CAS CS 561
Paper Presentation
TitleDB Array Data Storage Manager

Guanzhang Li, Huda Irshad, Kaize Shi, Stephany Yipchoy

Introduction

What is the problem?

- Multi-dimensional data is **inefficient** in storage
    - Updates have significant latency
    - Data storage for sparse is inefficient
    - Variable-length data per element in the multidimensional space is not always supported

Why is it important?

- Multi-dimensional data is being increasingly used and common transactions have latency (e.g., read, write, update)

Older Approaches

HDF5 Library

- Does not handle sparse regions efficiently, requires manual work
- Not efficient for making updates on data that are randomly distributed across blocks

Relational Databases

- Encoding indices for non-empty cells degrades read performance
- Reads & writes are not directly from the file, but through a server

Key Ideas

- Tile format and attributes
  - Global ordering of cells, sparse and dense data sets, tile/data ordering
- Data tile: group of non-empty cells of fixed capacity
- File system: the physical organization of the data
- Fragmenting: creation of new array of update values of a previous existing data set
- Consolidation: merging fragments

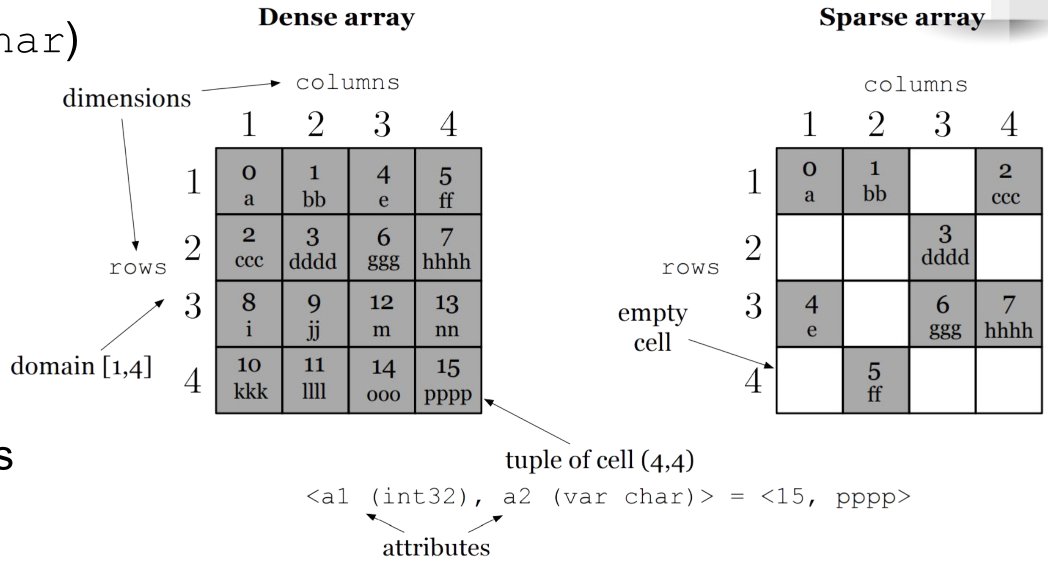
The Structure of an Array (Logical View)

Each cell (identified by coordinates) may be null or have a tuple of attribute entries

- Each attribute has a type
  - Primitive (e.g., `int`, `float`, `char`)
  - Fixed-sized vector
  - Variable-sized vector

Where to use sparse/dense array?

- Threshold of null-cell quantity
- E.g., dense array: discrete integers
- E.g., sparse array: continuous floats



Global Cell Order

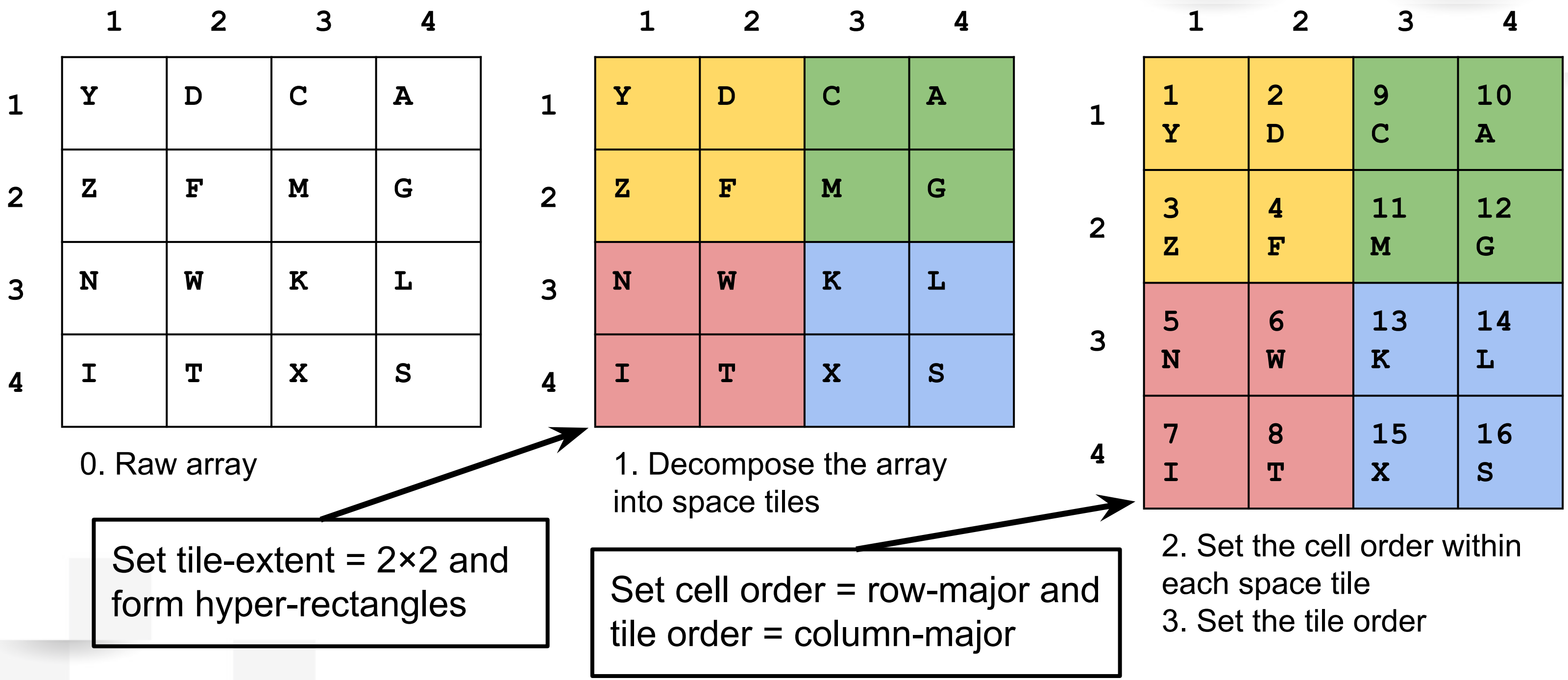
Each cell is identified by its coordinates `<row, col>`

Storage requires linear layout (mapping) for multi-dimensional data

TileDB enables **flexible** global cell order

- Co-location of cells depending on access pattern
- Ordering impacts performance
- Row-major layout vs. column-major layout
    - Row store vs. column store

Global Cell Order Construction (Dense Array)



1D representation: `[Y, D, Z, F, N, W, I, T, C, A, M, G, K, L, X, S]`

Space Tiles Structure (Dense Array)

1	2	9	10
3	4	11	12
5	6	13	14
7	8	15	16

space tile extents: 4×2
tile order: row-major
cell order: row-major

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

space tile extents: 4×2
tile order: row-major
cell order: column-major

1	3	5	7
2	4	6	8
9	11	13	15
10	12	14	16

space tile extents: 2×4
tile order: column-major
cell order: column-major

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

space tile extents: 2×2
tile order: row-major
cell order: row-major

1	2	9	10
3	4	11	12
5	6	13	14
7	8	15	16

space tile extents: 2×2
tile order: column-major
cell order: row-major

Space tile

- Color-coded hyper-rectangles (boxes) here
- A data tile is enclosed in logical space by a hyper-rectangle

Dense array: each data tile is a space tile

Global Cell Order Construction (Sparse Array)

Using the same method before can result in **empty/variant-sized tiles**

- Unused spaces in a space tile
- Ineffective compression
- Redundant reads

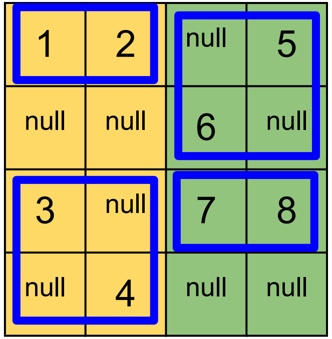
Solution: **data tile**

- Created by traversing the data in global cell order
- Atomic unit of compression

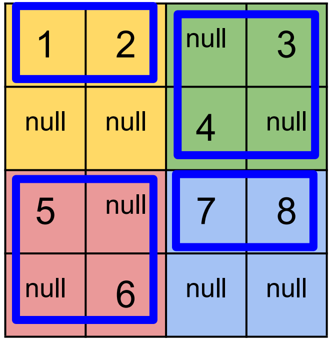
1	2	null	5
null	null	6	null
3	null	7	8
null	4	null	null

space tile extents: 4×2
tile order: row-major
cell order: row-major

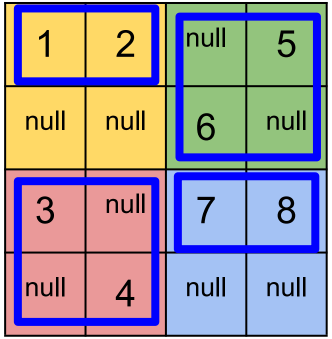
Space Tiles Structure (Sparse Array)



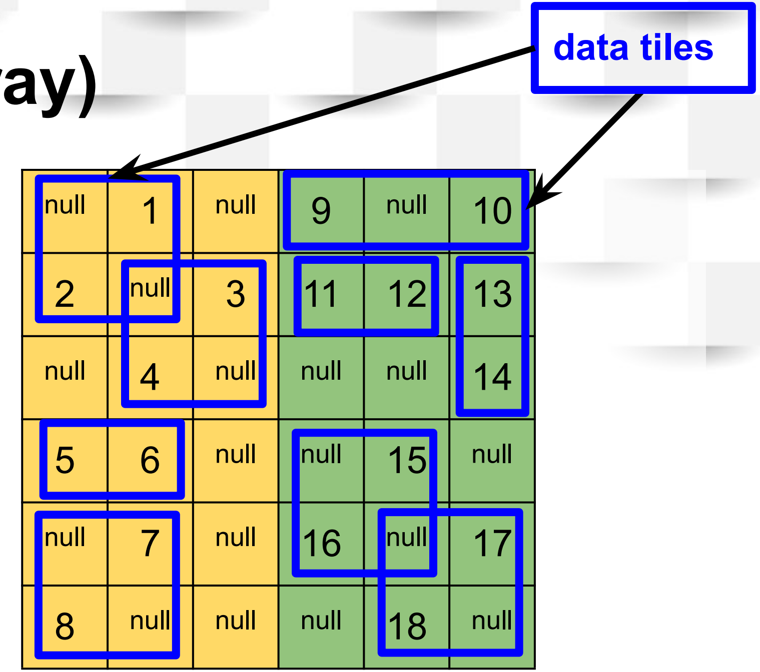
space tile extents: 4×2
tile order: row-major
cell order: row-major



space tile extents: 2×2
tile order: row-major
cell order: row-major



space tile extents: 2×2
tile order: column-major
cell order: row-major



space tile extents: 6×3
tile order: row-major
cell order: row-major

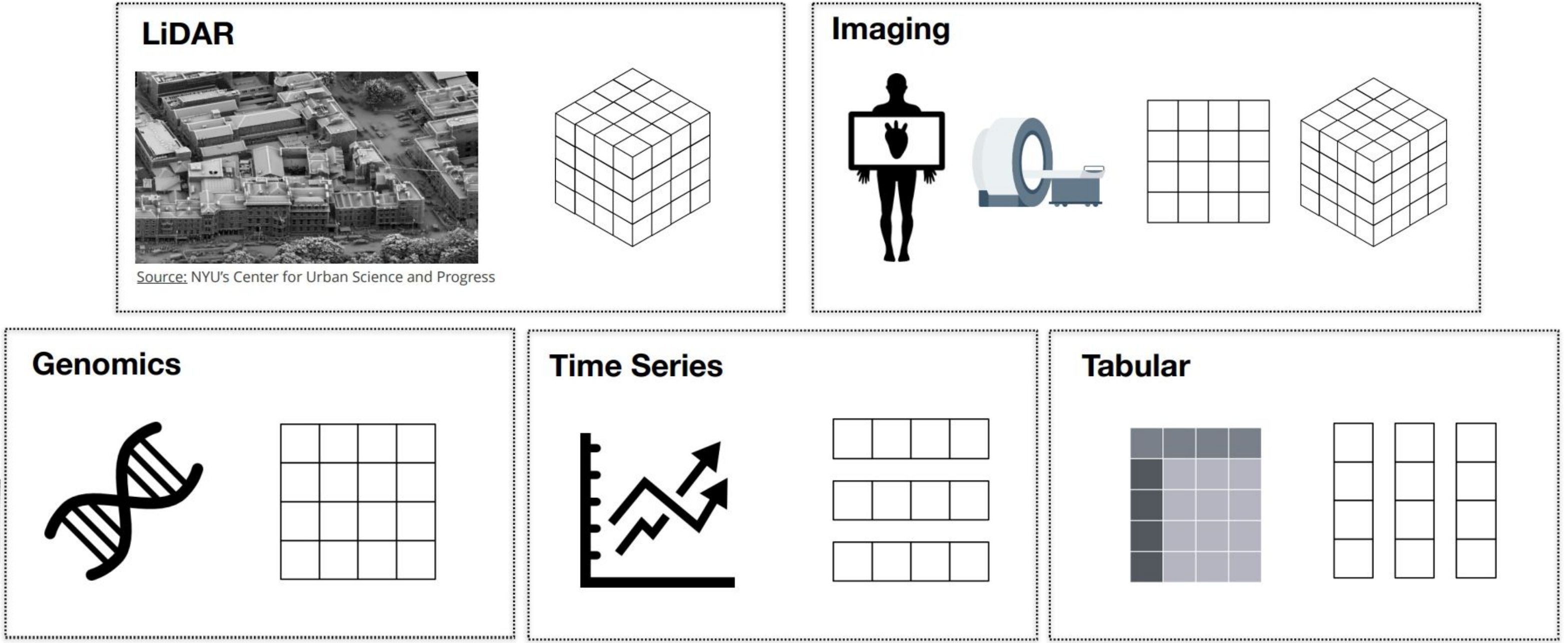
Sparse array: the user customizes data tile capacity

- Each data tile is an MBR (minimum bounding rectangle) of fixed size
- The size of an MBR is the quantity of non-null cells it has
- Data tiles can overlap, but each non-null cell only resides in one data tile

Space Tile Use Cases

Depending on how you determine the coordinates

Performing subarray queries based on coordinate ranges



Compression

Tile-based & attribute-based compressions

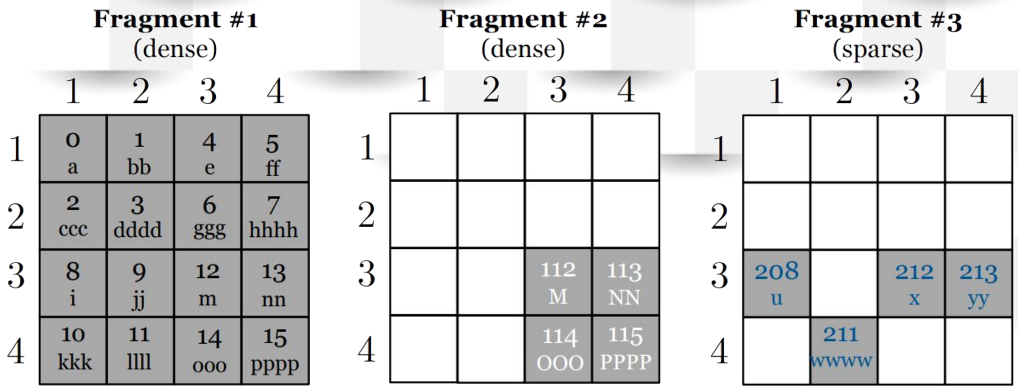
Customizable compression schemes for each attribute with a datatype

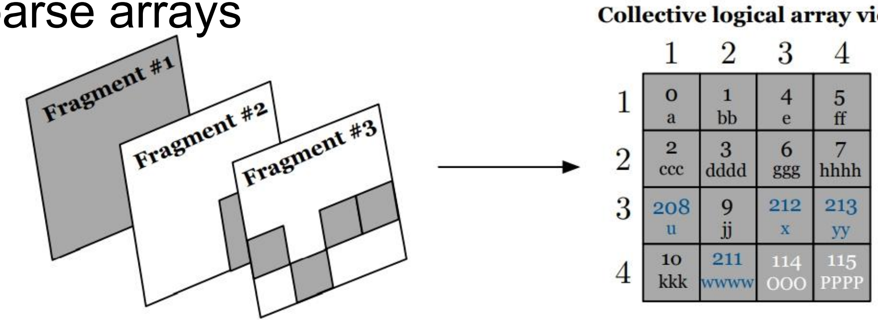
- Currently, only gzip is available
- More compression schemes on the way
  - Run-length encoding (RLE)
    - Describing repeated patterns
  - Lempel-Ziv-Welch (LZ)
    - Assigning code names to strings

Fragment

Handling **fast** updates & batch writes

- Timestamped snapshot (array)
- 1st fragment: initial loading
- 2nd to *n*-th fragment: additional modifications
- **Combining** all fragments retrieves the latest array
- Dense fragments handle dense arrays
  - A fragment can be dense w.r.t. a subarray
- Sparse fragments handle dense & sparse arrays



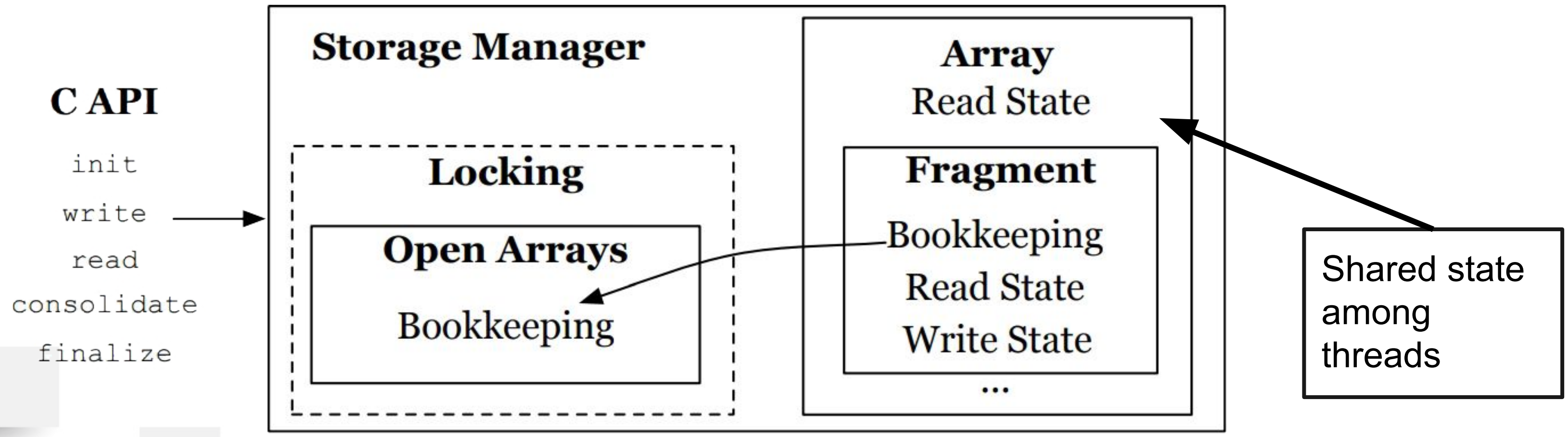


Array Metadata

1. Array schema (properties)
   - Name
   - Data schema (attributes)
   - Dimensions
   - Tile info
   - Compression schemes
   - … etc.
2. Fragment bookkeeping (sparse fragments)
   - Organization of the data in the fragment

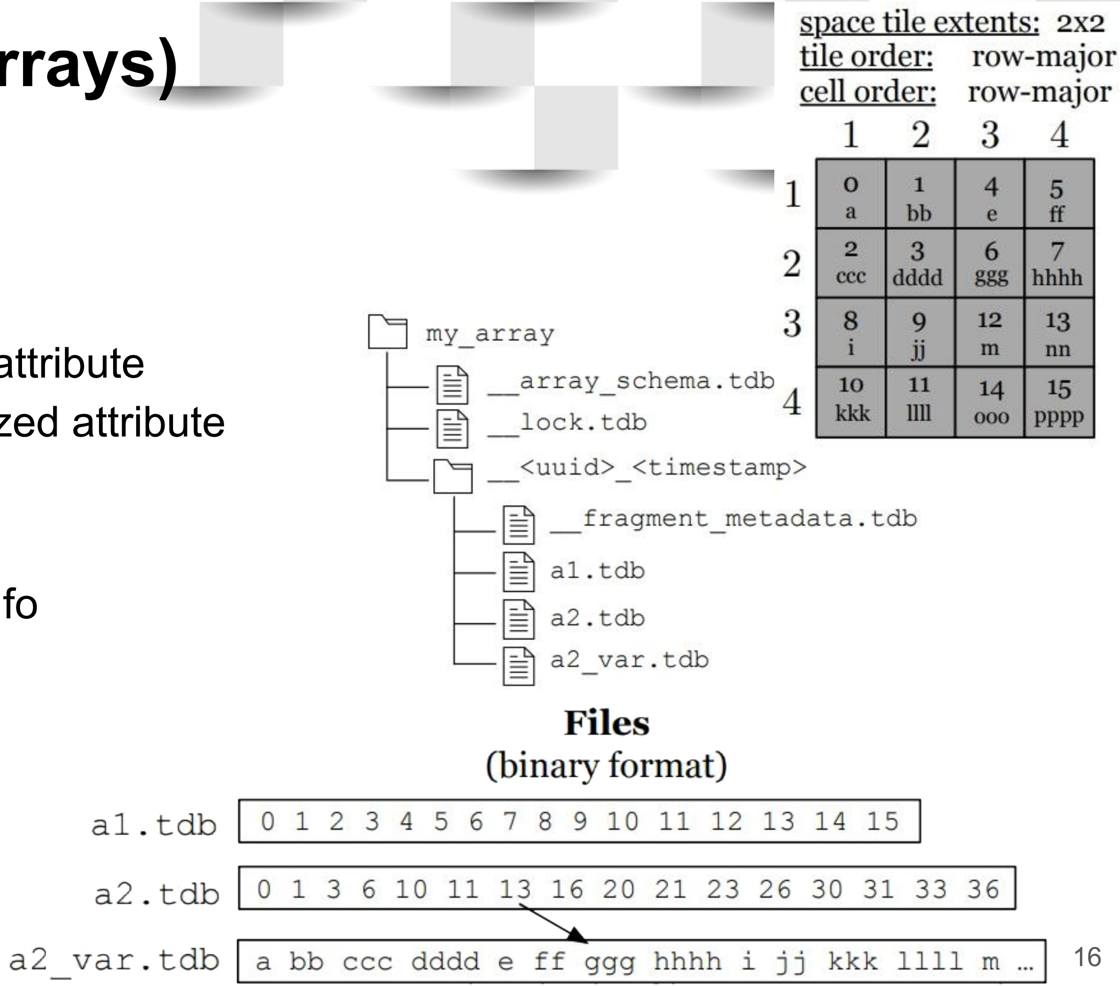
Storage Manager

- Maintaining in-memory state for open (initialized & non-finalized) arrays
- When shared among threads, locks are used to mediate the access
- Enabling parallelization



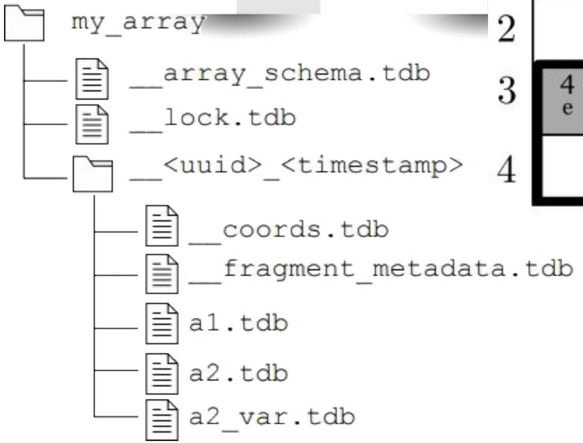
File System (Dense Arrays)

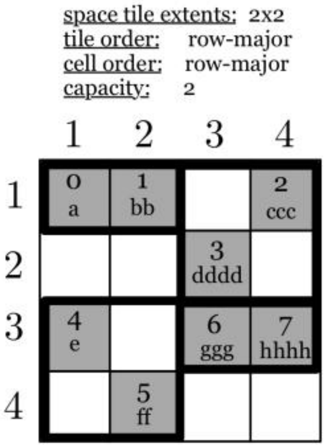
- ➔ 1 directory per array
- ➔ 1 subdirectory per fragment
  - ◆ Files (attributes)
    - 1 file per fixed-sized attribute
    - 2 files per variable-sized attribute
      - ○ Offset
      - ○ Value
- ➔ 1 binary file for array schema info

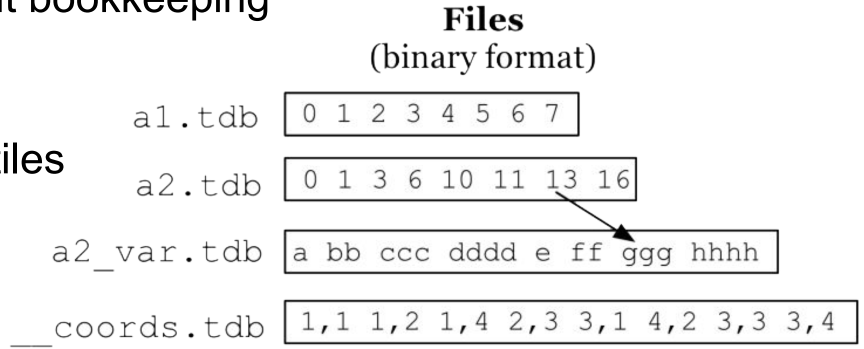


File System (Sparse Arrays)

- ➔ 1 directory per array (tracking non-empty cells)
- ➔ 1 subdirectory per fragment
  - ◆ Files (attributes)
    - 1 file per fixed-sized attribute
    - 2 files per variable-sized attribute
      - ○ Offset
      - ○ Value
  - ◆ 1 coordinates file
  - ◆ 1 compressed binary file for fragment bookkeeping metadata
    - MBRs
    - Bounding coordinates for data tiles
- ➔ 1 binary file for array schema info







Basic Operations

- Initialization (`init`)
- Read (`read`)
- Write (`write`)
- Consolidation (`consolidate`)
- Finalization (`finalize`)

Read (Dense Array)

We want to **read** the highlighted data

For each **space tile,** compute the coordinates `[sc, ec]` in **global cell order**

	1	2	3	4
1	0 a	1 bb	4 e	5 ff
2	2 ccc	3 dddd	6 ggg	7 hhhh
3	8 i	9 jj	12 m	13 nn
4	10 kkk	11 llll	14 ooo	15 pppp

Space tile extents: 2×2
Tile order: row-major
Cell order: row-major

Files
(binary format)

a1.tdb	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a2.tdb	0 1 3 6 10 11 13 16 20 21 23 26 30 31 33 36
a2_var.tdb	a bb ccc dddd e ff ggg hhhh i jj kkk llll m ...

Read (Dense Array)

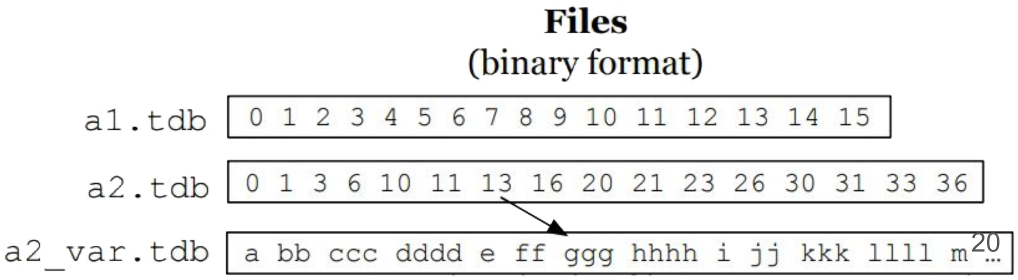
We want to **read** the highlighted data

For each **space tile,** compute the coordinates `[sc, ec]` in **global cell order**

`<(1,1), (2,2)>`

	1	2	3	4
1	0 a	1 bb	4 e	5 ff
2	2 ccc	3 dddd	6 ggg	7 hhhh
3	8 i	9 jj	12 m	13 nn
4	10 kkk	11 llll	14 ooo	15 pppp

Space tile extents: 2×2
Tile order: row-major
Cell order: row-major



Read (Dense Array)

We want to **read** the highlighted data

For each **space tile,** compute the coordinates `[sc, ec]` in **global cell order**

`<(1,1), (2,2)>`

`<(3,1), (3,1)>`

	1	2	3	4
1	0 a	1 bb	4 e	5 ff
2	2 ccc	3 dddd	6 ggg	7 hhhh
3	8 i	9 jj	12 m	13 nn
4	10 kkk	11 llll	14 ooo	15 pppp

Space tile extents: 2×2
Tile order: row-major
Cell order: row-major

Files
(binary format)

```
a1.tdb       0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a2.tdb       0 1 3 6 10 11 13 16 20 21 23 26 30 31 33 36
a2_var.tdb   a bb ccc dddd e ff ggg hhhh i jj kkk llll m ...
```

Read (Dense Array)

We want to **read** the highlighted data

For each **space tile,** compute the coordinates `[sc, ec]` in **global cell order**

`<(1,1), (2,2)>`

`<(3,1), (3,1)>`

`<(3,2), (3,2)>`

	1	2	3	4
1	0 a	1 bb	4 e	5 ff
2	2 ccc	3 dddd	6 ggg	7 hhhh
3	8 i	9 jj	12 m	13 nn
4	10 kkk	11 llll	14 ooo	15 pppp

Space tile extents: 2×2
Tile order: row-major
Cell order: row-major

Files
(binary format)

a1.tdb	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a2.tdb	0 1 3 6 10 11 13 16 20 21 23 26 30 31 33 36
a2_var.tdb	a bb ccc dddd e ff ggg hhhh i jj kkk llll m ...

Read (Dense Array)

```
<(1,1), (2,2)>
<(3,1), (3,1)>
<(3,2), (3,2)>
```

Take the coordinates, and compute the tuple that adds all fragment IDs **for each space tile considered**:

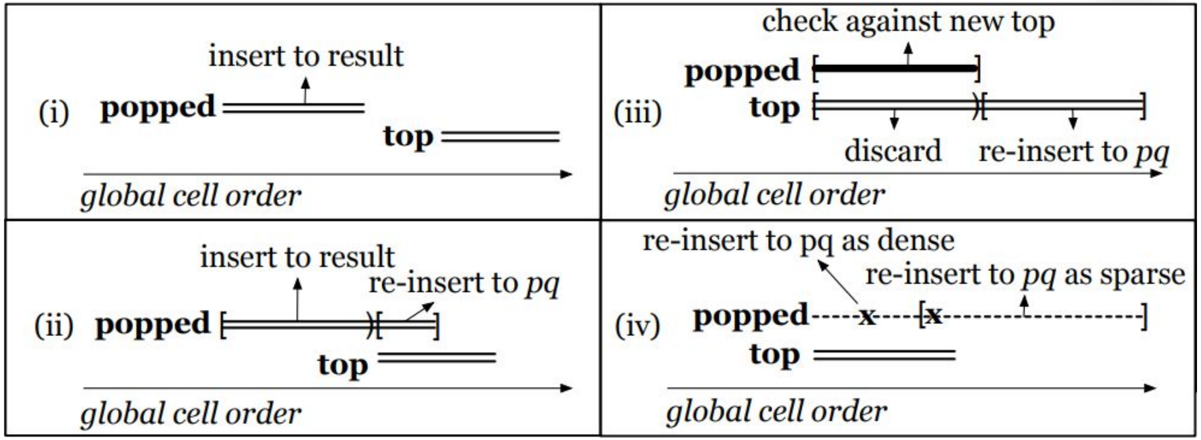
```
<[(1,1), (2,2)], 1>
<[(3,1), (3,1)], 1>
<[(3,2), (3,2)], 1>
```

Added to priority queue in global cell order

Read (Dense Array)

Pop the top element from the priority queue and compare it to the new one

```
<[(1,1), (2,2)], 1>
<[(3,1), (3,1)], 1>
<[(3,2), (3,2)], 1>
```

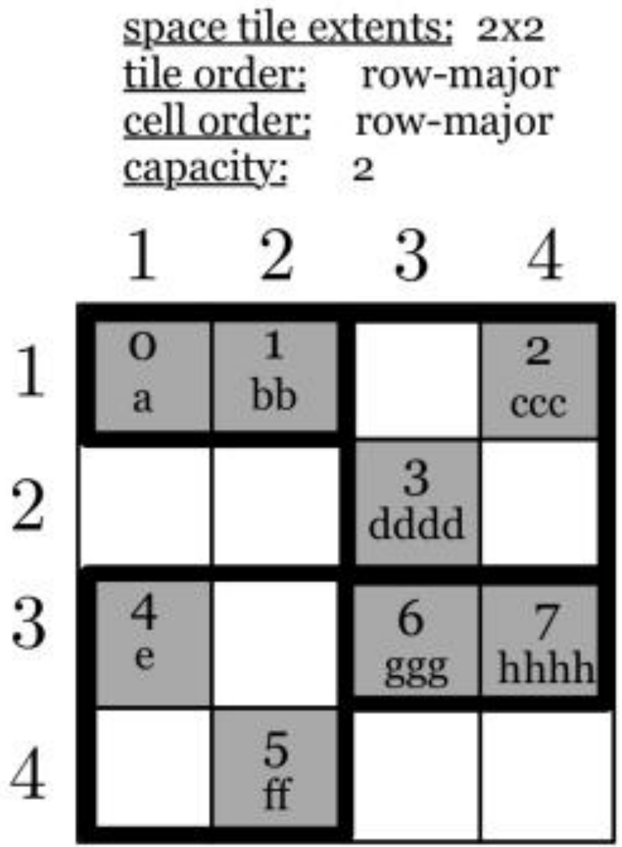


If pq coordinates aren't discarded or returned to pq, use a single I/O to place respective fragment files into the buffers

Read (Sparse Array)

Recall **bonding coordinates for data tiles**

Use data tiles instead of space tiles



Write (Dense Fragment)

0 a	1 bb	4 e	5 ff
2 ccc	3 dddd	6 ggg	7 hhhh
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

We want to update all positions, with **`a2_var == “hi” + i*a1`**

Files
(binary format)

a1.tdb	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a2.tdb	0 1 3 6 10 11 13 16 20 21 23 26 30 31 33 36
a2_var.tdb	a bb ccc dddd e ff ggg hhhh i jj kkk llll m …

Write (Dense Fragment)

Create 2 buffers (Variable-sized, but if fixed just one)

0 a	1 bb	4 e	5 ff
2 ccc	3 dddd	6 ggg	7 hhhh
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Buffer 1 (Offsets)

0														

Buffer 2 (Attribute value)

hi														

Write (Dense Fragment)

0 a	1 bb	4 e	5 ff
2 ccc	3 dddd	6 ggg	7 hhhh
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Buffer 1 (Offsets)

0	**2**													

Buffer 2 (Attribute value)

hi	**hii**													

Write (Dense Fragment)

0 a	1 bb	4 e	5 ff
2 ccc	3 dddd	6 ggg	7 hhhh
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Buffer 1 (Offsets)

0	2	**5**												

Buffer 2 (Attribute value)

hi	hii	**hiii**												

Write (Dense Fragment)

0 a	1 bb	4 e	5 ff
2 ccc	3 dddd	6 ggg	7 hhhh
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Buffer 1 (Offsets)

0	2	5	**9**											

Buffer 2 (Attribute value)

hi	hii	hii i	**hii ii**											

Write (Dense Fragment)

0 a	1 bb	4 e	5 ff
2 ccc	3 dddd	6 ggg	7 hhhh
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Buffer 1 (Offsets)

0	2	5	9	**13**										

Buffer 2 (Attribute value)

hi	hii	hii i	hii ii	**hiii ii**									

Write (Dense Fragment)

0 a	1 bb	4 e	5 ff
2 ccc	3 dddd	6 ggg	7 hhhh
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Buffer 1 (Offsets)

0	2	5	9	13	19	25	32	40	49	59	70	82	95	109

Buffer 2 (Attribute value)

hi	hii	hiii	hiii i	hiii ii	hiii iii	hiii iiii	hiiiii iii	hiii iiii ii	hiii iiii iii	hiii iiii iiii	hiii iiii iiii i	hiii iiii iiii ii	hiii iiii iiii iii	hiii iiii iiii iiii

Write (Dense Fragment)

Load information from buffer back into a new fragment

0 hi	1	4	5
2	3	6	7
8	9	12	13
10	11	14	15

Buffer 1 (Offsets)

0	2	5	9	13	19	25	32	40	49	59	70	82	95	109

Buffer 2 (Attribute value)

hi	hii	hiii	hii ii	hiii ii	hi ii ii i	hii iii ii	hii iii iii	hii iii iii i	hii iii iii ii	hii iii iii iii	hiii iiii iiii i	hiii iiii iiii ii	hiii iiii iiii iii	hiii iiii iiii iiii

Write (Dense Fragment)

0 hi	**1 hii**	4	5
2	3	6	7
8	9	12	13
10	11	14	15

Load information from buffer back into a new fragment

Buffer 1 (Offsets)

0	**2**	5	9	13	19	25	32	40	49	59	70	82	95	109

Buffer 2 (Attribute value)

hi	**hii**	hii i	hiii i	hiii ii	hi ii ii i	hii iii ii	hii iii iii	hii iii iii i	hii iii iii ii	hii iii iii iii	hiii iiii iiii i	hiii iiii iiii ii	hiii iiii iiii iii	hiii iiii iiii iiii

Write (Dense Fragment)

Load information from buffer back into a new fragment

AND SO ON ...

0 hi	1 hii	4	5
2	3	6	7
8	9	12	13
10	11	14	15

Buffer 1 (Offsets)

0	**2**	5	9	13	19	25	32	40	49	59	70	82	95	109

Buffer 2 (Attribute value)

hi	**hii**	hii i	hiii i	hiii ii	hi ii ii i	hii iii ii	hii iii iii	hii iii iii i	hii iii iii ii	hii iii iii iii	hiii iiii iiii i	hiii iiii iiii ii	hiii iiii iiii iii	hiii iiii iiii iiii

Write (Sparse Fragment Method 1)

We want to update the highlighted cells with
`a2_var == “hi” + i*a1`

Create 3 buffers:

Offsets for attribute values

Attribute values

Coordinates

0 a	1 bb	4 e	5 ff
2 ccc	3 dddd	6 ggg	7 hhhh
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Files
(binary format)

a1.tdb	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
a2.tdb	0 1 3 6 10 11 13 16 20 21 23 26 30 31 33 36
a2_var.tdb	a bb ccc dddd e ff ggg hhhh i jj kkk llll m …

Write (Sparse Fragment Method 1)

Offsets for attribute values

0			

Attribute values

hi			

Coordinates

(1,1)			

0 a	1 bb	4 e	5 ff
2 ccc	3 dddd	6 ggg	7 hhhh
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Write (Sparse Fragment Method 1)

Offsets for attribute values

0	**2**		

Attribute values

hi	**hii**		

Coordinates

(1,1)	**(1,2)**		

0 a	1 bb	4 e	5 ff
2 ccc	3 dddd	6 ggg	7 hhhh
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Write (Sparse Fragment Method 1)

0 a	1 bb	4 e	5 ff
2 ccc	3 dddd	6 ggg	7 hhhh
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Offsets for attribute values

0	2	**15**	

Attribute values

hi	hii	**hiiiiiii**	

Coordinates

(1,1)	(1,2)	**(2,3)**	

Write (Sparse Fragment Method 1)

0 a	1 bb	4 e	5 ff
2 ccc	3 dddd	6 ggg	7 hhhh
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Offsets for attribute values

0	2	15	**23**

Attribute values

hi	hii	hiiiiiii	**hiiiiiiii**

Coordinates

(1,1)	(1,2)	(2,3)	**(2,4)**

Write (Sparse Fragment Sorted)

Now we write everything back to a new fragment

Offsets for attribute values

0	2	5	13

Attribute values

hi	hii	hiiiii ii	hiiiii iii

Coordinates

(1,1)	(1,2)	(2,3)	(2,4)

0 hi	1 hii		
		6 hiiii iii	7 hiiii iiii

Deletion

Now we have a (consolidated) array that looks like this, but we want to delete A2 in the highlighted cell

Create 3 buffers:

Offset for attribute values

`0`

Attribute values

`NULL`

Coordinates

`(1,3)`

0 hi	1 hii	4 e	5 ff
2 ccc	3 dddd	6 hiii iiii	7 hiii iiii i
8 i	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Deletion

When we write this back and consolidate the array, we are left with the following

Deletions in TileDB are dealt with as **empty writes**

0 hi	1 hii	4 e	5 ff
2 ccc	3 dddd	6 hiii iiii	7 hiii iiii i
8 **NULL**	9 jj	12 m	13 nn
10 kkk	11 llll	14 ooo	15 pppp

Unsorted Writes

Many writes come in as random one off changes, so TileDB supports unsorted writes where each individual coordinate change creates a new fragment

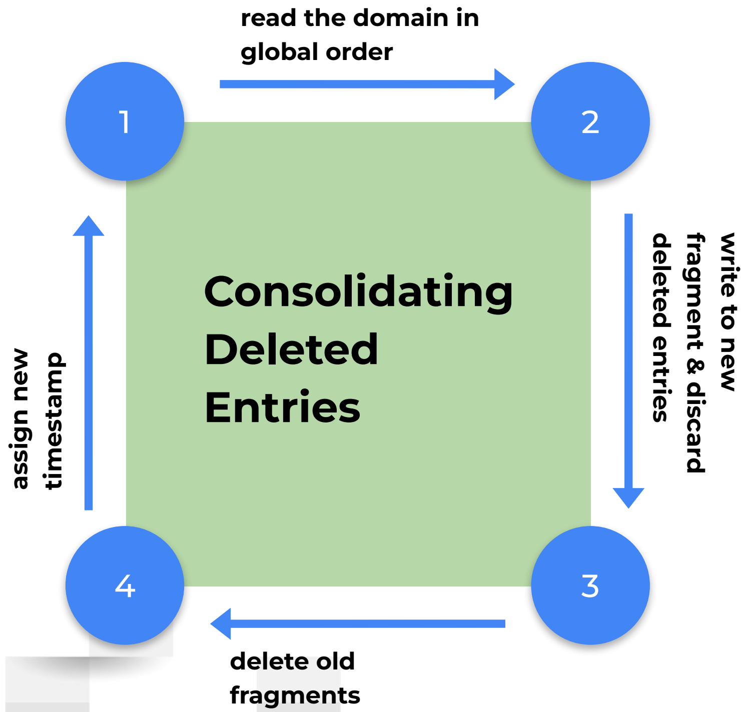
Creates issues with **read performance** because too many fragments, which is why consolidation is necessary

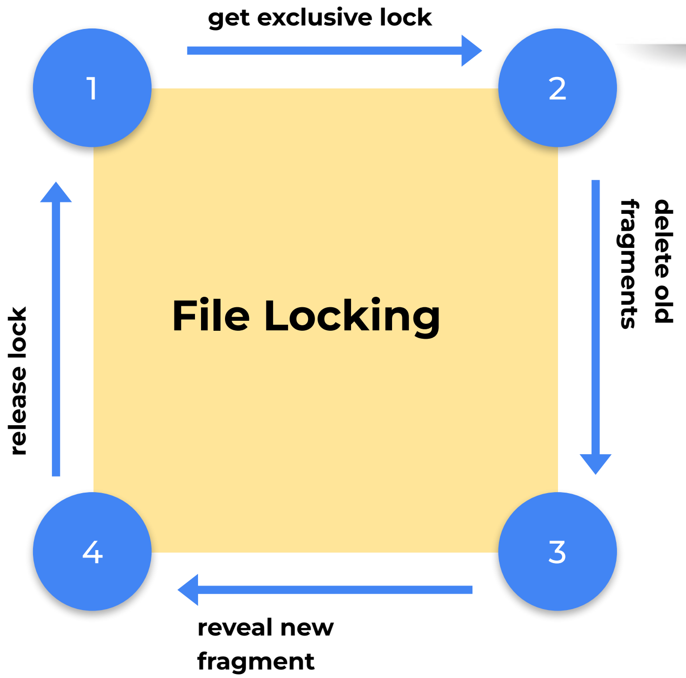
Consolidation

What happens if we have too many fragments?

- Read performance **degrades**
- Solution: **merge** the fragments
  - Input: multiple fragments
  - Output: single fragment
- Purging the deletion entries
- Prioritize the fragments with overlapping data tiles
  - Selective
  - Hierarchical
- This operation is independent from reads & writes
- Similar to LSM Tree Compaction

Consolidation





Parallel Programming

Concurrent write transactions

- ❖ Each process creates its own new fragment, so this action is stateless
- ❖ No lock required

Concurrent read transactions

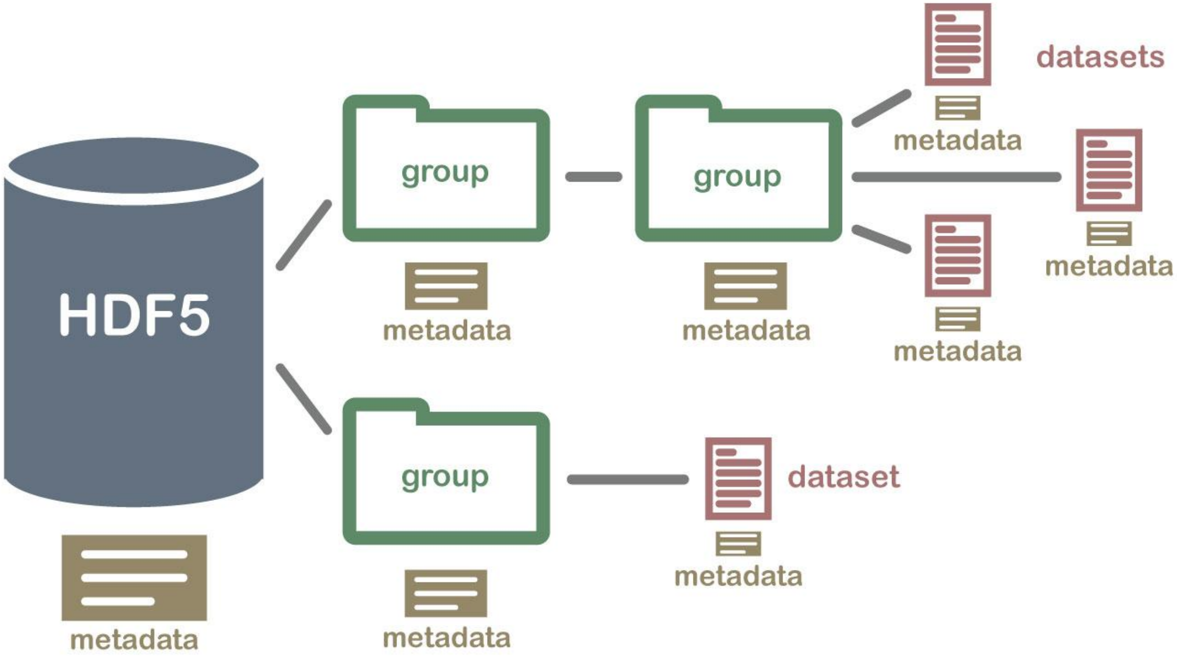
- ❖ Each process has its own copy of the bookkeeping of the data
- ❖ Read actions do not need a lock
- ❖ Accessing bookkeeping data requires the lock

Concurrent reads and writes

- ❖ Fragments remain invisible to read actions but are still written to by the write actions
- ❖ Fragments are not made visible to reads before consolidation takes place on that fragment

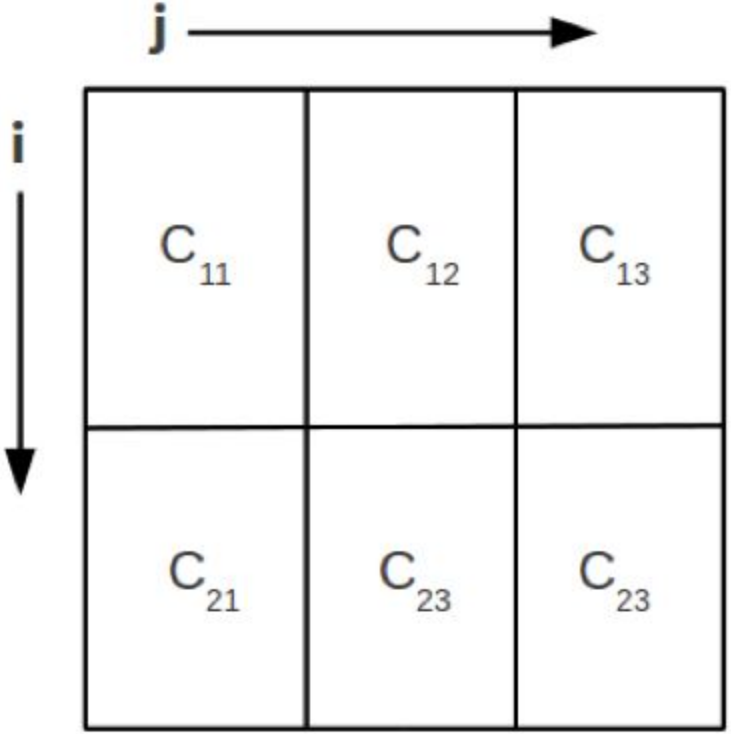
Experiments

Competitor - HDF5



Competitor - SciDB

Each dimension of an array is divided into chunks



Chunks arranged in row-major order

C_{11} -> server 1

C_{12} -> server 2

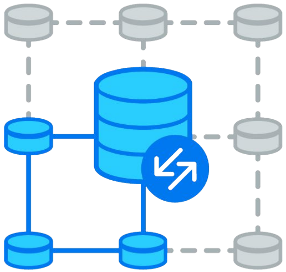
C_{13} -> server 3

C_{21} -> server 4

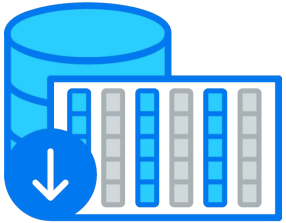
C_{22} -> server 1

C_{23} -> server 2

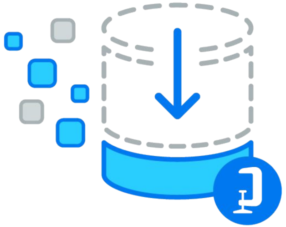
Competitor - Vertica



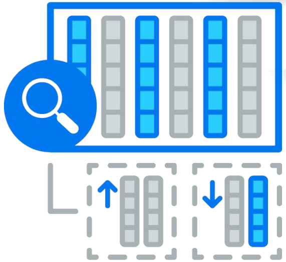
Parallel Processing



Column Storage



Compression



Projection

Datasets

Dense array

1. Synthetic 2D arrays
2. Single `int32` attribute
3. Array domain type is `int64`

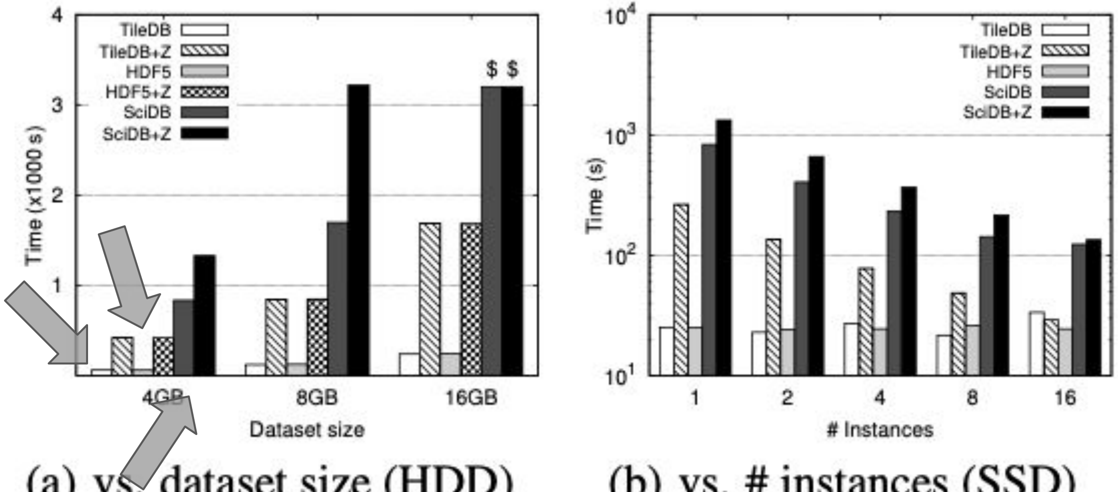
Sparse array

1. AIS database
2. Tracking ship vessels in the US and international waters
3. For simplicity, we represented all attributes as `int64`
4. The resulting array is very sparse and skewed

Dense Arrays - Load

TileDB and HDF5 read the input file in buffer and then write cells in bathes

SciDB loads the chunks directly into the array



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

Figure 9: Load performance of dense arrays

TileDB and HDF5 have stable performance because they are I/O bound

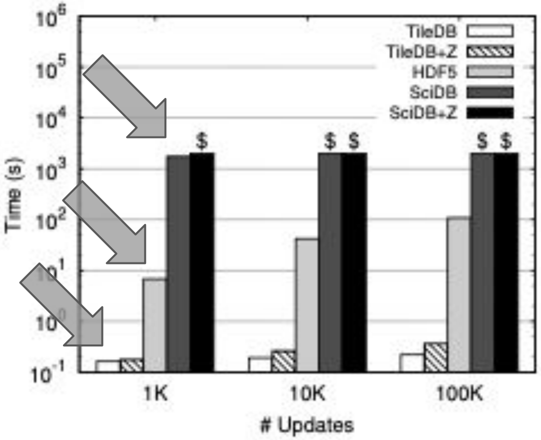
SciDB benefits from # instances scale because it is CPU bound

Dense Arrays - Update

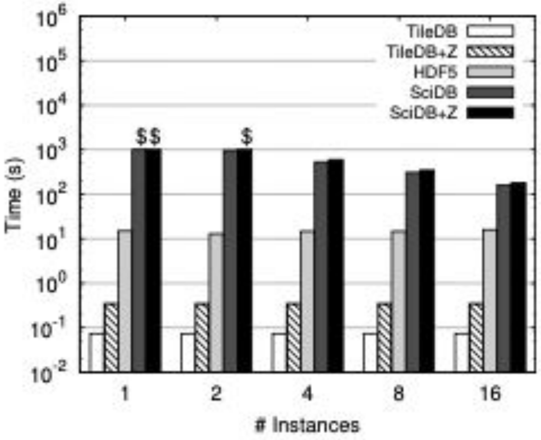
TileDB performs sequential & fragment-based writes

HDF5 performs in-place updates

SciDB performs chunk-based updates



(a) vs. # updates (HDD)



(b) vs. # instances (SSD)

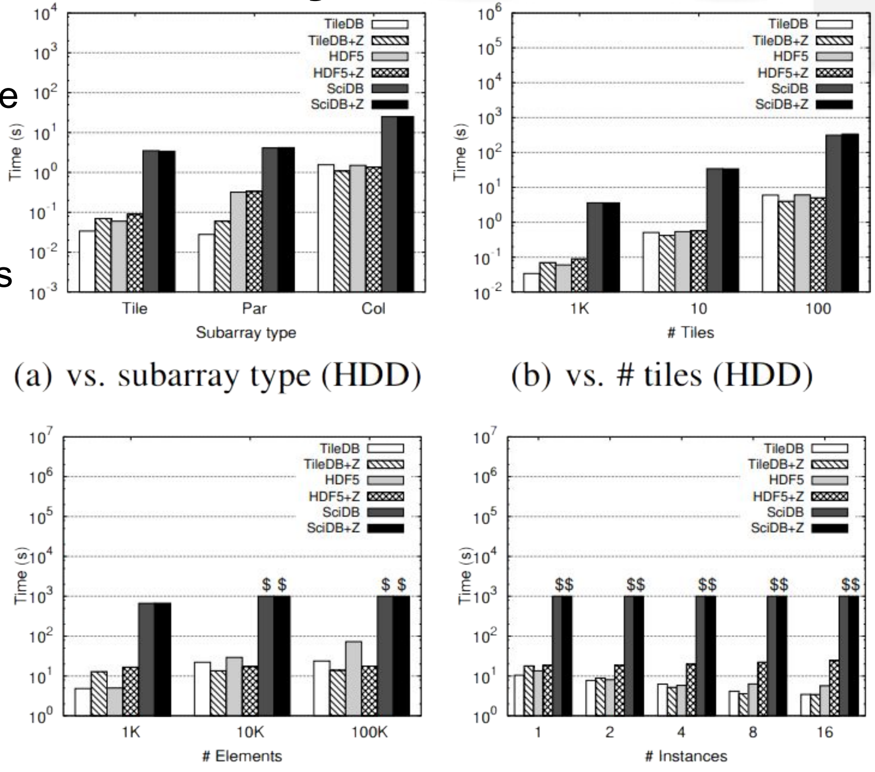
Dense Arrays - Subarray

Tile: subarray covers one tile
Par: subarray is 2499 × 999 in one tile
Col: full array column, vertically intersecting 20 tiles

Number of sequential tiles read

Read performance when reading random elements instead of contiguous subarrays

Parallel random element reads



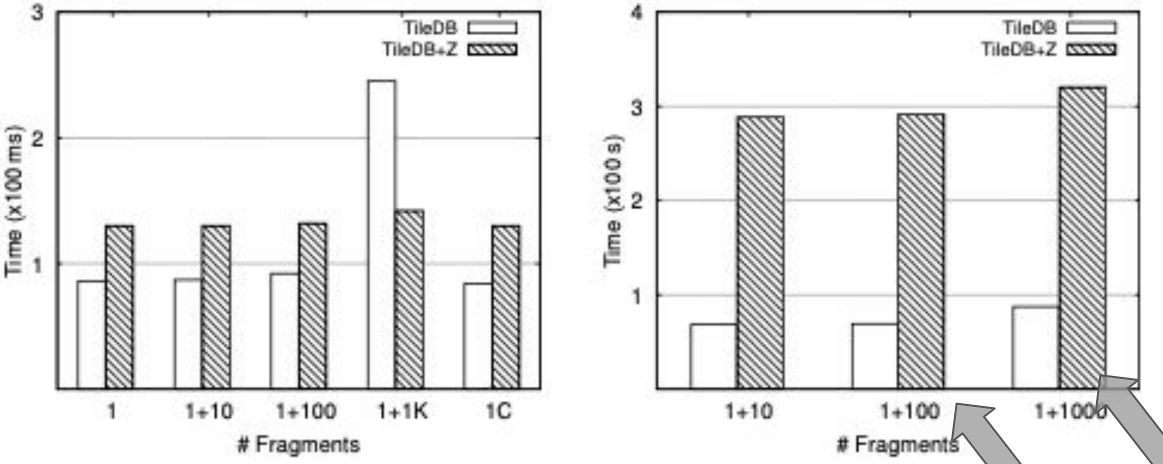
(a) vs. subarray type (HDD) (b) vs. # tiles (HDD)

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

Figure 11: Subarray performance for dense arrays

Dense Arrays - Effect of Number of Fragments and Consolidation

1+10, 1+100 and ***1+1000*** mean the initial fragment plus 10, 100 and 1000 sparse fragments, respectively, and ***1C*** is 1+1000 after consolidation into a single fragment



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

Figure 12: Effect of # fragments in dense arrays

Takes 3.4% longer than 1+100

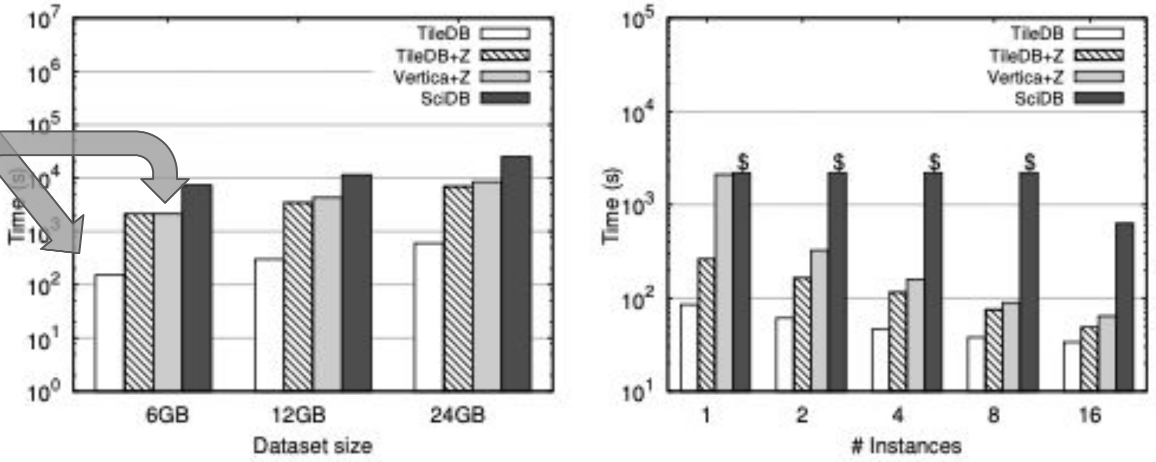
Take time as init load

Sparse Arrays - Load

TileDB *write* base on month

Vertica load data in relation table, then sort on X and Y, then enforcing a row-major order on array element.

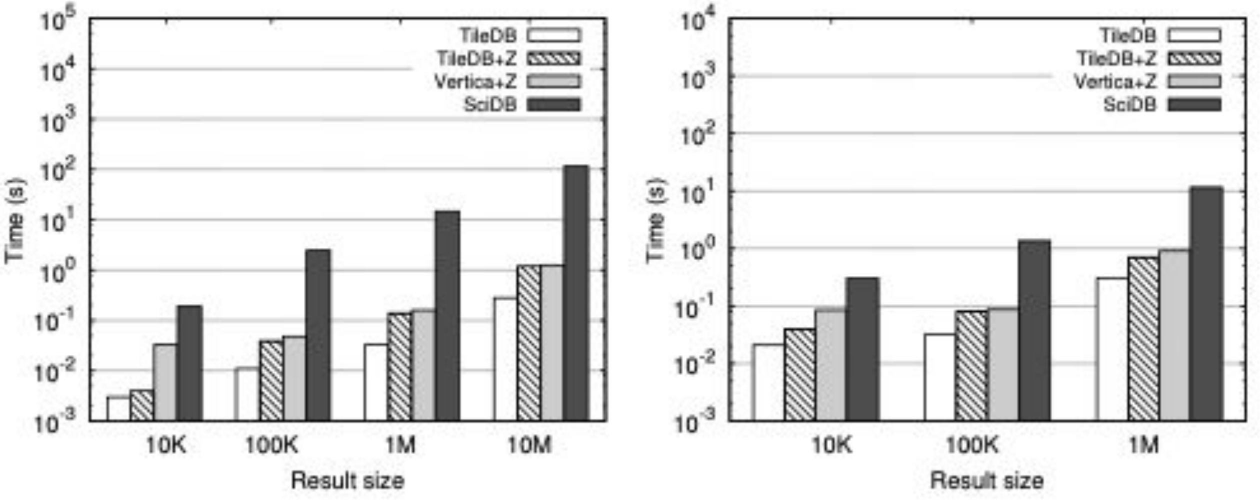
SciDB load data into 1D array, then *redimension*.



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

Figure 13: Load performance of sparse arrays

Sparse Arrays - Subarray



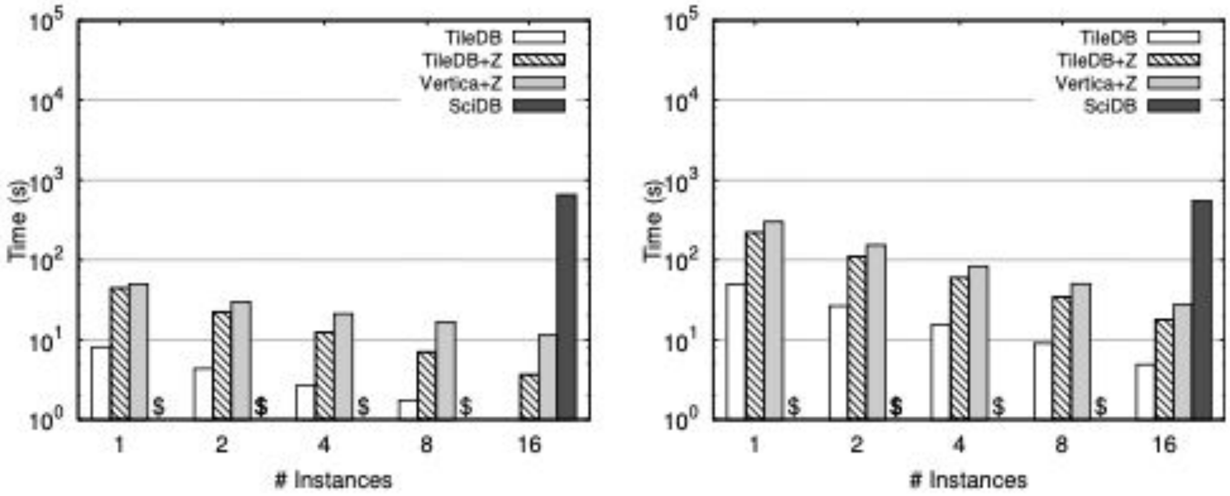
(a) DQ vs. result size (HDD) (b) SQ vs # result size (HDD)

DQ: Dense domain Query

SQ: Sparse domain Query

Sparse Arrays - Subarray

Evenly divide 320 random subarray queries across the instances, and report the total time.



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

Figure 14: Subarray performance for sparse arrays

Results

TileDB is faster than HDF5 in random element updates, and has 2× better performance on compressed data.

TileDB outperforms SciDB in all settings.

TileDB is 2× ~ 40× faster than Vertica in dense case, as fast as Vertica in sparse case, 2× faster in parallel reads.

TileDB read algorithm is robust.

TileDB has excellent scalability as the dataset size and level of parallelism increase.

Conclusion

1. Flexible with dense & sparse arrays
2. Data transactions of the store manager makes a fully functional system with some limitations but higher performance to others of its kind
3. Global ordering of the cells increases read performance of the multidimensional arrays
4. Expandable to parallel computing