing. Examples are human clicks, sensors, mobile devices, Google glass, Nest, etc. As a results array data management systems will need to handle these data streams, which requires in situ data processing. The data from these streams could be relevant only for a few steps in the analytics pipeline and not be suitable for offline data mining. Others may require long-term storage and processing. Current large-scale data management solutions are either designed and optimized for stream data analytics or offline data analytics. Future data management tools, in order to maintain the cycle of data exploration and analytics, should provide efficient support for the interactive analysis of heterogenous large-scale data streams as well as the offline data mining of large-scale collected data; a requirement that is not fully satisfied by current data management systems including array-based engines. Interactive analytical capability is a requirement for stream processing engines. In array-based systems, we can provide interactive analytics either through sophisticated sampling or by providing array at different levels of granularity. The latter is partially studied in Chapter 4 where we leverage array data at different levels of granularity for iterative computation. We believe caching array data with different resolutions has more potential to speed up different kinds of queries including those that require interactive analytics but tolerate approximated results and it requires further study.

Support large-scale data analytics as a service: Finally, cloud-based analytical capabilities ease the adaptation for many companies and end-users. Modern array data management systems should provide big data analytical capabilities using cloud delivery models over a varied set of use cases. Future data management tools should support both requirements of *large-scale* and *as a service* data analytics at the same time. Array engines are not an exception. Array data analytics services in the cloud bring multiple challenges including the proper design for the service model, storage and access methods, data model, and networking model. They also need to take into consideration advances in computer architecture.

BIBLIOGRAPHY

[1] Arge et. al. The priority r-tree: a practically efficient and worst-case optimal r-tree. In *Proc. of the SIGMOD Conf.*, pages 347–358, 2004.

[2] SeaFlow cytometer. `http://armbrustlab.ocean.washington.edu/resources/sea_flow`.

[3] Ballegooij et. al. Distribution rules for array database queries. In *16th. DEXA Conf.*, pages 55–64, 2005.

[4] R. Barga et al. Daytona: Iterative MapReduce on Windows Azure. `http://research.microsoft.com/en-us/projects/daytona/default.aspx`.

[5] L. Battle, M. Stonebraker, and R. Chang. Dynamic reduction of query result sets for interactive visualization. 2013.

[6] Beckmann et. al. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Record*, 19(2):322–331, 1990.

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

[8] Berchtold et. al. The pyramid-technique: towards breaking the curse of dimensionality. In *Proc. of the SIGMOD Conf.*, pages 142–153, 1998.

[9] Blaze: A python compiler for big data. `http://continuum.io/blog/blaze`.

[10] Y. Bu et al. HaLoop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1), 2010.

[11] European Organization for Nuclear Research. `http://home.web.cern.ch/about/computing/`.

[12] S. Chacon. the Git SCM community. `http://book.git-scm.com/`, 2010.

[13] 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 *Proc. of the 7th USENIX Symp. on Operating Systems Design & Implementation (OSDI)*, 2006.

[14] Chang et. al. Titan: A high-performance remote sensing database. In *Proc. of the 13th ICDE Conf.*, pages 375–384, 1997.

[15] Chang et. al. T2: a customizable parallel database for multi-dimensional data. *SIGMOD Record*, pages 58–66, 1998.

[16] Yu Cheng and F. Rusu. Astronomical data processing in extascid. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 47:1–47:4, 2013.

[17] International Standards Organization (ISO), Coding of Moving Pictures and Audio. `http://mpeg.chiariglione.org/standards/mpeg-1/mpeg-1.htm`.

[18] Cognos PowerPlay. `http://www-01.ibm.com/software/data/cognos/products/series7/powerplay/`.

[19] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD skills: new analysis practices for big data. *Proc. VLDB Endow.*, 2:1481–1492, August 2009.

[20] Cohen et. al. Mad skills: new analysis practices for big data. *vldbj*, 2(2):1481–1492, 2009.

[21] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[22] Roberto Cornacchia, Sándor Héman, Marcin Zukowski, Arjen P. Vries, and Peter Boncz. Flexible and efficient ir using array databases. *The VLDB Journal*, pages 151–168, 2008.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th Symposium on Operating System Design and Implementation*. Google, Inc.

[24] DeWitt et. al. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[25] DeWitt et. al. Client-server paradise. In *Proc. of the 20th Int. Conf. on Very Large DataBases (VLDB)*, pages 558–569, 1994.

[26] The earth microbiome project. `http://www.earthmicrobiome.org/`.

[27] J. Ekanayake et al. Twister: a runtime for iterative MapReduce. In *HPDC*, pages 810–818, 2010.

[28] Baumann et. al. The multidimensional database system RasDaMan. In *Proc. of the SIGMOD Conf.*, pages 575–577, 1998.

[29] Marathe et. al. Query processing techniques for arrays. *The VLDB Journal*, 11(1):68–91, 2002.

[30] S. Ewen et al. Spinning fast iterative data flows. In *Proc. of the 38th Int. Conf. on Very Large DataBases (VLDB)*, pages 1268–1279, 2012.

[31] P. M. Fernandez. Red brick warehouse: a read-mostly rdbms for open smp platforms. *Proc. of the SIGMOD Conf.*, pages 492–, 1994.

[32] Furtado et. al. Storage of multidimensional arrays based on arbitrary tiling. In *Proc. of the 15th ICDE Conf.*, page 480, 1999.

[33] S. Goil and A. Choudhary. Parsimony: An infrastructure for parallel multidimensional analysis and data mining. *Journal of Parallel and Distributed Computing*, pages 285 – 321, 2001.

[34] Gray, J. et. al. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, pages 29–53, 1997.

[35] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. of the SIGMOD Conf.*, pages 47–57, 1984.

[36] Hadoop. `http://hadoop.apache.org/`.

[37] Introduction to HDF5. `http://www.hdfgroup.org/HDF5/doc/H5.intro.html`.

[38] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proc. of the SIGMOD Conf.*, 1997.

[39] Hey et. al., editor. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[40] Hive. `http://hadoop.apache.org/hive/`.

[41] L. Hobbs and K. England. Rdb: A Comprehensive Guide. Digital Press, 1995.

[42] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[43] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull*, 2012.

[44] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of the European Conference on Computer Systems (EuroSys)*, pages 59–72, 2007.

[45] Ž. Ivezić, A.J. Connolly, J.T. Vanderplas, and A. Gray. *Statistics, Data Mining and Machine Learning in Astronomy*. Princeton University Press, 2014.

[46] Jensen, C.S. et. al. Temporal data management. *IEEE TKDE*, pages 36–44, 1999.

[47] H.G. Kang and C.W Chung. Exploiting versions for on-line data warehouse maintenance in molap servers. In *Proc. of the 28th VLDB Conf.*, pages 742–753, 2002.

[48] Y. Kwon et al. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *SSDBM*, 2010.

[49] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proc. of SOCC Symp.*, June 2010.

[50] Lacey et. al. Merger rates in hierarchical models of galaxy formation - part two - comparison with n-body simulations. *Monthly Notices of the Royal Astronomical Society (mnras)*, 271:676–+, December 1994.

[51] 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 *Proceedings of the Workshop on Interfaces and Architectures for Scientific Data Storage*, 2009.

[52] Lomet, D. et. al. Transaction time support inside a database engine. In *Proc. of the 22nd ICDE Conf.*, 2006.

[53] Y. Low et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In *Proc. of the 38th Int. Conf. on Very Large DataBases (VLDB)*, pages 716–727, 2012.

[54] Large Synoptic Survey Telescope. `http://www.lsst.org/`.

[55] Lukaszuk et. al. Efficient high-dimensional indexing by superimposing space-partitioning schemes. In *Proc. of the 8th IDEAS Symp.*, pages 257–264, 2004.

[56] Apache mahout. `http://mahout.apache.org/`.

[57] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *Proc. of the SIGMOD Conf.*, pages 135–146, 2010.

[58] D. Marcos, A. J. Connolly, K. S. Krughoff, I. Smith, and S. C. Wallace. ASCOT: A Collaborative Platform for the Virtual Observatory. In *Astronomical Data Analysis Software and Systems XXI*, volume 461 of *Astronomical Society of the Pacific Conference Series*, page 901, 2012.

[59] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Proc. of the Sixth Biennial Conf. on Innovative Data Systems Research (CIDR)*, 2013.

[60] S.R. Mihaylov et al. REX: Recursive, delta-based data-centric computation. In *Proc. of the 38th Int. Conf. on Very Large DataBases (VLDB)*, 2012.

[61] Moon et. al. Scalability analysis of declustering methods for multidimensional range queries. *IEEE TKDE*, 10(2):310–327, 1998.

[62] M. Moyers, E. Soroush, S.C. Wallace, S. Krughoff, J. Vanderplas, M. Balazinska, and A Connolly. A demonstration of iterative parallel array processing in support of telescope image analysis. In *Proc. of the 39th Int. Conf. on Very Large DataBases (VLDB)*, 2013.

[63] Munro, J.I. et. al. Deterministic skip lists. In *SODA '92*, 1992.

[64] P. Neophytou, R. Gheorghiu, R. Hachey, T. Luciani, D. Bao, A. Labrinidis, E. G. Marai, and P. K. Chrysanthis. Astroshelf: understanding the universe through scalable navigation of a galaxy of annotations. In *Proc. of the SIGMOD Conf.*, pages 713–716, 2012.

[65] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. Global arrays: a nonuniform memory access programming model for high-performance computers. *J. Supercomput.*, June 1996.

[66] Nieto-santisteban et. al. Cross-matching very large datasets. In *National Science and Technology Council(NSTC) NASA Conference*, 2006.

[67] National Oceanic and Atmospheric Administration. `http://nomads.ncdc.noaa.gov/`.

[68] R.W. Numrich and J Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, August 1998.

[69] Travis E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[70] C. Olston et al. Pig Latin: a not-so-foreign language for data processing. In *Proc. of the SIGMOD Conf.*, pages 1099–1110, 2008.

[71] Oracle Flashback Technology. (2005). `http://www.oracle.com/technology/deploy/availability/htdocs/Flashback_Overview.htm`.

[72] Oracle OLAP. http://www.oracle.com/technetwork/database/options/olap/index.html.

[73] Orlandic et. al. The design of a retrieval technique for high-dimensional data on tertiary storage. *SIGMOD Record*, 31(2):15–21, 2002.

[74] Otoo et. al. Optimal chunking of large multidimensional arrays for data warehousing. In *Proc. of the 10th DOLAP Conf.*, pages 25–32, 2007.

[75] Ozsoyoglu, G. et. al. Temporal and real-time databases: A survey. *IEEE TKDE*, pages 513–532, 1995.

[76] Palo. http://www.palo.net/.

[77] Pedersen et. al. Multidimensional database technology. *IEEE Computer*, 34(12):40–46, 2001.

[78] Fernando Pérez and Brian E. Granger. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29, May 2007.

[79] Phillip Cudre-Maroux et. al. SS-DB: A Standard Science DBMS Benchmark. http://www-conf.slac.stanford.edu/xldb10/docs/ssdb_benchmark.pdf, 2010.

[80] Vijayshankar Raman and Joseph M. Hellerstein. Partial results for online query processing. In *Proc. of the SIGMOD Conf.*, June 2002.

[81] Ratko et. al. A class of region-preserving space transformations for indexing high-dimensional data. *Journal of Computer Science*, 1:89–97, 2005.

[82] Reiner et. al. Hierarchical storage support and management for large-scale multidimensional array database management systems. In *13th. DEXA Conf.*, pages 689–700, 2002.

[83] [x]:anonymized due to double-blind requirements.

[84] The NetCDF Users' Guide. http://www.unidata.ucar.edu/packages/netcdf/guidec/.

[85] Rew et. al. Data management: Netcdf: an interface for scientific data access. *IEEE Comput. Graph. Appl.*, 10(4):76–82, 1990.

[86] John T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proc. of the SIGMOD Conf.*, pages 10–18, 1981.

[87] J. Rogers et al. Overview of SciDB: Large scale array storage, processing and analysis. In *Proc. of the SIGMOD Conf.*, 2010.

[88] Sarawagi et. al. Efficient organization of large multidimensional arrays. In *Proc. of the 10th ICDE Conf.*, pages 328–336, 1994.

[89] SciDB Guide. `http://scidb.org/HTMLmanual/13.3/scidb_ug/`.

[90] Scidb-py. `http://jakevdp.github.io/SciDB-py/tutorial.html`.

[91] Seamons et. al. Physical schemas for large multidimensional arrays in scientific computing applications. In *Proc of 7th SSDBM*, pages 218–227, 1994.

[92] Seering, A. et. al. Efficient versioning for scientific array databases. In *Proc. of the 28th Int. Conf. on Data Engineering (ICDE)*, 2012.

[93] M. Shaw, B. Howe, P. Koutris, and D. Suciu. Optimizing large-scale semi-naive Datalog evaluation in Hadoop. In *Proceedings of the Workshop on Datalog 2.0*, 2012.

[94] Shimada et. al. A storage scheme for multidimensional data alleviating dimension dependency. In *ICDIM*, pages 662–668, 2008.

[95] Sloan Digital Sky Survey. `http://cas.sdss.org`.

[96] R. Snodgrass and I. Ahn. A taxonomy of time databases. In *Proc. of the SIGMOD Conf.*, pages 236–246, 1985.

[97] Emad Soroush and Magdalena Balazinska. Hybrid merge/overlap execution technique for parallel array processing. In *Workshop on Array Databases in conjunction with EDBT*, March 2011.

[98] Emad Soroush and Magdalena Balazinska. Time travel in a scientific array database. In *Proc. of the 29th Int. Conf. on Data Engineering (ICDE)*, March 2013.

[99] Emad Soroush, Magdalena Balazinska, and Daniel Wang. ArrayStore: A storage manager for complex parallel array processing. In *Proc. of the SIGMOD Conf.*, pages 253–264, June 2011.

[100] M. Stonebraker. The design of the postgres storage system. In *Proc. of the 13th VLDB Conf.*, pages 289–300, 1987.

[101] Michael Stonebraker, Jacek Becla, David Dewitt, Kian tat Lim, and Stan Zdonik. Requirements for science data bases and scidb. In *Proc. of the Fourth CIDR Conf.*, 2009.

[102] Mike Stonebraker, Jacek Becla, David DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stan Zdonik. Requirements for science data bases and SciDB. In *Fourth CIDR Conf. (perspectives)*, 2009.

[103] Tableau. http://tableausoftware.com.

[104] `http://upc.lbl.gov/`.

[105] Alex R. van Ballegooij. Ram: A multidimensional array dbms. In *Proceedings of the 2004 International Conference on Current Trends in Database Technology*, pages 154–165, 2004.

[106] Jacob VanderPlas, Emad Soroush, K. Simon Krughoff, and Magdalena Balazinska. Squeezing a big orange into little boxes: The ascotdb system for parallel processing of data on a sphere. *IEEE Data Eng. Bull.*, pages 11–20, 2013.

[107] Yu et. al. Distributed aggregation for data-parallel computing: interfaces and implementations. In *Proc. of the 22st SOSP*, 2009.

[108] M. Zaharia et al. Spark: cluster computing with working sets. In *2nd Workshop on Hot Topics in Cloud Computing (HotCloud'10)*, 2010.

[109] Y. Zhang et al. PrIter: a distributed framework for prioritized iterative computations. In *Proc. of the 37th Int. Conf. on Very Large DataBases (VLDB)*, pages 13:1–13:14, 2011.

[110] Zhang et. al. RIOT: I/O-efficient numerical computing without SQL. In *Proc. of the Fourth CIDR Conf.*, 2009.