ArrayStore: A Storage Manager for Complex Parallel Array Processing

Emad Soroush, Magdalena Balazinska
Computer Science Department
University of Washington
Seattle, USA
{soroush, magda}@cs.washington.edu

Daniel Wang
SLAC National Accelerator Laboratory
Menlo Park, CA
danielw@slac.stanford.edu

ABSTRACT

We present the design, implementation, and evaluation of ArrayStore, a new storage manager for complex, parallel array processing. ArrayStore builds on prior work in the area of multidimensional data storage, but considers the new problem of supporting a *parallel* and more *varied* workload comprising not only range-queries, but also binary operations such as joins and complex user-defined functions.

This paper makes two key contributions. First, it examines several existing single-site storage management strategies and array partitioning strategies to identify which combination is best suited for the array-processing workload above. Second, it develops a new and efficient storage-management mechanism that enables parallel processing of operations that must access data from adjacent partitions.

We evaluate ArrayStore on over 80GB of real data from two scientific domains and real operators used in these domains. We show that ArrayStore outperforms previously proposed storage management strategies in the context of its diverse target workload.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management−−−Systems; H.2.8 [**Information Systems**]: Database Management−−−Database applications

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

Scientists today are able to generate data at unprecedented scale and rate [18, 23]. To support these growing data management needs many advocate that one should move away from the relational model and adopt a multidimensional array data model [14, 40]. The main reason is that scientists typically work with array data and simulating arrays on top of relations can be highly inefficient [40]. Scientists also need to perform array-specific operations such as feature extraction [19], smoothing [35], and cross-matching [26], which are not built-in operations in relational DBMSs. As a result, many engines are being built today to support multidimensional arrays [4, 14, 35, 42]. To handle today's large-scale datasets, arrays must also be partitioned and processed in a shared-nothing cluster [35].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

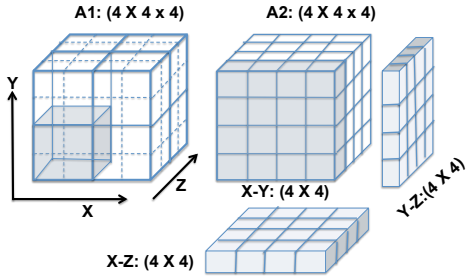


Figure 1: (1) The 4x4x4 array A1 is divided into eight 2x2x2 chunks. Each chunk is a unit of I/O (a disk block or larger). Each X-Y, X-Z, or Y-Z slice needs to load 4 I/O units. (2) Array A2 is laid out linearly through nested traversal of its axes without chunking. X-Y needs to load only one I/O unit,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

In this paper, 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 [4, 11, 14, 42], 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 paper, 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). A standard approach to storing an array is to partition it into sub-arrays called chunks [36] as illustrated in Figure 1. Each chunk is typically the size of a storage block. Chunking an array helps alleviate "dimension dependency" [38], 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 pro-

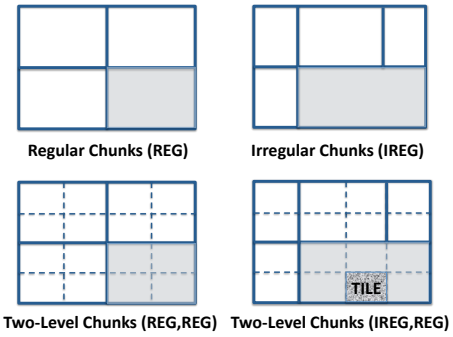


Figure 2: Four different chunking strategies applied to the same rectangular array. Solid lines shows chunk boundaries of the logical array (sample chunks shaded). Inner-level tiles are represented by dashed lines (one tile is textured).

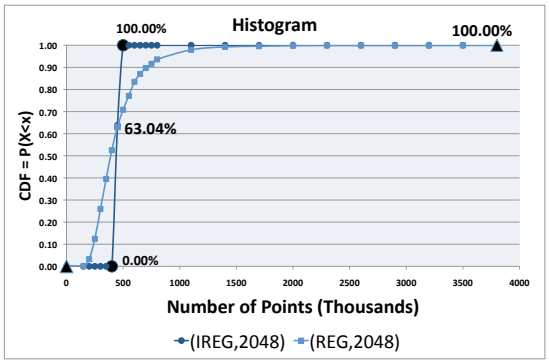


Figure 3: Cumulative distribution function of number of points (*i.e.*, non-null cells) per chunk for regular (REG) and irregular (IREG) chunking in astronomy simulation snapshot S92. Both strategies use 2048 chunks. Large circles for IREG and large triangles for REG mark the 0% and 100% points in each distribution.

cessing, 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, and other "lookup-style" operations [7, 8, 16, 24, 25, 28, 29, 34, 36, 38]. In contrast, our goal is to support a more varied workload as required by the science community [40]. 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 [8, 9, 35]), (2) array scans (*e.g.*, filters, regrids [40], and other operations that process an entire array), (3) binary array operations (*e.g.*, joins, cross-match [26]), and (4) operations that need to access data from adjacent partitions during parallel processing (*e.g.*, canopy clustering [1]). 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 [16]. In contrast, user-defined functions may incur overhead when chunks are too small and processed in parallel [19] 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 [12, 19, 20], 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 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. In this paper, 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 this paper 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 3.1 through 3.3). As in prior work, ArrayStore breaks arrays into multidimensional chunks, although we consider much larger chunks 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: 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 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 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 first technique leverages directly our two-level REG-REG storage layout to

[1]In this paper, 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.

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 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 [22] (3D data). The second dataset is the output of a flow cytometer from the oceanography domain [3] (6D data). For the first question, 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, 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. PROBLEM STATEMENT

We start with a more formal problem statement. We define an array similarly to Furtado and Baumann [16]: 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 1, which we formalize by presenting one or more representative operators for each type of operations. We use these operators throughout the paper.

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* [40], 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 this paper, 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 [35], 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 [26] 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 3.4 and 4.

Non-requirements. Due to space constraints, in this paper, 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 latter elsewhere [39].

3. ArrayStore STORAGE MANAGER

In this section, we present the design of ArrayStore.

3.1 Basic Array Chunking

As in prior work on storage management for multidimensional data (see Section 6), 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* [13, 16], 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 [8, 15, 29, 36]. Figure 2 illustrates this approach.

Irregular Chunks (IREG). Several schemes have also been proposed where an array is fragmented in a less regular fashion [16, 9]. In this paper, we call all such strategies *irregular* chunking schemes and illustrate them in Figure 2. Irregular chunking can speed-up range-selection queries when the chunk size and shape is tuned to the workload [16]. 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 [20] as shown in Figure 3.[2] One approach that has been proposed for creating such chunks [20] is to use a kd-tree [6], 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 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. 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.

Two-level chunking has been studied before [33, 38], 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 4), by leveraging the two-level structure to efficiently support overlap processing (see Section 3.4), and by exploring the regular-regular (*REG-REG*) approach as an alternative to IREG-REG. Through experiments (see Section 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]For dense arrays, this approach is identical to regular chunks.

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 3.4). For irregular chunking, a chunk index is stored in a separate file. In this paper, we do not study how chunk layout on disk affects performance as it mostly matters for dicing queries [38]. 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.

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.

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 [1] 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 4

Canopy clustering is a fast clustering method typically used to create preliminary clusters that are then further processed by more sophisticated algorithms [1]. 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 [19].

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 [1, 19, 20]. For canopy clustering, this second step clusters canopy centers found in individual partitions and assigns points to these final canopies [1]. Such a post-processing phase, however, can add significant overhead as we show in Section 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 [35, 37], 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 optimization below.

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 ← all chunks overlapping overlap_region.
5: tileSet ← ∅
6: for all Chunk ochunk_i in ochunkSet − core_chunk do
7:     Load ochunk_i into memory.
8:     tis ← all tiles in ochunk_i overlapping overlap_region.
9:     tileSet ← tileset ∪ tis
10: end for
11: Combine tilesSet into one chunk result_chunk.
12: return result_chunk.
```

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 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 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.

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 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

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 for**
11: **return** *result_chunk*.

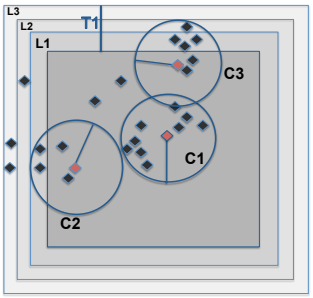


Figure 4: Example of multi-layer overlap used during canopy clustering. $C2$ necessitates that the operator loads a small amount of overlap data denoted with $L1$. $C3$, however, requires an additional overlap layer. So $L2$ is also loaded.

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 Secion 5.3.

4. ACCESS METHOD

ArrayStore provides a single access method that supports the various operator types presented in Section 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 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

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

Table 1: Access Method API

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 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. We illustrate ArrayStore's access method by showing how several representative operators (from Section 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()` 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, we consider a structural join [35] 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

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 while
15: return result_array
```

co-located. As a result different array partitioning strategies yield different performance results for join (see Section 5).

Canopy Clustering. We described the canopy clustering algorithm in Section 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 [21]. 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.

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 comprises two snapshots, $S43$ and $S92$, from a large-scale astronomy simulation [22] 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. We store each snapshot 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 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 the output of a flow cytometer [3]. 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. Join queries thus run on 14 GB of 6D data.

Table 2 shows the naming convention for the experimental setups. ArrayStore's best-performing strategy is highlighted

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 5 and 6 show the results. In both figures, the y-axis is the total query processing time.

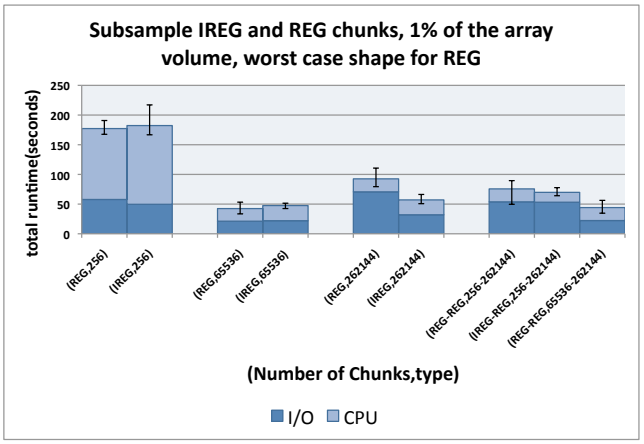
Array dicing query. Figure 5(a) shows the results of a range selection query, when the selected region is a 3D rectangular slice of $S92$ (we observe the same trend in $S43$). Each bar shows the average of 10 runs. The error bars show the minimum and maximum runtimes. In each run, we randomly select 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 [38]) 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

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: Naming convention used in experiments.



(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)
(REG,4096)	28	115
(REG,262144)	46	51
(REG,2097152)	90	66
(REG-REG,4096-2097152)	28	64

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

Figure 5: Array dicing query on 3D and 6D datasets.

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

We first consider the first three bars in Figure 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 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.

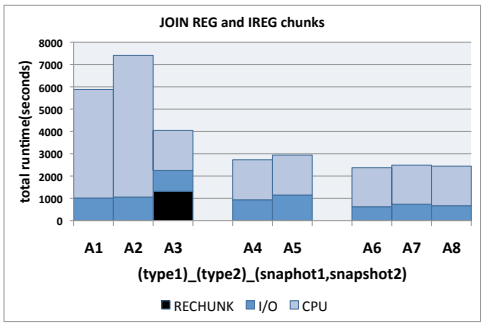
Interestingly, the overhead of chunk misalignment (always occurring with IREG and occurring in some REG configurations as discussed above) can rapidly grow with array dimensionality. The processing time of non-aligned 3D arrays is $3.5X$ that of aligned ones, while the factor is $6X$ for 6D arrays (Figure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

Finally, the last three bars in Figure 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.

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.

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.



(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.

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

Parallel Selection. Figure 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 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.

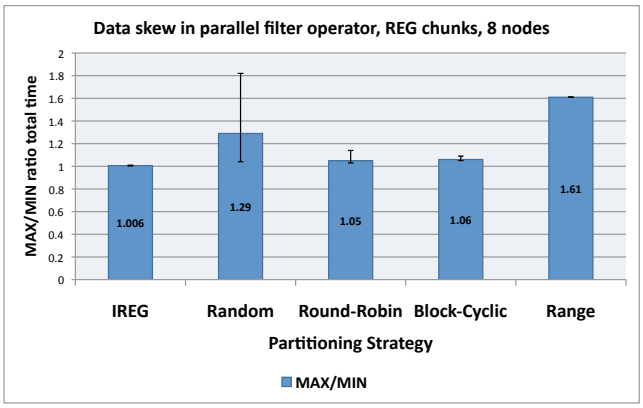


Figure 7: Parallel selection on 8 nodes with different partitioning strategies on REG chunks. We vary chunk sizes from 140 MB to 140 KB. Error bars show the variation of MAX/MIN runtime ratios in that range of chunk sizes. Round-Robin and Block-Cyclic have the lowest skew and variance. Results for these strategies are similar to those of IREG.

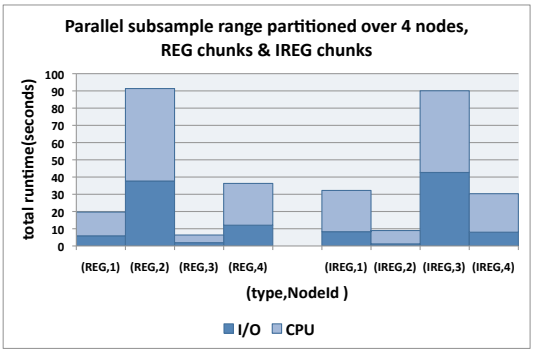


Figure 8: Parallel subsample with REG or IREG chunks distributed using range partitioning across 4 nodes. Subsample runs only on a few nodes causing skew, independent of the chunking scheme chosen.

We measure a MAX/MIN ratio of just 1.11 with 4 nodes and 65536 chunks with a std deviation of 0.036 (figure not shown due to space constraints).

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 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 par-

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 required.		
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 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 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.

titioning, however, exhibits skew in the "local join phase" (Table 4), leaving block-cyclic as the winning approach.

Table 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: 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.

5.3 Performance of Overlap Storage

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

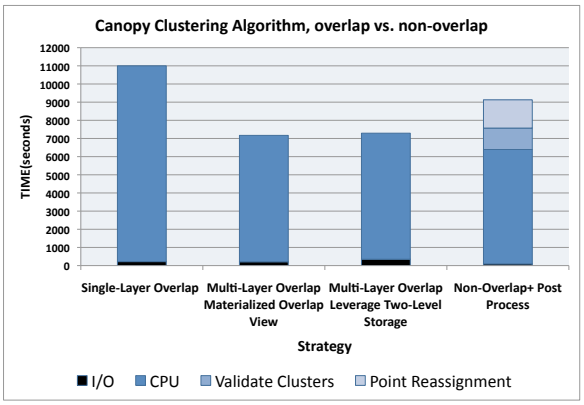


Figure 9: Canopy Clustering algorithm with or without overlap on the 3D dataset. Single-layer overlap does not perform well because of a large maximum overlap-size chosen. Both multi-layer overlap techniques outperforms no-overlap by 25%.

Figure 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 [1]. 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 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 10 shows the results from the volume-density application described in Section 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 which is not shown due to space constraints.

In this experiment, materialized views also outperform multi-layer using the two-level storage because views can

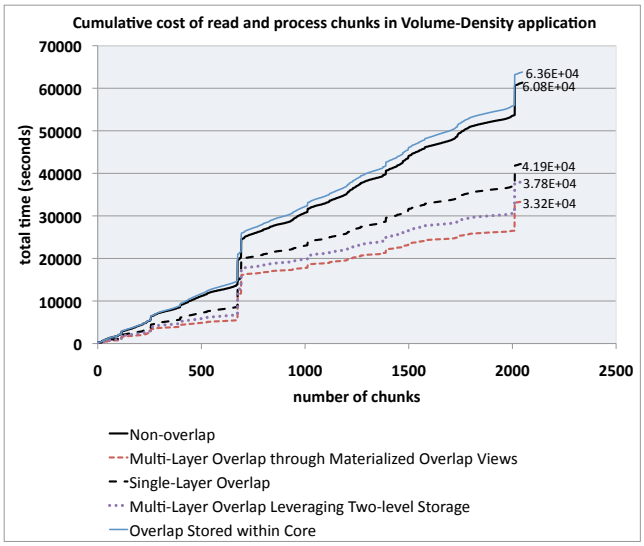


Figure 10: Volume-density application on 3D dataset with and without overlap. Multi-layer overlap outperforms no-overlap by almost $2X$.

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 [35, 37]. 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 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).

6. RELATED WORK

MOLAP systems store data in multidimensional arrays [31, 41], but they focus on unary aggregation and selection queries, while 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 [27], Cognos PowerPlay [10], and others to analyze large datasets. To the best of our knowledge, Palo [30] 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.

Many existing array-processing systems [8, 13, 15, 35] use regular tiling for data storage. Others support user-defined irregular tiles [9]. 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 [8] and across disks [8, 25] for range-selection queries.

RasDaMan [14] is a multidimensional array DBMS implemented on top of a relational DBMS. Furtado and Baumann [16] 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 ours since they find that arbitrary tiling tuned to a specific workload outperforms regular tiling. Reiner *et al.* [33] studied hierarchical storage support for large-scale multi-dimensional arrays in RasDaMan. Their approach is analogous to the two-level, IREG-REG, chunking strategy. However, their study is constrained to range-selection queries.

Shimada *et al.* [38] 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. 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 [29, 36]. This work is orthogonal to ours since we comparatively study regular *v.s.* irregular *v.s.* two-level chunking schemes and support for overlap data.

Seamons and Winslett [37] 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 this paper. 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 [35].

There exist many data structures for indexing multi-dimensional data including the R-Tree [17] and its variants [2, 5], the KD-Tree [6], the KDB-Tree [34], the GammaSLK [24], the Pyramid technique [7], the Gamma strategy [28], RPST [32], 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, we study storage management techniques for more varied array operations.

7. 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. It does not consider iterative operations nor array updates, which may be poorly served by our regular-tiled,

read-only store. 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.

8. ACKNOWLEDGEMENTS

The astronomy simulation dataset was graciously supplied by T. Quinn and F. Governato of the UW Dept. of Astronomy. We thank T. Quinn, J. P. Garner and Y. Kwon for their help with the data, user-defined functions, and comments on drafts of this paper. We thank the SciDB team for insightful discussions and the anonymous reviewers for their comments. This work is partially supported by NSF CDI grant OIA-1028195, gifts from Microsoft Research, and Balazinska's Microsoft Research Faculty Fellowship.

9. REFERENCES

[1] `http://mahout.apache.org/`.

[2] 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.

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

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

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

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

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

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

[9] Chang et. al. T2: a customizable parallel database for multi-dimensional data. *SIGMOD Record*, 27(1):58–66, 1998.

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

[11] Cohen et. al. MAD skills: new analysis practices for big data. *PVLDB*, 2(2):1481–1492, 2009.

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

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

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

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

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

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

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

[19] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proc. of SOCC Symp.*, June 2010.

[20] Y. Kwon, D. Nunley, J.P. Gardner, M. Balazinska, B. Howe, and S. Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *Proc of 22nd SSDBM*, 2010.

[21] 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.

[22] Loebman et. al. 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 (IASDS)*, 2009.

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

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

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

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

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

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

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

[30] Palo. `http://www.palo.net/`.

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

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

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

[34] 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.

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

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

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

[38] Shimada et. al. A storage scheme for multidimensional data alleviating dimension dependency. In *Proc. of the 3rd ICDIM Conf.*, pages 662–668, 2008.

[39] E. Soroush and M. Balazinska. Hybrid merge/overlap execution technique for parallel array processing. In *1st Workshop on Array Databases (AD2011)*, 2011.

[40] Stonebraker et. al. Requirements for science data bases and SciDB. In *Fourth CIDR Conf. (perspectives)*, 2009.

[41] Tsuji et. al. An extendible multidimensional array system for MOLAP. In *Proc. of the 21st SAC Symp.*, pages 503–510, 2006.

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