The TileDB Array Data Storage Manager

Stavros Papadopoulos* Intel Labs & MIT
Kushal Datta‡ Intel Corporation
Samuel Madden* MIT
Timothy Mattson‡ Intel Labs

*{stavrosp, madden}@csail.mit.edu
‡{kushal.datta, timothy.g.mattson}@intel.com

ABSTRACT

We present a novel storage manager for *multi-dimensional arrays* that arise in scientific applications, which is part of a larger scientific data management system called TileDB. In contrast to existing solutions, TileDB is optimized for *both* dense and sparse arrays. Its key idea is to organize array elements into ordered collections called *fragments*. Each fragment is dense or sparse, and groups contiguous array elements into *data tiles* of fixed capacity. The organization into fragments turns random writes into sequential writes, and, coupled with a novel read algorithm, leads to very efficient reads. TileDB enables parallelization via multi-threading and multi-processing, offering thread-/process-safety and atomicity via lightweight locking. We show that TileDB delivers comparable performance to the HDF5 dense array storage manager, while providing much faster random writes. We also show that TileDB offers substantially faster reads and writes than the SciDB array database system with both dense and sparse arrays. Finally, we demonstrate that TileDB is considerably faster than adaptations of the Vertica relational column-store for dense array storage management, and at least as fast for the case of sparse arrays.

1. INTRODUCTION

Many scientific and engineering fields generate enormous amounts of data through measurement, simulation, and experimentation. Examples of data produced in such fields include astronomical images, DNA sequences, geo-locations, social relationships, and so on. All of these are naturally represented as *multi-dimensional arrays*, which can be either *dense* (when every array element has a value) or *sparse* (when the majority of the array elements are empty, i.e., zero or *null*). For instance, an astronomical image can be represented by a dense 2D array, where each element corresponds to a pixel. Geo-locations (i.e., points in a 2D or 3D coordinate space) can be represented by non-empty elements in a sparse 2D or 3D array that models the coordinate space.

Scientific array data can be very large, containing billions of non-null values that do not readily fit into the memory of a single or even multiple machines. As a result, many applications need to read and write both individual (random) elements as well as large sequential extents of these arrays to and from the disk. Simply storing arrays as files forces application programmers to handle many issues, including array representation on disk (i.e., sparse vs. dense layouts), compression, parallel access, and performance. Alternatively, these issues can be handled by optimized, special-purpose *array data storage management* systems, which perform complex analytics on scientific data. Central to such systems are efficient data access primitives to read and write arrays. These primitives are the focus of this work.

1.1 Existing Array Management Systems

A number of others systems and libraries for managing and accessing arrays exist. HDF5 [16] is a well-known array data storage manager. It is a dense array format, coupled with a C library for performing the storage management tasks. Several scientific computing packages integrate HDF5 as the core storage manager (such as NetCDF-4 [9], h5py [6] and PyTables [13]). HDF5 groups array elements into regular hyper-rectangles, called *chunks*, which are stored on the disk in binary format in a single large file. HDF5 suffers from two main shortcomings. First, it does not efficiently capture sparse arrays. A typical approach is to represent denser regions of a sparse array as separate dense arrays, and store them into a (dense) HDF5 array of arrays. This requires enormous manual labor to identify dense regions and track them as they change. Second, HDF5 is optimized for *in-place* writes of large blocks. In-place updates result in poor performance when writing small blocks of elements that are randomly distributed, due to the expensive random disk accesses they incur. Parallel HDF5 (PHDF5) is a parallel version of HDF5 with some additional limitations: (i) it does not allow concurrent writes to compressed data, (ii) it does not support variable-length element values, and (iii) operation atomicity requires some coding effort from the user, and imposes extra ordering semantics [4].

Proceedings of the VLDB Endowment, Vol. 10, No. 4

Random writes are important in many applications. For instance, random writes occur with large OLAP cube applications [1] and selective image processing (e.g., applying blur and de-noising filters). In dense linear algebra, while traditional algorithms [15] are designed around regular updates of large blocks, more recent algorithms based on DAG scheduling [11] often result in small block updates irregularly distributed throughout the matrix. When combined with Asynchronous Many-Task runtime systems [3, 7] for extreme scale systems, this irregularity requires an array storage manager optimized for random updates of small blocks.

Array-oriented databases implement their own storage managers and, thus, they can arguably serve as the storage layer for other scientific applications built on top (see [24] for a survey of these systems). SciDB [18] is a popular array database, which however also has significant storage management limitations. It is not truly designed for sparse arrays as it relies on regular dimensional chunking (similar to HDF5). It also requires reading an entire chunk even if a small portion of it is requested, and updating an entire chunk even when a single element changes, leading to poor performance in the case of random writes. ArrayStore [25] propose optimizations for the sparse case on top of SciDB, including grouping underutilized chunks into bigger ones to address imbalance. However, the original chunks still remain the atomic units of processing, thus many of the above problems persist.

Alternatively, relational databases (e.g., MonetDB[20] or Vertica [21]) have been used as the storage backend for array management (e.g., in RAM [26] for dense and SRAM [19] for sparse arrays), storing non-empty elements as records and explicitly encoding the element indices as extra table columns. Subarray queries (used to retrieve any part of the array) are performed via optimized `SELECT WHERE` SQL queries on the array indices. This explicit index encoding leads to poor performance in the case of dense arrays. Similar to RAM and SRAM, RasDaMan [17] stores array data as binary large objects (BLOBs) in PostgreSQL [12], and enables query processing via a SQL-based array query language called RasQL. Like RAM and SRAM, array storage and access has to go through PostgreSQL, which is typically quite slow. Moreover, RasDaMan is designed mainly for dense arrays; its support for sparse arrays relies on the user manually decomposing the array space into irregular regions of approximately equal number of non-empty elements, which is cumbersome (similar to the case of HDF5).

A second concern with the database approach (for some users) is that databases run as a separate server that is accessed via SQL queries over a ODBC/JDBC interface. In contrast, programmers of high-performance scientific and image processing applications who make heavy use of arrays usually want API-based access to directly write and read array elements; they either prefer libraries like HDF5 for dense data, or develop their own special-purpose tools (such as GenericIO [5], designed to manage sparse 3D particles).

1.2 Our Approach

In this paper we introduce the first array storage manager that is optimized for *both* dense and sparse arrays, and which lies at the core of a new scientific data management system called TileDB. TileDB is an open-source software system written in C++ and maintained by Intel Labs and MIT. More information and code can be found at `http://www.tiledb.org`. TileDB is already in use by the Broad Institute, one of the largest genomics institutes in the world, for storing many Terabytes of genomics data, which are modeled as a huge sparse 2D array [2]. The focus of this paper is on TileDB's storage module (we defer a discussion of TileDB's distributed architecture and complex analytics engine to follow-up work). Throughout the rest of the paper, we use *TileDB* to refer to the storage manager component of the larger TileDB system.

TileDB exposes a C library for efficient writes and reads to arrays using a novel dense and sparse on-disk representation, while supporting important features such as compression, parallelism, and more. Contrary to specialized software like GenericIO, TileDB handles arrays of arbitrary dimensionality and schema and, thus, can be readily used in a wide range of large-scale scientific applications. The key idea of TileDB is that it organizes array elements into *ordered* collections called *fragments*. Fragments may overlap in the index space of the array and are used to represent batches of updates to the array. They may also have either a dense or a sparse representation. Dense fragments group their elements into regular-sized chunks in the index space, which we call *data tiles*. A sparse fragment represents an element by explicitly storing its indices in a specific, global order. Sparse elements are grouped into data tiles of *fixed* element capacity (in contrast to SciDB), which balances I/O cost when accessing the array.

The concept of fragments dramatically improves write performance as writes are performed in *batches*, and each batch (which may consist of many small, random writes) is written to a separate fragment *sequentially*. Most importantly, sparse fragments can be used to speed up random writes even in dense arrays. In contrast to HDF5's in-place updates, TileDB batches random requests and sequentially writes many at a time as a sparse fragment. This approach also enables efficient representation and update of *variable-length* values (such as strings).

Turning random-writes into sequential appends is a classic technique in storage systems [23], but implementing it in an array storage manager is non-trivial. This is because multiple fragments may cover the same logical region of an array, and some fragments may be sparse, which makes it hard for the read algorithm to find the most recent fragment that updated each returned element. TileDB implements an efficient read algorithm that does this; it also avoids unnecessary tile reads when a portion of a fragment is totally covered by a newer fragment. Finally, as the number of fragments grows and read performance may degrade, TileDB employs a *consolidation* algorithm that merges multiple fragments into a single one. Consolidation can be performed in the background while other concurrent reads and writes are performed.

TileDB supports parallelism through both multi-threading (pthreads and OpenMP) and multi-processing (e.g., via forks and MPI). It provides thread-/process-safety with lightweight locking. Moreover, it guarantees operation atomicity, without imposing any additional ordering constraints (this is in contrast to parallel implementations of HDF5, like PHDF5 [10]).

The key contributions of this paper are the following:

- We describe the TileDB storage manager designed to support both sparse and dense arrays. TileDB uses tiles of fixed element capacity, making efficient use of the I/O subsystem. It also employs a fragment-based layout, which makes all writes sequential, and implements protocols for efficiently reading elements in the presence of multiple fragments, as well as a consolidation algorithm for compacting many fragments into one.

- We describe how these features allow TileDB to provide desirable properties for parallel programming, such as thread- and process-safety, as well as atomicity of concurrent reads and writes to a single array.

- We conduct a thorough experimental evaluation. We show that, for dense arrays, TileDB delivers comparable performance to HDF5, while being orders of magnitude faster on random element writes. We also demonstrate that TileDB offers orders of magnitude faster reads and writes than SciDB in most settings for both dense and sparse arrays. In addition, TileDB is 2x-40x faster than Vertica for dense arrays, and at least as fast as Vertica for sparse arrays. Finally, we confirm the effectiveness of our multi-fragment read and consolidation, and the fact that TileDB offers excellent scalability, both in terms of parallelism and when operating on large datasets.

2. TileDB OVERVIEW

Data Model. TileDB provides an array-oriented data model that is similar to that of SciDB [18]. An array consists of *dimensions* and *attributes*. Each dimension has a *domain*, and all dimension domains collectively orient the *logical space* of the array. A combination of dimension domain values, referred to as *coordinates*, uniquely identifies an array element, called *cell*. A cell may either be empty (*null*) or contain a tuple of attribute values. Each attribute is either a primitive type (`int`, `float`, or `char`), or a fixed or variable-sized vector of a primitive type. For instance, consider an attribute representing complex numbers. Each cell may then store two `float` values for this attribute, one for the real part and the other for the imaginary part. As another example, a string can be represented as a vector of characters.

Arrays in TileDB can be *dense*, where every cell contains an attribute tuple, or *sparse*, where some cells may be *empty*; all TileDB operations can be performed on either type of array, but the choice of dense vs. sparse can significantly impact application performance. Typically, an array is stored in sparse format if more than some threshold fraction of cells are empty, which highly depends on the application.

All dimension domains have the same type. For dense arrays, only `int` dimensions are supported; for sparse arrays `float` dimensions are also allowed. This is because TileDB materializes the coordinates of the non-empty cells for sparse arrays and, thus, can store them as real numbers (see Section 3). In other words, TileDB natively supports continuous multi-dimensional spaces, without the need for discretization. On the other hand, dense arrays have inherently discrete domains that are naturally represented as integers.

This is sufficient to capture many applications, as long as each data item can be uniquely identified by a set of coordinates. For example, in an imaging application, each image can be modeled by a 2D dense array, with each cell representing the RGB values for that pixel. Another application is Twitter, where geo-tagged tweets can be stored in a 2D sparse array whose cells are identified by `float` geographical coordinates, with each tweet stored in a variable-length `char` attribute.

Figure 1 shows two example arrays, one dense one sparse, which have two dimensions, `rows` and `columns`. Both dimensions have domain [1,4]. The arrays have two attributes, `a1` of type `int32` and `a2` of type variable `char`. Each cell is identified by its coordinates, i.e., a pair of `rows`, `columns`. In our examples, empty cells are shown in white, and non-empty cells in gray. In the dense array, every cell has an attribute tuple, e.g., cell (4,4) contains $\langle 15, \text{pppp} \rangle$, whereas several cells in the sparse array are empty, e.g., cell (4,1).

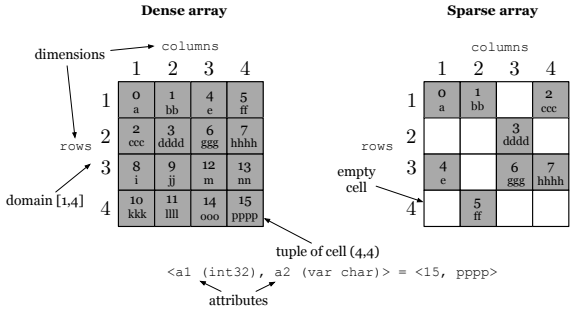


Figure 1: The logical array view

Global cell order. Whenever multi-dimensional data is stored to disk or memory it must be laid out in some linear order, since these storage media are single-dimensional. This choice of ordering can significantly impact application performance, since it affects which cells are near each other in storage. In TileDB, we call the mapping from multiple dimensions to a linear order the *global cell order*.

A desirable property is that cells that are accessed together should be *co-located* on the disk and in memory, to minimize disk seeks, page reads, and cache misses. The best choice of global cell order is dictated by application-specific characteristics; for example, if an application reads data a row-at-a-time, data should be laid out in rows. A columnar layout in this case will result in a massive number of additional page reads.

TileDB offers various ways to define the global cell order for an array, enabling the user to tailor the array storage to his or her application for maximum performance. For dense arrays, the global cell order is specified in three steps. First, the user decomposes the dimensional array domain into *space tiles* by specifying a *tile extent* per dimension (e.g., 2×2 tiles). This effectively creates equi-sized hyper-rectangles (i.e., each containing the same number of cells) that cover the entire logical space. Second, the user determines the cell order within each space tile, which can be either *row-major* or *column-major*. Third, the user determines a tile order, which is also either *row-major* or *column-major*. Figure 2 shows the global cell orders resulting from different choices in these three steps (the space tiles are depicted in blue).

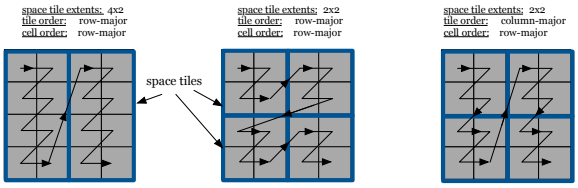


Figure 2: Global cell orders in dense arrays

The notion of a global cell order also applies to sparse arrays. However, creating sparse tiles is somewhat more complex because simply using tiles of fixed logical size could lead to many empty tiles. Even if we suppressed storage of empty tiles, skew in many sparse datasets would create tiles of highly varied capacity, leading to ineffective compression, bookkeeping overheads, and some very small tiles where seek times represent a large fraction of access cost. Therefore, to address the case of sparse tiles, we introduce the notion of *data tiles*.

Data tiles. A *data tile* is a group of *non-empty* cells. It is the atomic unit of compression and has a crucial role during reads (explained in Section 4.1). Similarly to a space tile, a data tile is enclosed in logical space by a hyper-rectangle. For dense arrays, each data tile has a one-to-one mapping to a space tile, i.e., it encompasses the cells included in the space tile.

For the sparse case, TileDB instead allows the user to specify a data tile *capacity*, and creates the data tiles such that they all have the same number of non-empty cells, equal to the capacity. To implement this, assuming that the fixed capacity is denoted by c, TileDB simply traverses the cells in the global cell order imposed by the space tiles and creates one data tile for every c non-empty cells. A data tile of a sparse array is represented in the logical space by the tightest hyper-rectangle that encompasses the non-empty cells it groups, called the *minimum bounding rectangle* (MBR). Figure 3 illustrates various data tiles resulting from different global cell orders, assuming that the tile capacity is 2. The space tiles are depicted in blue color and the (MBR of the) data tiles in black. Note that data tiles in the sparse case may overlap, but each non-empty cell corresponds to exactly one data tile.

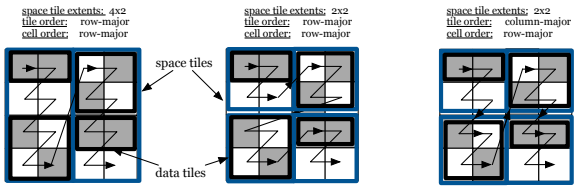


Figure 3: Data tiles in sparse arrays

Compression. TileDB employs tile-based compression. Additionally, it allows the user to select different compression schemes on a per-attribute basis, as attributes are stored separately (as discussed in Section 3). The API supports defining different compression schemes for each attribute, but currently only gzip is implemented. In the future, TileDB will be able to support more compression schemes, such as RLE, LZ, as well as user-defined schemes.

Fragments. A fragment is a *timestamped snapshot of a batch of array updates*, i.e., a collection of array modifications carried out via write operations and made visible at a particular time instant. For instance, the initial loading of the data into the array constitutes the first array fragment. If at a later time a set of cells is modified, then these cells constitute the second array fragment, and so on. In that sense, an array is comprised of a collection of array fragments, each of which can be regarded as a separate array, whose collective logical overlap composes the current logical view of the array.

A fragment can be either *dense* or *sparse*. Dense fragments are used only with dense arrays, but sparse fragments may be applied to both dense and sparse arrays.

Figure 4 shows an example of an array with three fragments; the first two are dense and the third is sparse. Observe that the second fragment is dense within a hyper-rectangular subarray, whereas any cell outside this subarray is empty. The figure also illustrates the collective logical view of the array; the cells of the most recent fragments *overwrite* those of the older ones.

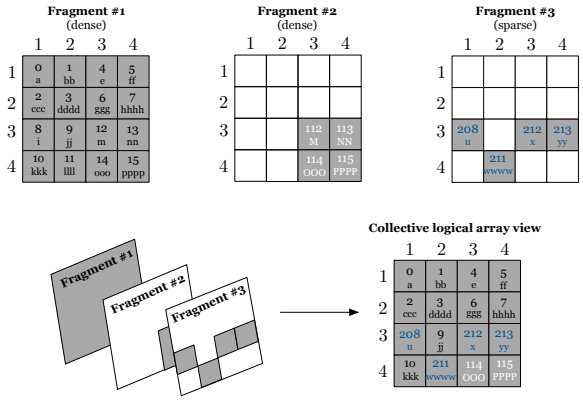


Figure 4: Fragment examples

Fragments are the key concept that enables TileDB to perform rapid writes. If they cover relatively disjoint subareas of the domain, or if their number is modest (e.g., hundreds to thousands as shown in our experiments in Section 6), then their presence does not significantly affect the read performance. If numerous fragments are created and the read performance becomes unsatisfactory, TileDB employs an efficient *consolidation* mechanism that coalesces fragments into a single one. Consolidation can happen in parallel in the background, while reads and writes continue to access the array. The concept of fragments and their benefits are explained in Section 4.

Array metadata. TileDB stores two types of metadata about an array: the *array schema* and the *fragment bookkeeping*. The former contains information about the definition of the array, such as its name, the number, names and types of dimensions and attributes, the dimension domain, the space tile extents, data tile capacity, and the compression types. The latter summarizes information about the physical organization of the stored array data in a fragment (explained in Section 3).

System architecture. The TileDB storage manager is exposed as a C API library to users. The system architecture is described in Figure 5. The basic array operations are `init`,

`write`, `read`, `consolidate` and `finalize`; these are thoroughly explained in Section 4. The storage manager keeps in-memory state for the *open arrays*, i.e., arrays that have been initialized and not yet finalized. This state keeps the bookkeeping metadata for each fragment of every open array in memory; this is shared between multiple threads that access the same array at the same time. Locks are used to mediate access to the state (covered in detail in Section 5). Each thread has its own array object, which encapsulates a *read state* and one fragment object with its own read and *write state*.

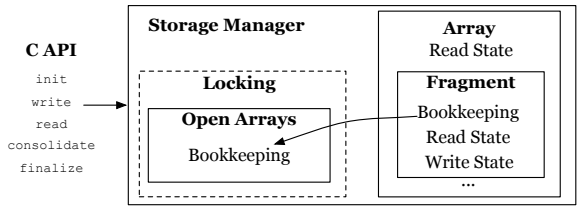


Figure 5: The TileDB storage manager architecture

3. PHYSICAL ORGANIZATION

An array in TileDB is physically stored as a directory in the underlying file system. The array directory contains one sub-directory per array fragment, plus a file storing the array schema in binary format. Every fragment directory contains one file per fixed-sized attribute, and two files per variable-sized attribute. The attribute files store cell values in the global cell order in a binary representation. Storing a file per attribute is a form of "vertical partitioning", and allows efficient subsetting of the attributes in read requests. Each fragment directory also contains a file that stores the bookkeeping metadata of the fragment in binary, compressed format. We explain the physical storage for dense and sparse arrays with examples below.

Figure 6 shows the physical organization of the dense array of Figure 1, assuming that it follows the global cell order illustrated in the middle array of Figure 2. The cell values along the fixed-sized attribute `a1` are stored in a file called `a1.tdb`, always following the specified global cell order. Attribute `a2` is variable-sized and, thus, TileDB uses two files to store its values. File `a2_var.tdb` stores the variable-sized cell values (serialized in the global cell order), whereas `a2.tdb` stores the *starting offsets* of each variable-sized cell value in file `a2_var.tdb`. This enables TileDB to efficiently locate the actual cell value in file `a2_var.tdb` during read requests. For instance, cell (2,3) appears 7th in the global cell order. Therefore, TileDB can efficiently look-up the 7th offset in file `a2.tdb` (since the offsets are fixed-sized), which points to the beginning of the cell value ggg in `a2_var.tdb`.

Using this organization, TileDB does not need to maintain any special bookkeeping information. It only needs to record the subarray in which the dense fragment is constrained, plus some extra information in the case of compression.

Figure 7 illustrates the physical organization of the sparse array of Figure 1, assuming that it follows the global cell order of the middle array of Figure 3. The attribute files of sparse arrays are organized in the same manner as those of dense,

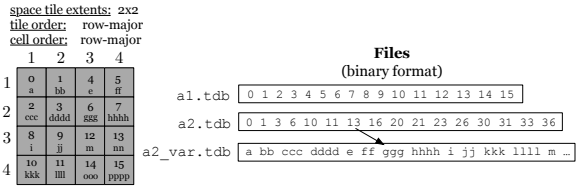


Figure 6: Physical organization of dense fragments

except that they contain only *non-empty cells*. That is, the cell values follow the global cell order in the attribute files, and there is an extra file for variable-sized attributes storing the starting offsets of the variable-sized cell values. Unlike dense arrays where the offset of each cell value can be directly computed, the same is not possible in sparse arrays, since the exact distribution of the non-empty cells is unknown. In order to locate the non-empty cells, TileDB stores an additional file with the explicit *coordinates* of the non-empty cells (`__coords.tdb` in the figure), which are again serialized in the global cell order. Note that the coordinates follow the order of the dimensions, as specified in the array schema.

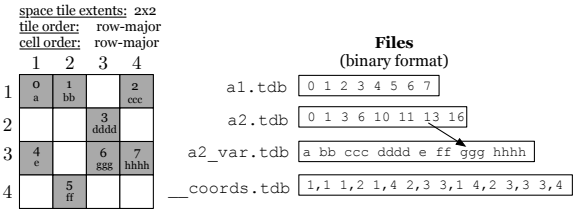


Figure 7: Physical organization of sparse fragments

In order to achieve efficient reads, TileDB stores two types of bookkeeping information about the data tiles of a sparse array. Recall that a data tile is a group of non-empty cells of fixed capacity, associated with an MBR. TileDB stores the MBRs of the data tiles in the bookkeeping metadata, which facilitate the search for non-empty cells during a read request. In addition, it stores *bounding coordinates*, which are the first and last cell of the data tile along in the global cell order. These are important for reads as well, as explained in Section 4.1.

A final remark on physical storage concerns *compression*. As mentioned earlier, TileDB compresses each attribute data tile separately, based on the compression type specified in the array schema for each attribute. In other words, for every attribute file, TileDB compresses the data corresponding to each tile separately prior to writing them into the file. Note that different tiles may be compressed into blocks of different size, even for fixed-sized attributes. TileDB maintains bookkeeping information in order to be able to properly locate and decompress each tile upon a read request.

4. OPERATIONS

In this section, we describe the core functions of TileDB.

4.1 Read

`Read` returns the values of any subset of attributes inside a user-supplied subarray. The result is sorted on the global

cell order within the subarray. The user specifies the subarray and attributes in the `init` call. During initialization, TileDB loads the bookkeeping data of each array fragment from the disk into main memory. This bookkeeping data is of negligible size (a few KB) in the dense case, whereas in the sparse case it depends on the tile capacity. For typical tile capacities of 10K-1M, the bookkeeping size is 4-6 orders of magnitude smaller than the total array size.

The user calls `read` one or more times with allocated buffers that will store the results, one per fixed-length attribute, two per variable-lengthed attribute, along with their sizes. TileDB populates the buffers with the results, following the global cell order within the specified subarray. For variable-sized attributes, the first buffer contains the starting cell offsets of the variable-sized cells in the second buffer. For variable-length attributes and sparse arrays, the result size may be unpredictable, or exceed the size of the available main memory. To address this issue, TileDB gracefully handles buffer overflow. If the read results exceed the size of some buffer, TileDB fills in as much data as it can into the buffer and returns. The user can then consume the current results, and resume the process by invoking `read` once again. TileDB maintains *read state* that captures the location where reading stopped.

The main challenge of reads is the presence of multiple fragments in the array. This is because the results must be returned in the global cell order, and `read` cannot simply search each fragment individually; if a recent fragment overwrites a cell of an older fragment, no data should be retrieved from the old fragment for this cell, but the old fragment should be used for adjacent cells. TileDB implements a read algorithm that allows it to efficiently access all fragments, skipping data that does not qualify for the result. The algorithm is slightly different for dense and sparse arrays, explained in turn below.

Dense arrays. The algorithm has two stages. The first computes a *sorted list* of tuples of the form $\langle [sc, ec], fid \rangle$; $[sc, ec]$ is a range of cells between start coordinates sc and end coordinates ec (inclusive), where sc precedes or is equal to ec in the global cell order, and fid is a fragment id. Each fragment is assigned a unique fid based on its timestamp, with larger fid values corresponding to more recent fragments (ties are broken arbitrarily based on the id of the thread or process that created the fragment). The following rules hold: (i) all ranges must be disjoint, (ii) the ranges must be sorted in the global cell order, (iii) the ranges in the ordered list must contain all and only the actual, *up-to-date* result cells, and (iv) the cells covered in each range must appear contiguously on the disk. Given this list, the second stage retrieves the actual attribute values from the respective fragment files, with a *single* file I/O into the user buffers for each range of each requested attribute. We focus the rest of this section on the first stage, as the second stage is simple (given sc and ec, it easy to calculate their offsets in the files via closed formulas).

The first stage proceeds iteratively, with each iteration operating on a single space tile. Space tiles are visited in the global order. For each tile T, the algorithm considers each fragment overlapping T. It computes a set of disjoint coordinate ranges $[sc, ec]$ that are in the query subarray within T, such that the cells in these ranges are *adjacent* in the global cell order. For instance, if the subarray covers the entire tile [1:2, 1:2] (i.e., the upper left tile) of our dense array examples, then a single range is created, $[(1,1),(2,2)]$. If the subarray covers only one row of this tile, say [1:1, 1:2], then the range $[(1,1),(1,2)]$ is created. However, if it covers a column, say [1:2, 1:1] (recall that our cell order is row-major), then two ranges are created, namely $[(1,1),(1,1)]$ and $[(2,1),(2,1)]$. This is because we cannot construct a single range such that $(1,1)$ and $(2,1)$ appear adjacent in the global cell order. In other words, the algorithm operates on whole qualifying ranges of cells stored *contiguously* on the disk, rather than individual cells, thus optimizing performance. For sparse fragments, similar ranges are created, but this time considering the *bounding coordinates* of each tile overlapping with the subarray. Note that these ranges are sparse, i.e., they may cover also empty cells.

Next, the algortihm creates one tuple $\langle [sc, ec], fid \rangle$ for each such range of fragment fid, and inserts them into a *priority queue* pq. The comparator of pq gives precedence to tuples with *smallest* sc value, breaking ties by giving precedence to the tuple with the *largest* fid. For each tuple, through fid we can know whether it comes from a dense or sparse fragment.

Once a priority queue has been built for T, the algorithm pops a tuple at a time from pq (we call this tuple $popped$). The algorithm compares popped to the new top tuple, emitting new result tuples for the second stage of the algorithm to consume and re-inserting some tuples into pq. Figure 8 illustrates the possible overlaps between $popped$ and top tuples. Here a solid thick line corresponds to a range of a dense fragment, a dashed thin line to a range of a sparse fragment, and a double solid line to a range of either a dense or a sparse fragment (recall that a dense array may consist of both dense and sparse fragments).

In case (i), since $popped$ does not overlap with top, it is guaranteed not to overlap with any other range of any other fragment (by the definition of pq) and, thus, it is simply inserted in the result. In case (ii), only the disjoint portion of the $popped$ range can be inserted in the result. The other part must be re-inserted in pq, because the overlapping portion of top may be from a more recent fragment (since the priority queue is sorted by sc). This re-insertion will result in (iii) or (iv) triggering. In case (iii), $popped$ is a range of a dense fragment that begins at the same sc value as top. This must come from a more recent fragment than top, thus top can be discarded (the disjoint portion of top is re-inserted into pq.) Note, however, that $popped$ may still overlap with some other older fragment starting at a later sc value, hence it cannot be emitted yet. Instead, the algorithm pops pq and processes the new top versus this $popped$; this will generally result in case (i), (ii) or (iii) being triggered.

Finally, case (iv) applies when $popped$ is sparse. Recall that initially the sparse ranges inserted into pq may also cover empty cells. Thus, the algorithm identifies the first two non-empty cells in $popped$, which requires reading from the data tile (these are shown as **x**'s in the figure). It then inserts the first coordinate as a unary *dense* tuple in pq, to be handled by case (iii). Finally, it re-inserts the truncated sparse range starting at the second coordinate into pq, as shown in the figure.

In summary, the algorithm performs a *right sweep* on the global cell order, and inserts disjoint ranges into the ordered list, such that recent fragments always overwrite older ones.

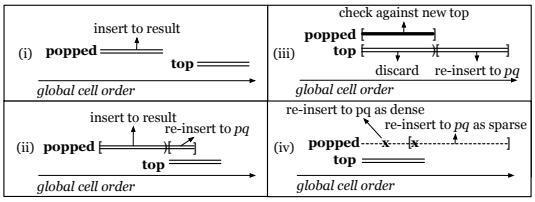


Figure 8: Possible overlaps of popped and top tuples in pq

Due to this invariant, the emitted list abides by the rules we stated above and, thus, the algorithm is correct.

Sparse arrays. The algorithm for sparse arrays is very similar to that for dense. There are only two differences in the first stage of computing the sorted cell range list. First, iteration does not focus on space tiles (since they may be too large in sparse arrays). Instead, it focuses only on ranges that start before the minimum end bounding coordinate of a data tile in a fragment, and then proceeds by properly updating the next minimum end bounding coordinate. This keeps the queue small in each iteration. Second, case (iii) of Figure 8 never arises, since a sparse array consists only of sparse fragments.

TileDB supports various read I/O methods, such as POSIX read, memory map (`mmap`), MPI-IO [14], even asynchronous I/O, which can be configured easily even upon run-time. The benefits of each method vary based on the application.

The above read algorithm demonstrates the importance of the global cell order, which maps a multi-dimensional problem into a single-dimensional one that can be handled more intuitively and efficiently. Because the algorithm spends more time when the cell ranges overlap considerably across fragments, it is beneficial for the user to *co-locate* updates when possible across batch writes. If this is not possible and the number of fragments becomes excessive, read performance may degrade. This is mainly because I/O is not performed sequentially, since data retrievals from multiple fragment files are interleaved. Moreover, storage space is wasted if there are many overwrites in both dense and sparse arrays, or numerous deletions in sparse arrays. TileDB offers a way to *consolidate* multiple fragments into a single one, described in Section 4.3.

4.2 Write

`Write` loads and updates data in TileDB arrays. Each *write session* writes cells sequentially in batches, creating a separate fragment. A write session begins when an array is initialized in write mode (with `init`), comprises one or more `write` calls, and terminates when the array is finalized (with `finalize`). Depending on the way the array is initialized, the new fragment may be dense or sparse. A dense array may be updated with dense or sparse fragments, while a sparse array is only updated with sparse fragments. We explain the creation of dense and sparse fragments separately below.

Dense fragments. Upon array initialization, the user specifies the subarray region in which the dense fragment is constrained (it can be the entire domain). Then, the user populates one buffer per array attribute, storing the cell values respecting the *global cell order*. This means the user must be aware of the tiling, as well as the tile and cell order specified upon the array creation (HDF5 and SciDB operate similarly). The buffer memory is managed solely by the user. The `write` function simply *appends* the values from the buffers into the corresponding attribute files, writing them sequentially, and without requiring any additional internal buffering (i.e., it writes directly from the user buffers to the files). Note that the user may call `write` repeatedly with different buffer contents, in the event that memory is insufficient to hold all the data at once. Each `write` invocation results in a series of sequential writes to the same fragment, one write per attribute file.

Sparse fragments. There are two different modes for creating sparse fragments. In the first, similar to the dense case, the user provides cell buffers to `write`. These buffers must be sorted in the global cell order. There are three differences with the dense case. First, the user provides values only for the *non-empty* cells. Second, the user includes an extra buffer with the coordinates of the non-empty cells. Third, TileDB maintains some extra *write state* information for each created data tile (recall that a data tile may be different from a space tile). Specifically, it counts the number of cells seen so far and, for every c cells (where c is the data tile capacity specified upon array creation), TileDB stores the minimum bounding rectangle (MBR) and the bounding coordinates (first and last coordinates) of the data tile. Note that the MBR initially includes the first coordinates in the data tile, and expands as more coordinates are seen. For each data tile, TileDB stores its MBR and bounding coordinates into the bookkeeping structures of the fragment.

Sparse fragments are typically created when random updates arrive at the system, and it may be cumbersome for the user to sort the random cells along the global order. To handle this, TileDB also enables the user to provide *unsorted* cell buffers to `write`. TileDB internally sorts the buffers (using multiple threads), and then proceeds as explained above for the sorted case. The only difference is that each `write` call in unsorted mode creates a *separate fragment*. This is very similar to the sorted runs created in the traditional *merge sort* algorithm. In Section 4.3 we explain that TileDB can effectively merge all the sorted fragments into a single one.

Sparse fragments also support *deletions*. TileDB handles deletions as insertions of empty cells (TileDB uses special constants for identifying values of "empty" cells). The user can then test if a retrieved cell is empty or not (using the special constants). If the number of deletions is high (e.g., many more deletion entries appear in a subarray query than actual results), the read performance will likely be impacted. In such cases, the user can invoke `consolidate` (see Section 4.3), which merges the fragments *purging* the deletion entries.

Two final notes concern *compression* and *variable-length* attributes, applicable to both dense and sparse fragments. If compression is enabled for some attribute, TileDB buffers the cell values corresponding to the same data tile for that attribute. When the tile fills up, TileDB compresses it internally and appends it to the corresponding file. For variable-length attributes, `write` takes *two* buffers from the user; the first stores the variable-length cell values, and the second stores the starting offsets of each cell value in the first buffer. TileDB uses this information to create the extra attribute file that stores the starting cell offsets, as explained in Section 3.

4.3 Consolidate

Consolidation takes a set of fragments as input and produces a single new output fragment. It is easy to implement in TileDB given the multi-fragment read algorithm discussed in Section 4.1: it simply repeatedly performs a `read` on the entire domain, providing buffers whose size is configurable depending on the available memory. After every read, the function invokes a `write` command which writes the retrieved cells into the new fragment. Due to the way it handles buffer overflow, TileDB stops reading when the buffers are full. If any of the read fragments are dense, the consolidated fragment is also dense. If all fragments are sparse, then the consolidated fragment is sparse. Since the cells are read in the global cell order, the resulting fragment stores the cells in the global cell order as well. In addition, the consolidation process discards any deleted entries. Upon successful termination, the function deletes the old fragments. The new fragment is identified by a timestamp that is larger than that of all the old fragments.

Consolidating many small fragments with a few large ones can be costly, since the total consolidation time is dominated by reading and writing the large fragments. This suggests consolidation should be applied on fragments of approximately equal size. Moreover, read performance degrades when many data tiles of different fragments overlap in the logical space. Thus, it is important to consolidate those fragments before others. To address these issues, TileDB enables the user to run consolidation on a specific subset of fragments. An additional benefit of selective consolidation applies specifically for the case where all the fragments have approximately the same size. In particular, the user can implement a *hierarchical* consolidation mechanism similar to that employed in LSM-Trees [22], to amortize the total consolidation cost.

5. PARALLEL PROGRAMMING

TileDB allows both concurrent reads and writes on single array, offers *thread-/process-safety* and atomic reads and writes. It also enables consolidation to be performed in the background without obstructing concurrent reads and writes. Fragments enable all of these features as writes happen to independent fragments, and partial fragments are not visible to readers.

Concurrent writes are achieved by having each thread or process create a separate fragment. No internal state is shared and, thus, no locking is necessary. Each thread/process creates a fragment with a *unique* name, using its id and the current time. Thus, there are no conflicts even at the file system level.

Reads from multiple processes are independent and no locking is required. Every process loads its own bookkeeping data from the disk, and maintains its own read state. For multi-threaded reads, TileDB maintains a single copy of the fragment bookkeeping data, utilizing locking to protect the open arrays structure so that only one thread modifies the bookkeeping data at a time. Reads themselves do not require locking.

Concurrent reads and writes can be *arbitrarily mixed*. Each fragment contains a special file in its sub directory that indicates that this fragment should be visible. This file is not created until the fragment is finalized, so that fragments under creation are not visible to reads. Fragment-based writes make it so that reads simply see the logical view of the array without the new fragment. This is in contrast to PHDF5, which imposes ordering constraints to ensure atomicity.

Finally, consolidation can be performed in the background in parallel with other reads and writes. Locking is required only for a very brief period. Specifically, consolidation is performed independently of reads and writes. The new fragment that is being created is not visible to reads before consolidation is completed. The only time when locking is required is after the consolidation finishes, when the old fragments are deleted and the new fragment becomes visible. TileDB enforces its own *file locking* at this point. After all current reads release their shared lock, the consolidation function gets an exclusive lock, deletes the old fragments, makes the new fragment visible, and releases the lock. After that point, any future reads see the consolidated fragment.

6. EXPERIMENTS

In this section we experimentally evaluate TileDB in a number of dense and sparse array settings. We focus on 3 competitors, 2D arrays, and fixed-length attributes due to space limitations. However, TileDB is an active project; we will keep on posting benchmark results and include future suggested comparisons online at `www.tiledb.org`.

Competitors. We compared TileDB against HDF5 [16] (for dense arrays), SciDB [18] (for dense and sparse arrays), and Vertica [21] (for dense and sparse arrays). For all parallel experiments, we used the parallel version of HDF5, namely PHDF5. SciDB and Vertica are complete database systems, but we only focused on their storage management capabilities on a single node, namely writes (load and update) and reads (various types of subarray queries), as well as parallel execution. All our benchmarks can found at `https://github.com/Intel-HLS/TileDB/tree/benchmarks`.

System configuration. We performed all our experiments on an Intel® x86_64 platform with a 2.3 GHz 36-core CPU and 128 GB of RAM, running CentOS6. We utilized a 4 TB, 7200 rpm Western Digital HDD and a 480 GB Intel® SSD both not RAID-ed and equipped with the *ext4* file system. Because the HDD is I/O bound even with a single thread, we ran serial experiments on the HDD. Because the SSD is so much faster, we ran our parallel experiments on SSDs.

We compared against SciDB v15.12, Vertica v7.02.0201, and HDF5 v1.10.0. TileDB is implemented in C++, and the source code is available online at `www.tiledb.org`. For TileDB, SciDB and HDF5, we experimented both with and without gzip compression, using compression level 6. Note that PHDF5 does not support parallel load or updates on compressed datasets. In addition, Vertica cannot be used without compression; the default compression method is RLE, which is currently not supported by TileDB. Therefore, we used Vertica with gzip (compression level 6). Note that Vertica uses a fixed 64 KB compression block. We gave an unlimited cache size to HDF5, SciDB and Vertica. TileDB does not use a caching mechanism, but it rather relies on the OS caching. Before every experiment we flush all caches, and after every load/update we sync the files. Finally, note that Vertica by

default uses all the available cores in the system. In order to vary the number of threads used in our parallel experiments, we simply shut down a subset of the cores at the OS level.

Datasets. For dense arrays, we constructed *synthetic* 2D arrays of variable sizes, with a single `int32` attribute, since the distribution of cell values in our experiments affects only compression (we perform writes and reads, not array computations). To make compression effective, we stored in each cell (i, j) the value $i * \#col + j$, where $\#col$ is the number of columns in the array (this led to a 2.9 compression ratio). The array domain type is `int64`. For the case of sparse arrays, we used datasets retrieved from the AIS database [8], which was collected by the National Oceanic and Atmospheric Administration by tracking ship vessels in the U.S. and international waters. We extracted attributes X (longitude), Y (latitude), SOG, COG, Heading, ROT, Status, VoyageID, and MMSI. We used X,Y as the dimensions (i.e., we created a 2D array), and the rest characteristics as the attributes. SciDB does not support real dimension values, thus we converted X and Y into `int64`, transforming the domain of (X,Y) to [0, 360M], [0, 180M]. For simplicity, we represented all attributes as `int64`. The resulting array is very sparse and *skewed*; most ship locations appear around the coasts of the U.S., they become very sparse as they move into the ocean, and are never on land.

Takeaways. Our evaluation reveals the following main results: (i) TileDB is several orders of magnitude faster than HDF5 in the case of random element updates for dense arrays, and at least as efficient in other settings, offering up to 2x better performance on compressed data; (ii) TileDB outperforms SciDB in all settings, generally by several orders of magnitude; (iii) TileDB is 2x-40x faster than Vertica in the dense case. It is also at least as fast as Vertica in the sparse case, featuring up to more than 2x faster parallel reads; (iv) the performance of the read algorithm in TileDB is robust up to a large number of fragments, whereas the consolidation mechanism requires marginally higher time than that needed for the initial loading; (v) TileDB exhibits excellent scalability as the dataset size and level of parallelism increase.

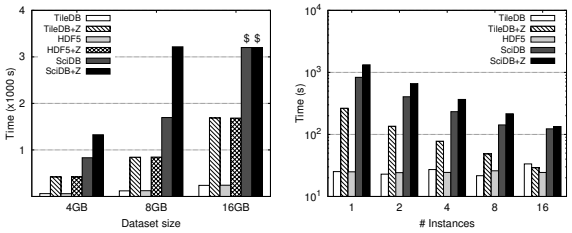
6.1 Dense Arrays

We compared versus HDF5, SciDB and Vertica. HDF5 and SciDB are natural competitors for the dense case. On the other hand, there are multiple ways to use Vertica for dense array management. In the following we focus mostly on HDF5 and SciDB, including a discussion on Vertica and a summary of our results at the end of the sub section.

Load. Figure 9(a) shows the time to load a dense array into TileDB, HDF5 and SciDB, as well as their gzip-compressed versions, denoted TileDB+Z, HDF5+Z and SciDB+Z, respectively. In this experiment, only one CPU core is activated and each system runs on a single instance. We generated 2D synthetic arrays with sizes 4 GB, 8 GB, and 16 GB (a scalability experiment with much larger arrays is described later), and dimension domains $50,000 \times 20,000$, $50,000 \times 40,000$, and $100,000 \times 40,000$, respectively. For all methods, we fix the space tile extents (chunk extents in HDF5 and SciDB) to $2,500 \times 1,000$, which effectively creates tiles/chunks of size 10 MB. The cell and tile order is row-major for all systems.

The input data are saved as binary files. The cell values are sorted in the global cell order. TileDB and HDF5 read the input file in internal buffers, and then issue `write` commands that write cells in batches. In SciDB, we follow the recommended method that loads the chunks directly into the array without the expensive `redimension` process.

TileDB matches the performance of HDF5 (it takes only 60s to load 4GB), and it is consistently more than an order of magnitude faster than SciDB. A dollar ('$') symbol indicates that the system did not manage to terminate within the shown time. Note that we experimented also with larger tile sizes for the case of 4 GB dataset (plot omitted), and found that all systems are unaffected by the tile size. However, the performance of TileDB+Z, HDF5+Z and SciDB+Z deteriorates as we increase the tile size; this is due to overheads of of gzip on large files.



(a) vs. dataset size (HDD) (b) vs. # instances (SSD)

Figure 9: Load performance of dense arrays

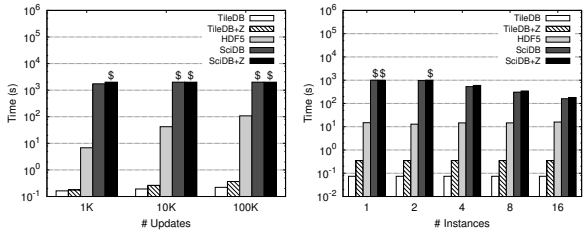
Figure 9(b) assesses the cost of loading data in parallel, for the 4GB array. For SciDB, we spawn multiple worker instances, whereas for TileDB and HDF5 we spawn multiple MPI processes. Each instance writes an equal number of tiles to the array. For SciDB, we follow the recommended approach that avoids `redimension`. We activate as many CPU cores as the number of tested instances. We exclude HDF5+Z, since PHDF5 does not allow parallel writes with compression.

TileDB and HDF5 are unaffected by the level of parallelism, as they are I/O bound. Moreover, each process writes large data chunks at a time monopolizing the internal parallelism of the SSD and, thus, no extra savings are noticed with parallel processes. SciDB seems to scale because it is CPU bound. Moreover, the compressed versions of all systems scale nicely, since they are CPU bound as well due to the CPU cost of gzip compression. The performance of TileDB and HDF5 match, whereas TileDB and TileDB+Z are between 2x and 7x faster than SciDB and SciDB+Z.

Update. Figure 10(a) shows the time to perform random element updates to the 4 GB array, as a function of the number of updated elements. We generated random element coordinates and stored them in a binary file. In HDF5 we batched the random updates using `select_elements`, whereas in SciDB we loaded the updates into a sparse array and then called `insert` with `redimension`. We exclude HDF5+Z as it does not support updates on compressed data.

TileDB is up to more than 2 orders of magnitude faster than HDF5, with 100K updates running in less than 1s in TileDB vs. more than 100s in HDF5, and more than 4 orders of magnitude faster than SciDB. This is due to the sequential, fragment-based writes of TileDB, as opposed to the in-place updates

in HDF5 and the chunk-based updates in SciDB. The performance gap increases with the number of updated elements.



(a) vs. # updates (HDD) (b) vs. # instances (SSD)

Figure 10: Random update performance of dense arrays

Figure 10(b) illustrates the update cost on SSDs when varying the number of instances and fixing the number of element updates to 100K. We split the updates equally into separate binary files, one for each instance. TileDB and HDF5 are unaffected by the number of instances and, thus, all the previous observations hold. SciDB performance improves with the number of instances. However, it is still up to more than 3 orders of magnitude slower than TileDB.

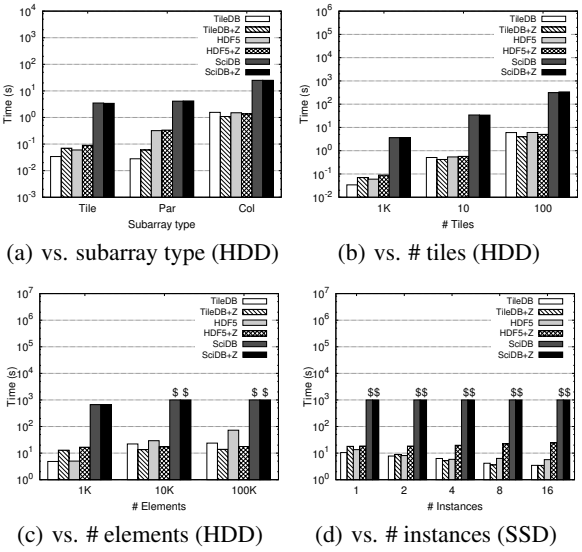
We performed an additional experiment with large sequential writes (e.g., whole tiles), not shown here. Note that in that experiment the performances of TileDB and HDF5 match, and both systems are substantially faster than SciDB (the observations are very similar to the case of single-instance load). Finally, we experimented with different tile/chunk sizes (plot omitted due to space limitations). TileDB and HDF5 are practically unaffected by this parameter (TileDB writes sequentially to a new fragment, whereas HDF5 writes in-place, utilizing a B-tree for finding the chunks whose height grows logarithmically with the number of chunks). However, SciDB's performance deteriorates substantially with the chunk size, as for every update an entire chunk must be read from and written to the disk. Note that this is true in both SciDB and SciDB+Z.

Subarray. Figure 11(a) shows the read performance of all systems (on a single instance), versus different subarray types, focusing on the 4 GB array with a fixed 2500×1000 (10 MB) tile size. In TileDB and HDF5, the results are written into the user's memory buffers. In SciDB, we execute the `between` function (followed by the recommended `consume`) and write the results in a temporary main-memory binary table. In TileDB we use the Linux `read` I/O method, whereas in TileDB+Z we use the `mmap` I/O method that leads to better performance (as it avoids double-buffering the compressed tiles). As noted in Section 4.1, TileDB can run on either of these read modes, and properly tuning the mode may affect performance.

We tested three different subarray types: (i) Tile, where the subarray covers exactly one tile, (ii) Par (for partial), where the subarray query is a 2499×999 rectangle completely contained in a tile, and (iii) Col, where the subarray is a full array column, vertically intersecting 20 tiles.

The performances of TileDB and HDF5 once again match, except for the case of Par where TileDB is 10x faster than HDF5. This is due to TileDB's efficient way of handling entire contiguous cell ranges instead of investigating cells one by one. TileDB is 1-2 orders of magnitude faster than SciDB.

Figure 11(b) plots the time as a function of the number of sequential tiles read. This experiment shows that all methods scale linearly with the number of tiles, maintaining their performance differences.



(a) vs. subarray type (HDD) (b) vs. # tiles (HDD)

(c) vs. # elements (HDD) (d) vs. # instances (SSD)

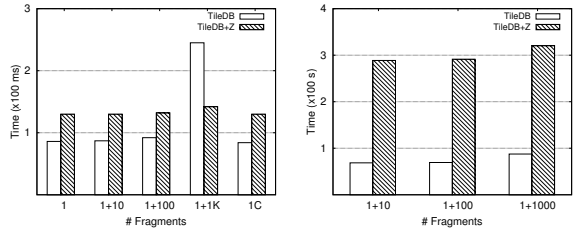
Figure 11: Subarray performance for dense arrays

Figure 11(c) shows read performance when reading random elements instead of contiguous subarrays. For this experiment, we generated random coordinates for the elements, which we batched in HDF5 with the `select_elements` API. In TileDB we used a similar API. In SciDB we used the `cross_between` function, which performs a similar batching of the random element reads. The cost in TileDB and HDF5 is the same. TileDB is up to 2 orders of magnitude faster than SciDB. This is because TileDB and HDF5 read only the relevant cells to the query from the disk, whereas SciDB fetches entire tiles.

Figure 11(d) shows the performance of parallel random element reads, as a function of the number of instances. We fixed the number of elements to 100K, and divided them equally across the instances. In TileDB, we found out that Linux `read` works better than `mmap` and, thus, we report the times using the latter I/O method. The reason is that a `mmap` call is more expensive than a `read` call, which becomes noticeable on SSDs with sub-millisecond access times. TileDB matches the performance of HDF5, and both systems seem to scale with the number of instances. However, TileDB+Z scales better than HDF5+Z, becoming up to 7x faster than HDF5+Z. On the other hand, SciDB did not complete in a reasonable amount of time for any setting.

Effect of number of fragments and consolidation. In Figure 12(a) we show `read` time as a function of the number of fragments present in an array. We take the initial 4 GB array as the first dense fragment, and update it with a number of batches, where each batch contains 1000 randomly generated elements, creating a new sparse fragment. on the x-axis of the figure, 1 means the initial 4 GB dense fragment, 1+10, 1+100 and 1+1000 means the initial fragment plus 10, 100 and 1000

sparse fragments, respectively, and 1C is 1+1000 after consolidation into a single fragment. We report the average time of 100 1K×1K subarray query randomly positioned inside the array space. Even after inserting 100 fragments, the read performance deteriorates only by 7% (86 ms vs. 92 ms). After 1000 fragments, it becomes 2.8x worse. However, after consolidation, the read time becomes 84 ms again. Note that the memory consumption of TileDB is negligible, regardless of the number of existing fragments.



(a) Subarray time (HDD) (b) Consolidation time (HDD)

Figure 12: Effect of # fragments in dense arrays

Figure 12(b) depicts the consolidation time, when varying the number of fragments. Consolidating 100 fragments into the 4 GB array takes the same time as the initial load, whereas merging 1000 fragments takes 3.4% longer. The memory consumption is 10 MB (a system parameter that determines the buffering size) regardless of the number of fragments being consolidated. As described in Section 4.3, the system can continue to process array operations while consolidation occurs.

Scalability. The last experiment for the dense case loads into TileDB and queries two large arrays with sizes 128 GB and 256 GB (we omit the plots due to space limitations), with tiles of size $2,500 \times 1,000$. The loading times were 1,815.78 s and 3,630.89 s, respectively, which indicates perfect scalability. We issued random 1K×1K subarray queries to both arrays (similar to those used in Figure 12(a)), and got times of 80 ms and 84 ms, respectively. Comparing these times to the 75 ms we observed in the case of the 6 GB array, we conclude that the read performance of TileDB is practically unaffected by the array size. Finally, the memory consumption upon loading in TileDB is negligible.

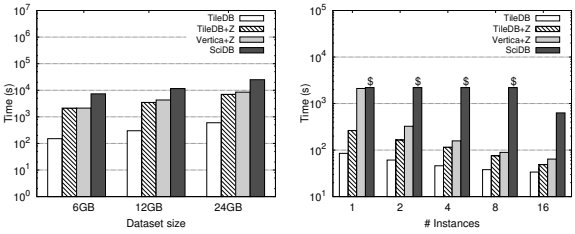
Vertica experiments. We conducted additional experiments with three different variants for implementing dense array management with Vertica, which offer different trade-offs: (i) we treated each dense element as a tuple, explicitly storing the X and Y element indices as table columns, and sorting first on X and then on Y (row-major order), (ii) similar to (i), but adding an extra column for the tile ID, and sorting on tile ID first, and then on X and Y (this imposes an identical global cell order to TileDB), and (iii) we followed the RAM [26] approach and stored a cell ID per element instead of X and Y (i.e., a single extra column), where the cell IDs are calculated based on the TileDB global cell order. For all these three alternatives, we tested compressing the extra columns with both GZIP and RLE. TileDB was consistently 2x-40x better in all settings versus all these solutions. The main reason is that Vertica performs operations on a cell-basis, whereas TileDB operates on large batches of cells during reads and writes. Individual benchmarks are available in the TileDB Github repo.

6.2 Sparse Arrays

We next focus on sparse arrays, comparing TileDB with Vertica+Z (gzip-compressed and following SRAM [19]) and SciDB on the AIS dataset. HDF5 is not optimized for sparse arrays, thus we omit it from these experiments.

Load. Figure 13(a) shows the load performance of the three systems. We created three different datasets with sizes 6 GB, 12 GB, 24 GB by loading from the dataset (originally in CSV format), after de-duplicating the coordinates. The dataset is organized based on the months of data collection. In TileDB, we called `write` for each month. In Vertica, we used the `COPY` statement (which bypasses the write optimized store to directly write to the read optimized store) to load each dataset into a relational table. Each tuple in this table corresponds to a (non-empty) array cell, where the X, Y coordinates represent different database columns and constitute the composite key. Vertica sorts the data first on X and then on Y, effectively enforcing a row-major order on the array elements. In SciDB, we first loaded the data into a 1D array, and then invoked `redimension` followed by `store`. The tile extents for TileDB and SciDB were 10K×10K, and the data tile capacity for TileDB was 10K.

TileDB is once again more than an order of magnitude faster than SciDB. Moreover, TileDB+Z is always slightly faster than Vertica+Z. Note that we experimented with different tile extents (not shown here) and discovered that they do not considerably affect TileDB and SciDB.



(a) vs. dataset size (HDD) (b) vs. # instances (SSD)

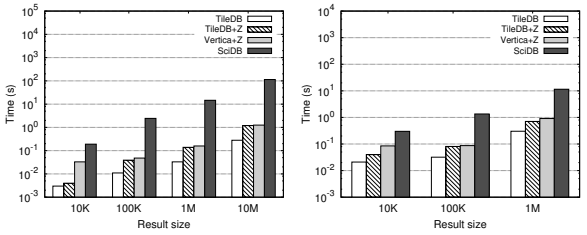
Figure 13: Load performance of sparse arrays

Figure 13(b) plots the time to load the 6 GB dataset in parallel, versus the number of instances. Note that Vertica by default uses all the cores in the system. Thus, when we vary the number of instances, we activate only as many CPU cores as the number of tested instances. The input files are evenly split by lines across the multiple processes in TileDB. In Vertica, the `COPY` command automatically utilizes the available cores to load from a single file. In SciDB, the evenly split files are saved inside the instance directories. TileDB, TileDB+Z and Vertica+Z scale nicely with the number of instances. The performance differences among the systems are similar to what was explained above for the serial case.

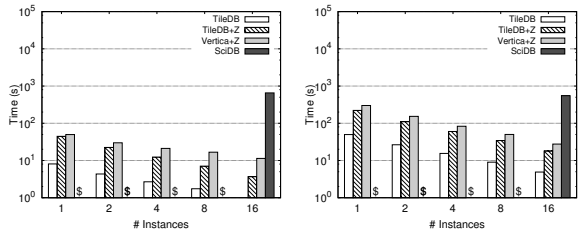
Subarray. The next set of experiments evaluates the subarray performance. In this experiment, we identified two array regions, one dense (Los Angeles harbor) and one sparse (the middle of Pacific ocean); henceforth, DQ (SQ) refers to this dense (sparse) domain area. In each of the following experiments, we selected a subarray with the target result size, and

then derived from it 50 subarrays of similar size by shifting it using a random offset. Each query returns the X and Y cell coordinates that fall in the specified subarray. In TileDB, we called `read` once per subarray; in SciDB we batched all reads with the `cross_between` command; in Vertica we issued a SQL `SELECT WHERE` query. Once again, the results of Vertica and SciDB are written into temporary main-memory binary tables. In TileDB, we fix the data tile capacity to 10K.

Figures 14(a) and 14(b) show the subarray times for the dense and sparse areas, as a function of the result size (single instance on HDDs). TileDB is 1-2 orders of magnitude faster than SciDB, and at least as fast as Vertica in all settings. Note that because we cannot disable compression in Vertica, it is not fair to compare uncompressed TileDB to Vertica+Z.



(a) DQ vs. result size (HDD) (b) SQ vs # result size (HDD)



(c) DQ vs. # instances (SSD) (d) SQ vs. # instances (SSD)

Figure 14: Subarray performance for sparse arrays

Figures 14(c) and 14(d) plot the subarray times for the dense and sparse areas, as a function of the number of instances (on SSD). In these two experiments, we evenly divided 320 random subarray queries (created inside the dense and sparse area, respectively) across the instances, and reported the total time. Each subarray returns 1M elements. TileDB+Z and Vertica+Z scale nicely with the number of instances and feature similar performance, whereas TileDB is more than an order of magnitude faster than SciDB in all settings.

Consolidate. We conducted a similar experiment to the dense case, drawing randomly the new cells from the AIS dataset. The observations are similar to the dense case (we omit the plot). The read time of TileDB deteriorates by only 18% after inserting 100 fragments, and by about 2x after 1000 fragments, but returns to normal after consolidation. Moreover, consolidation takes the same time as the original load (and, once again, can be performed in the background).

Acknowledgements

This work was partially supported under NSF grant IIS-1110370.

7. CONCLUSION

We presented the TileDB multi-dimensional array storage manager, which is optimized for both dense and sparse arrays. We described the basic concepts of TileDB, such as the organization of elements into tiles and fragments, its read and consolidation algorithms, and the effect of parallelism. We experimentally demonstrated that TileDB offers (i) orders of magnitude better performance than the HDF5 dense array manager for random element writes, while matching its read performance, (ii) better performance in all settings than the SciDB array database for both dense and sparse arrays, most of the time by several orders of magnitude, and (iii) an equivalent performance to the parallel Vertica columnar database on sparse arrays, while offering a programmer-friendly API-based interface similar to HDF5.

8. REFERENCES

[1] Apache Kylin. `http://kylin.apache.org/`.
[2] Broad Institute, Intel work together to develop tools to accelerate biomedical research . `http://genomicinfo.broadinstitute.org/acton/media/13431/broad-intel-collaboration`.
[3] Charm++. `http://charm.cs.illinois.edu/research/charm`.
[4] Enabling a Strict Consistency Semantics Model in Parallel HDF5. `https://www.hdfgroup.org/HDF5/doc/Advanced/PHDF5FileConsistencySemantics/PHDF5FileConsistencySemantics.pdf`.
[5] GenericIO. `http://trac.alcf.anl.gov/projects/genericio`.
[6] HDF5 for Python. `http://www.h5py.org/`.
[7] Legion Parallel System. `http://legion.stanford.edu/`.
[8] National Oceanic and Atmospheric Administration. Marine Cadastre. `http://marinecadastre.gov/ais/`.
[9] NetCDF. `http://www.unidata.ucar.edu/software/netcdf`.
[10] Parallel HDF5. `https://www.hdfgroup.org/HDF5/PHDF5/`.
[11] PLASMA. `http://www.netlib.org/plasma/`.
[12] PostgreSQL. `http://www.postgresql.org/`.
[13] PyTables. `http://www.pytables.org/`.
[14] ROMIO: A High-Performance, Portable MPI-IO Implementation. `http://www.mcs.anl.gov/projects/romio/`.
[15] ScaLAPACK. `http://www.netlib.org/scalapack/`.
[16] The HDF5 Format. `http://www.hdfgroup.org/HDF5/`.
[17] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The Multidimensional Database System RasDaMan. In *SIGMOD*, 1998.
[18] P. G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD*, 2010.
[19] R. Cornacchia, S. Héman, M. Zukowski, A. P. Vries, and P. Boncz. Flexible and Efficient IR Using Array Databases. *The VLDB Journal*, 17(1):151–168, 2008.
[20] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engin. Bulletin*, 35(1):40–45, 2012.
[21] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. *Proc. VLDB Endow.*, 5(12):1790–1801, 2012.
[22] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.
[23] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM TOCS*, 10(1):26–52, Feb. 1992.
[24] F. Rusu and Y. Cheng. A Survey on Array Storage, Query Languages, and Systems. *ArXiv e-prints*, 2013.
[25] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *SIGMOD*, 2011.
[26] A. R. van Ballegooij. RAM: A Multidimensional Array DBMS. In *EDBT Extended Database Technology Workshops*, 2004.