The TileDB Array Data Storage Manager

Ziyang Chen, Yuansheng Dong

Introduction

- Basic Concepts
- Existing Array Managment system
- Introduction of TileDB
- Physical Organization
- Core Functions of TileDB
- Paralell Programming
- Evaluation
- Conclusion

Basic Concepts

- **Dense array**: every array element has a value
  - i.e. an astronomical image

- **Sparse array**: the majority of the array elements are **empty**
  - i.e. geo-locations: points in a 2D coordinate space

Existing Array Management Systems

- HDF5
- SciDB
- Relational Databases

Existing Array Management Systems

- HDF5
  - groups array elements into regular hyperrectangles (chunks) which are stored on the disk
  - Shortcomings

Existing Array Management Systems

- Shortcomings
    - Can not efficiently capture sparse arrays
        - represent denser regions of a sparse array as separate dense array
        - large cost to track their changes
    - HDF5 is optimized for **in-place** writes of large blocks
        - result in poor performance of writing small blocks of elements

Existing Array Management Systems

- PHDF5 limitation:

    × concurrent writes to compressed data

    × variable length element values

    operation atomicity requires some coding format from user

Existing Array Management Systems

- SciDB
  - array orientation database
  - implement own storage managers
  - can serve as the storage layer for other scientific applications built on top

Existing Array Management Systems

- Shortcomings
  - not design for sparse arrary
  - requires reading and updating an entire chunk (even a small portion)

Existing Array Management Systems

- Relational databases (MonetDB or Vertica)
    - used as the storage backend for array management
    - storing non-empty elements as records
    - encoding the element indices as extra table columns
    - poor performance for dense array

Introduction of TileDB

What is TileDB?

- efficient writes and reads to arrays
- for both dense and sparse array
- supporting compression, parallelism and more

KEY IDEA:

It organizes array elements into ordered collections called fragments.

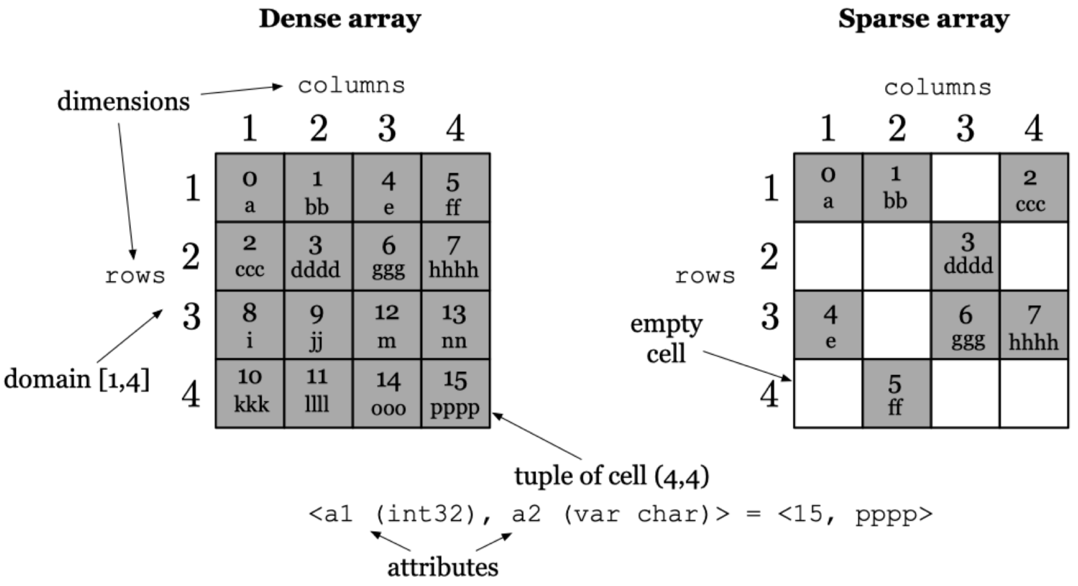
Introduction of TileDB

- Data Model
- Global cell order
- Data tiles
- Compression
- Fragments
- Array metadata
- System architecture

Introduction of TileDB

- Data Model
  - dimensions
  - attributes
  - dense: only int dimensions
    - i.e. image modeled by 2D dense array
  - sparse: int or float dimensions
    - as TileDB materilizes the coordiniates of the non-empty cells
    - i.e. geo-locations

Introduction of TileDB



Introduction of TileDB

Global cell order

Mapping from multiple dimensions to a linear order

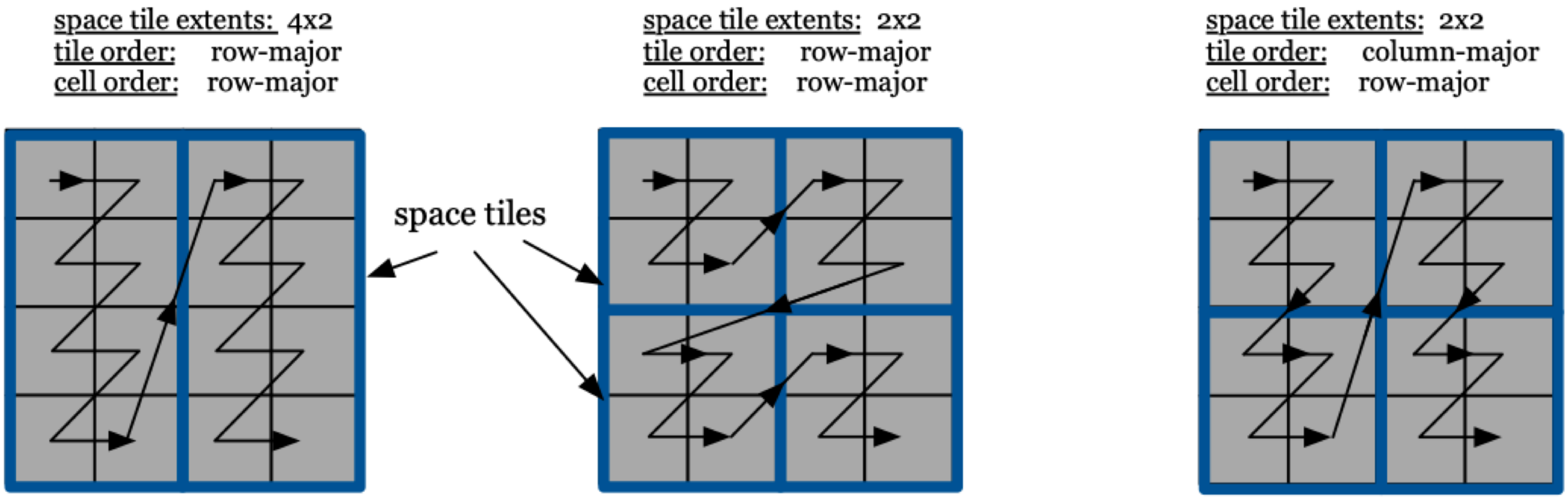


Figure 2: Global cell orders in dense arrays

Introduction of TileDB

3 steps to specified global cell order in dense array:

- Decompose the domain into space tiles
- Determine the cell order within each space tile
  - row-major
  - column-major
- Determine the tile order

Introduction of TileDB

For sparse array:

creating sparse tile is complex

- ➔ many empty tiles
  - ◆ tiles of highly varied capacity
  - ◆ ineffective compression
  - ◆ bookkeeping overheads
  - ◆ small tiles wasting seeking time

Introduction of TileDB

Data tile: a group of non-empty cells

For dense array: each data tile has a one-to-one mapping to a space tile

For sparse array

- determine a capacity of each data tile (i.e capacity = c)
- create one data tile for every c non-empty cells

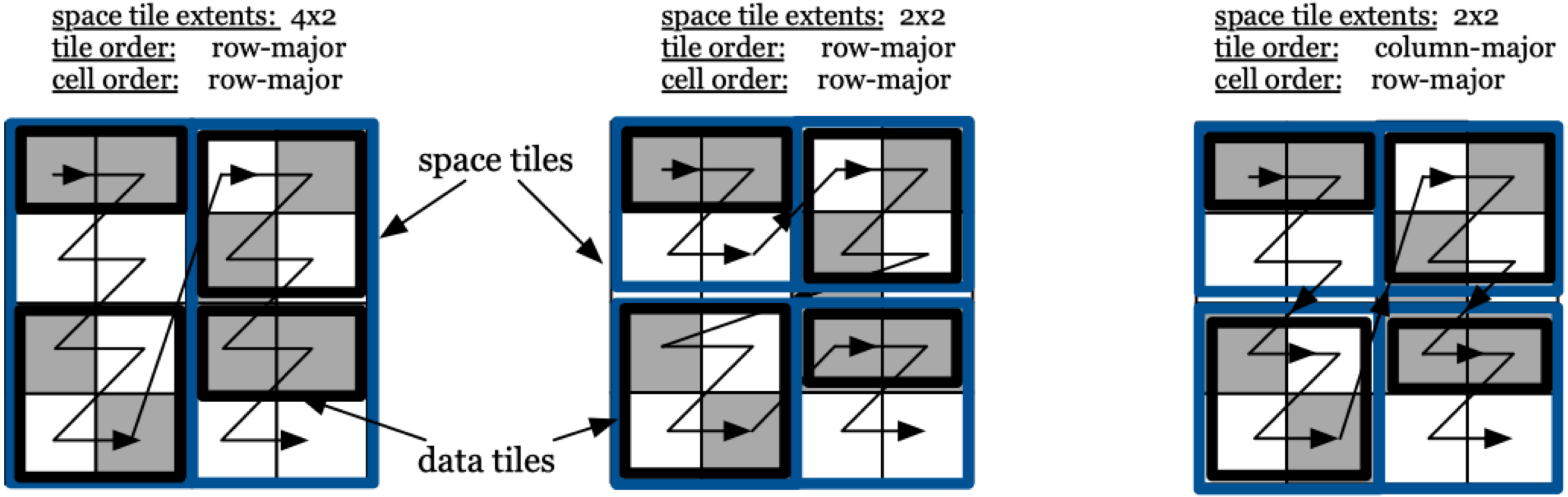


Figure 3: Data tiles in sparse arrays

Introduction of TileDB

Fragment

a timestamp of snapshot of batch of array update

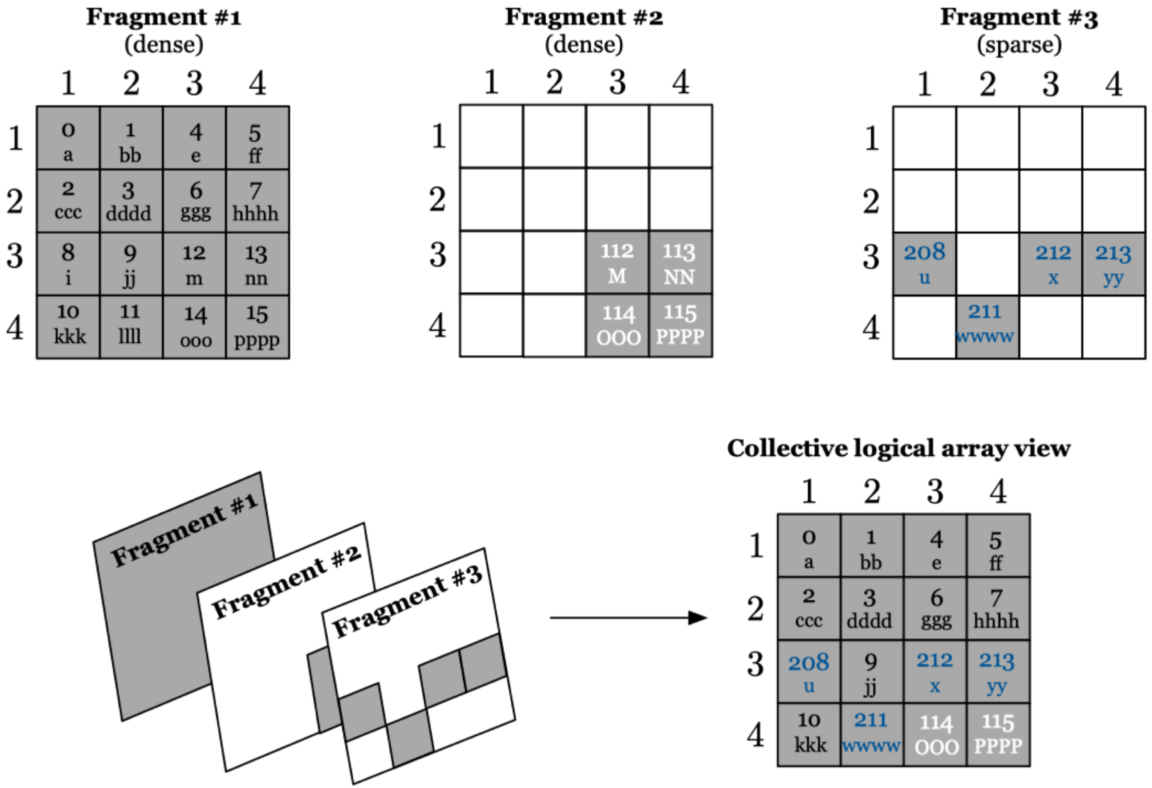


Figure 4: Fragment examples

Introduction of TileDB

Fragment is a key concept enables TileDB perform rapid writes

- ➢ If numerous fragments produces (bad for read performance)
  - Then TileDB consolidates them into a single one
  - Happening in parallel in the background
  - Reads and writes continue processing

Introduction of TileDB

Array metadata

- array schema and fragment bookkeeping
  - definition of array (name, number, name and types of dimensions and attributes, the dimension domain...)
  - the later summarizes information about the physical organization of the stored array data in a fragment

Introduction of TileDB

System architecture

- init
- write
- read
- conslidate
- finalize

Physical Organization

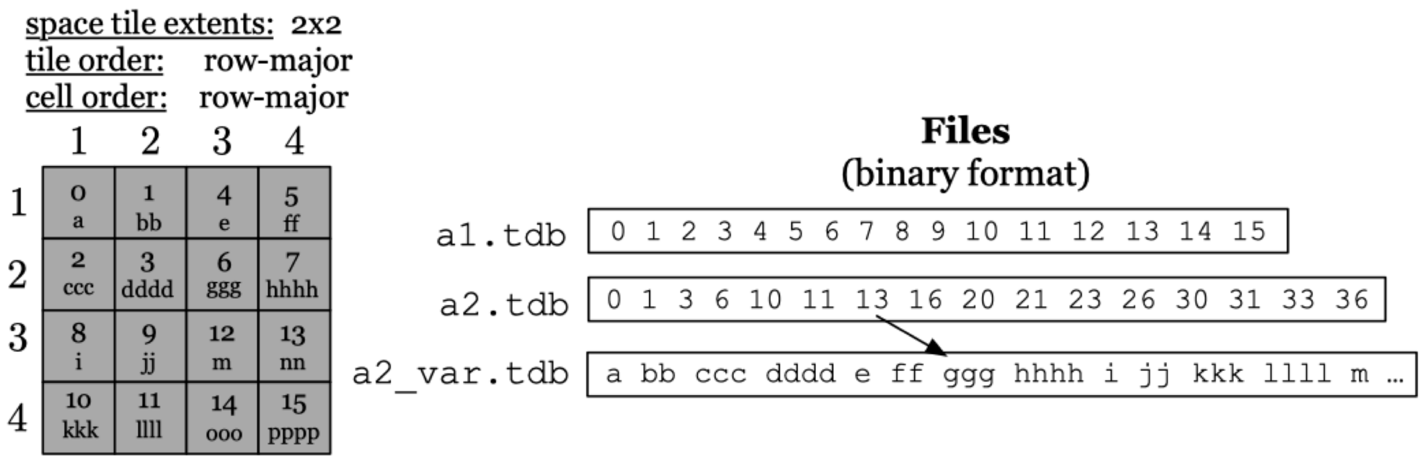


Figure 6: Physical organization of dense fragments

Physical Organization

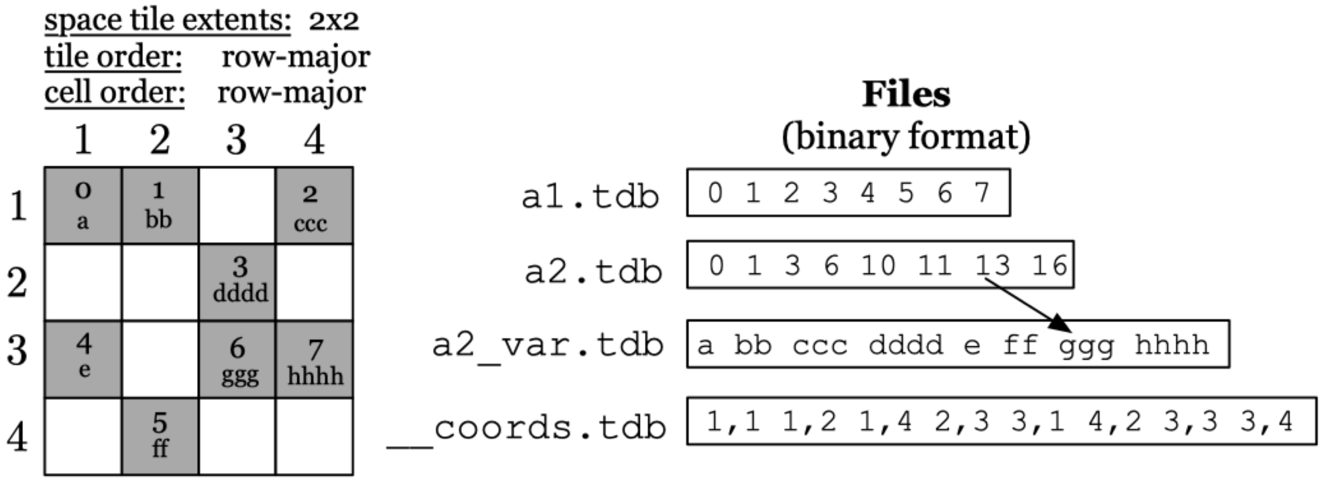


Figure 7: Physical organization of sparse fragments

Core Functions of TileDB

- Read
  - dense fragment
  - sparse fragment
- Write
  - dense fragment
  - sparse fragment
- Consolidate

Core Functions of TileDB

Read

- read returns the values of any subset of attributes inside a user supplied subarray
- result is sorted on the global cell order
- user specifies the subarray and attributes in the init call
- TileDB load **bookkeeping data** of array fragments into main memory
  - for dense case: negligible
  - for sparse case: depends on the tile capacity

Core Functions of TileDB

Read

issue: for variable length attributes and sparse array, the result size is unpredictable

solution:

- ➢ If exceeding the size of some buffers, TileDB fills in data into buffers and returns
- ➢ user can consume the result, and invoking read to resume process

Core Functions of TileDB

Read

Main Challenge:

- the presence of multiple fragments in the array
- read can not search each fragment individually

TileDB read algorithm (dense and sparse):

- efficiently access all fragments
- skipping unqualified data

Core Functions of TileDB

Read algorithm for dense array:

- first stage: computes a sorted list of tuples of the form <[sc, ec], fid>
- second stage: retrieves the actual attribute values from the respective fragment files

Core Functions of TileDB

<[sc, ec], *fid*>:

[sc, ec]: range of cells between start coordinates sc and end coordinates ec

fid: a fragment id, based on timestamp

Core Functions of TileDB

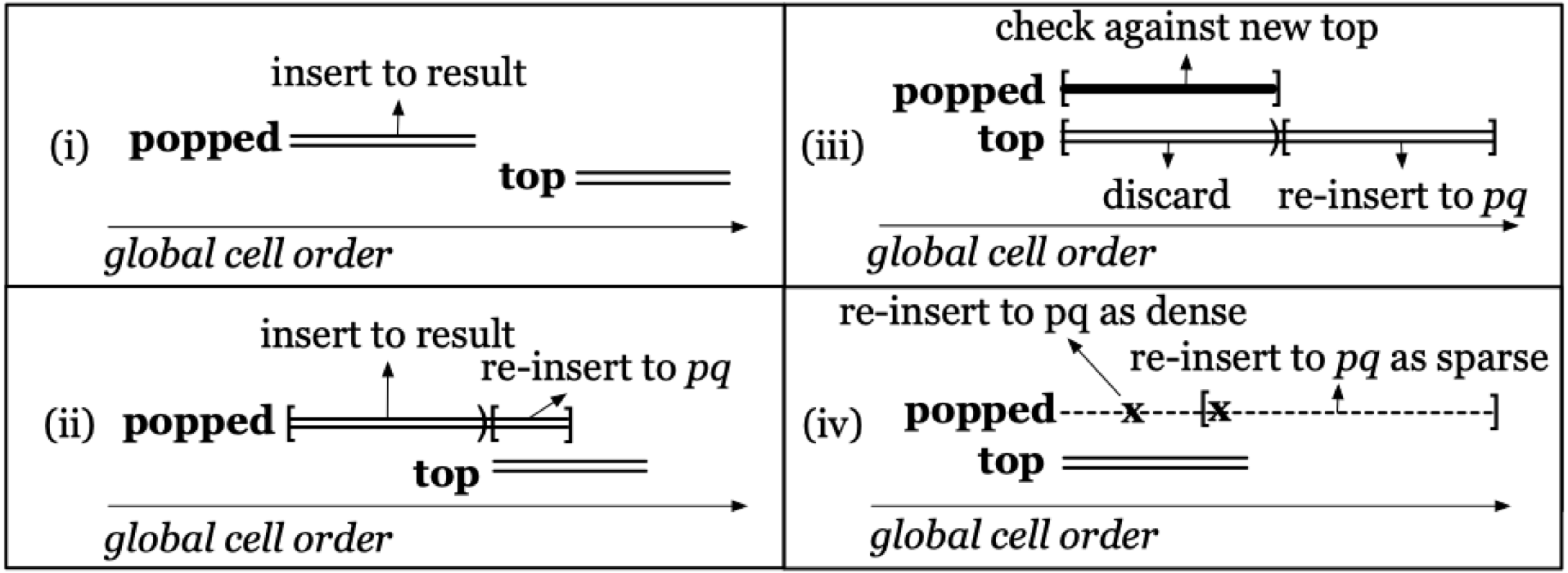
for fist stage:

- all ranges must be disjoint
- the ranges must be sorted in the global cell order
- the ranges in the ordered list must contain all and only the actual, up-to-date result cells
- the cells covered in each range must appear contiguously on the disk

Core Functions of TileDB

- ➢ creates on tuple <[sc, ec], *fid*> , and insets them into a priority queue *pq*
- ➢ the comparator of *pq* gives precedence to the tuple with smallest value
- ➢ breaking ties: the tuple with largest *fid*
- ➢ pops a tuple at a time from *pq*(called popped)
- ➢ compares popped to the new top tuple
- ➢ emitting new result tuples for second stage to consuming and reinserting tuples into *pq*

Core Functions of TileDB



Core Functions of TileDB

Read algorithm for sparse fragment

2 differences:

- iteration does not focus on space tile, but focus on ranges
  - start before minimum
  - end bounding coordinate of a data file
- case iii never arises, since the sparse array consist only of sparse fragment

Core Functions of TileDB

Write:

- writes session write cells sequentially in batches, createing a separate fragment
- begins when an array is initialized in write mode(with init)
- terminates when the array is finalized(with finalize)

Core Functions of TileDB

Write algorithm for dense fragment:

- Upon initialization, user specifies the subarray region in which the dense fragment is constrained
- then user populates one buffer/array attribute
- storing the cell value in global cell order

Core Functions of TileDB

write function:

- simply appends the values from buffers into the corresponding attribute files
- writing them sequentially
- without requiring additional internel buffering

Core Functions of TileDB

Write algorithm for sparse fragment

3 differences with dense case:

- provide value only for non-empty cells
- user includes an extra buffer with the coordinates of the non-empty cells
- TileDB maintains some extra write state info for each created data tile
    - counts number of cells
    - stores minimum bounding rectangle and bounding coordinate of data tile

Core Functions of TileDB

random updates arrive ar the system:

TileDB enable users to provide **unsorted cell** buffers to write

- ➢ sort the buffer internally
- ➢ then proceed for the sorted case

main difference:

Each write call in this mode creates a seperate fragment

Core Functions of TileDB

Consolidate:

- takes a set of fragment as input and produces a single new output fragment
- simply repeated perform a read on entire domain
- providing buffers depends on the avaliable main memory
- after every read, write command has been invoked
- stop reading when the buffers are full

Core Functions of TileDB

in read fragments:

any of them are **dense**: the consolidated fragment is dense

all of them are **sparse**: the consolidated fragment is sparse

Core Functions of TileDB

suggestion:

Consolidation should be applied on fragments of approximately eqaul size

Parallel Programming

- Concurrent Reads
- Concurrent Writes
  - multiple process
  - multiple threads
- Concurrent Read and Write

Parallel Programming

- Concurrent Read and Write

  fragments not-visible to reads finalized → visible

- Locks --Consolidation old fragments deleted
  new become visible

  Reads Shared Lock → Exclusive lock

Experiments

3 Competitors

HDF5	SciDB	Vertica
v1.10.0	v15.12	v7.02.0201
RLE		

Experiments

Dense--synthetic 2D arrays

int32 i*#col+j

Sparse--AIS database

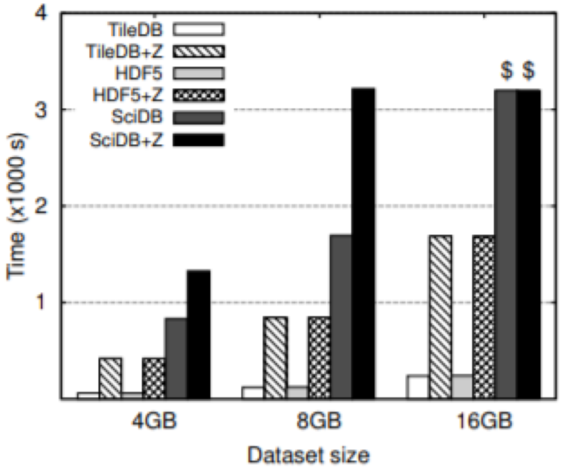
Experiments

Dense	Arrays
HDF5	SciDB

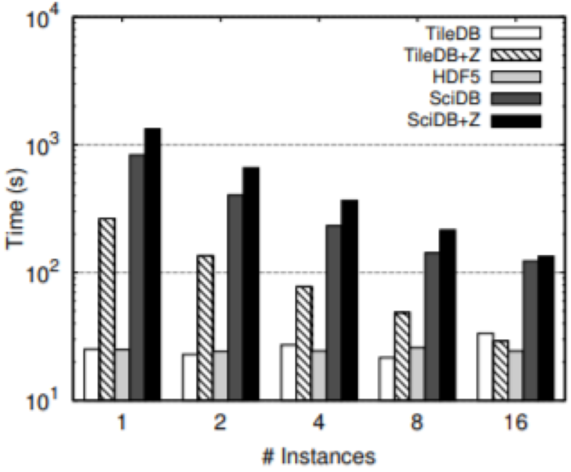
Experiments

Load

One CPU Core



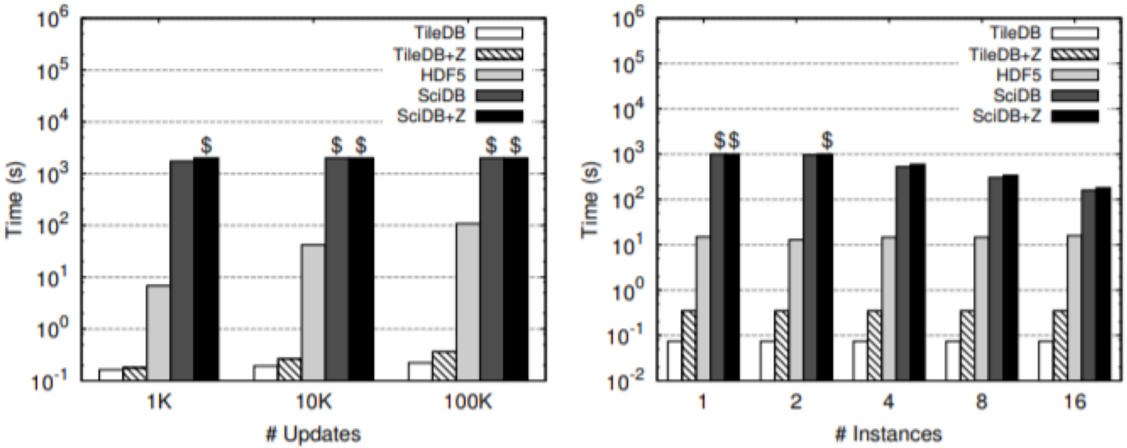
(a) vs. dataset size (HDD)



(b) vs. # instances (SSD)

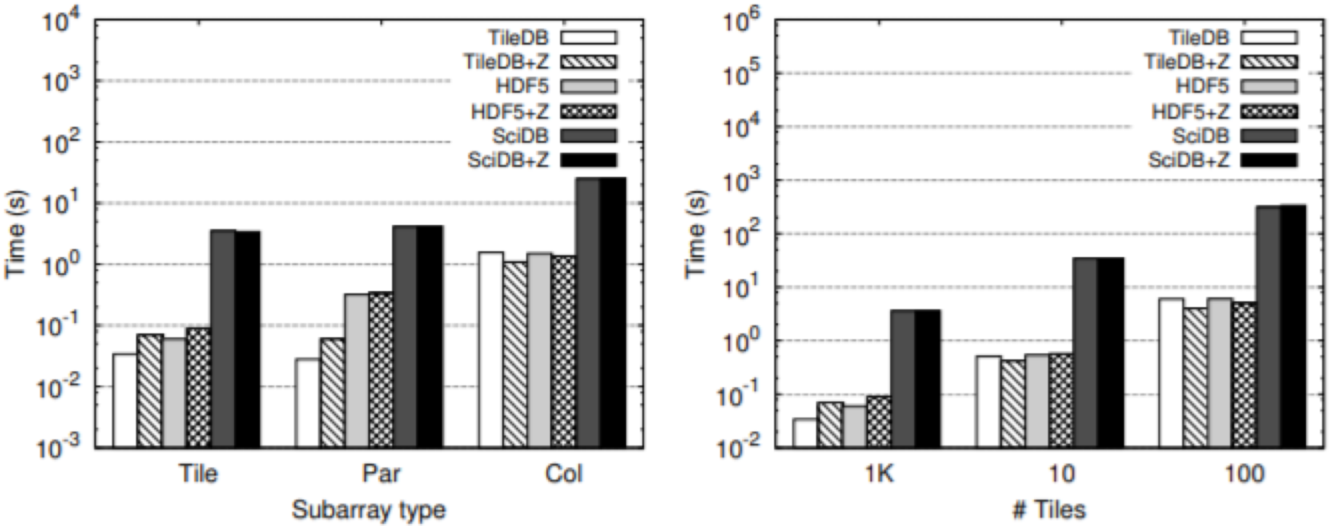
Experiments

Update



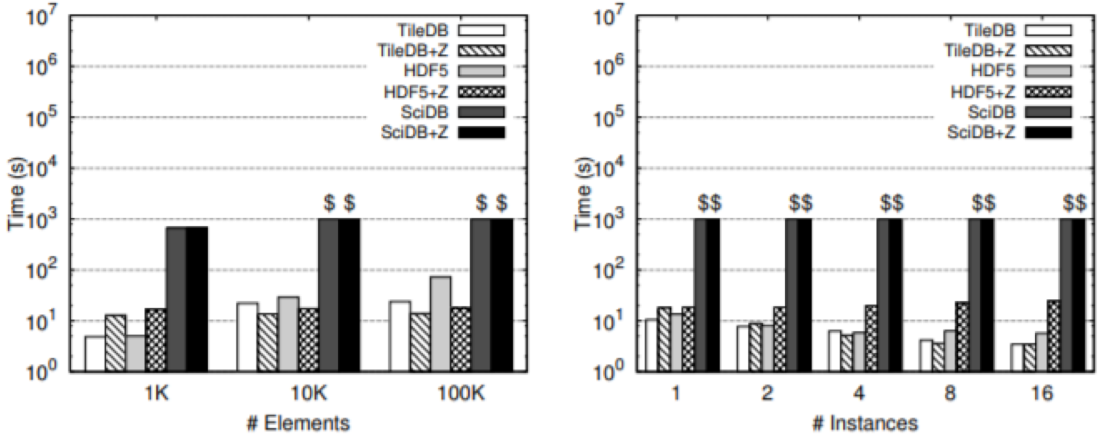
(a) vs. # updates (HDD) (b) vs. # instances (SSD)

Experiments



subarrays

Experiments

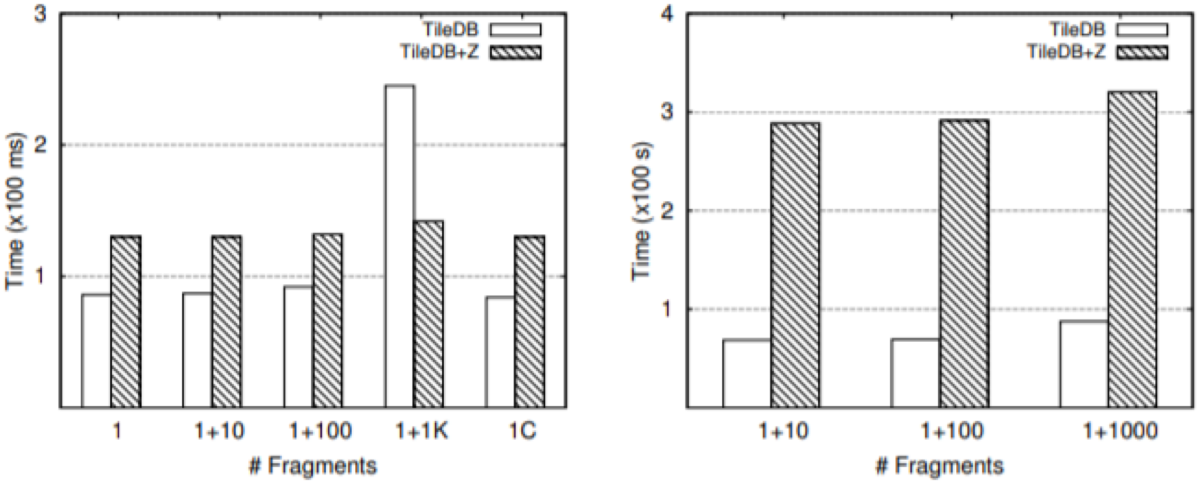


(c) vs. # elements (HDD) (d) vs. # instances (SSD)

Figure 11: Subarray performance for dense arrays

subarrays

Experiments



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

Figure 12: Effect of # fragments in dense arrays

#fragments

consolidation

Experiments

Scalability

two large arrays with sizes 128 GB and 256 GB

1,815.78 s and 3,630.89 s

Subarray queries 80 ms and 84 ms, 75 ms

unaffected by the array size

the memory consumption upon loading negligible.

Experiments

Vertica

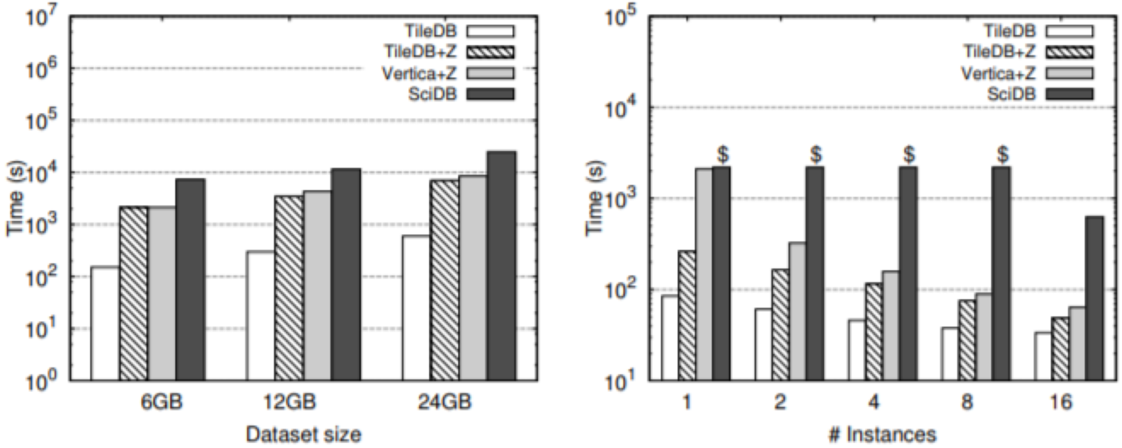
GZIP and RLE

TileDB 2x-40x better in all settings

Experiments

Sparse	Arrays
Vertica+Z	SciDB

Experiments

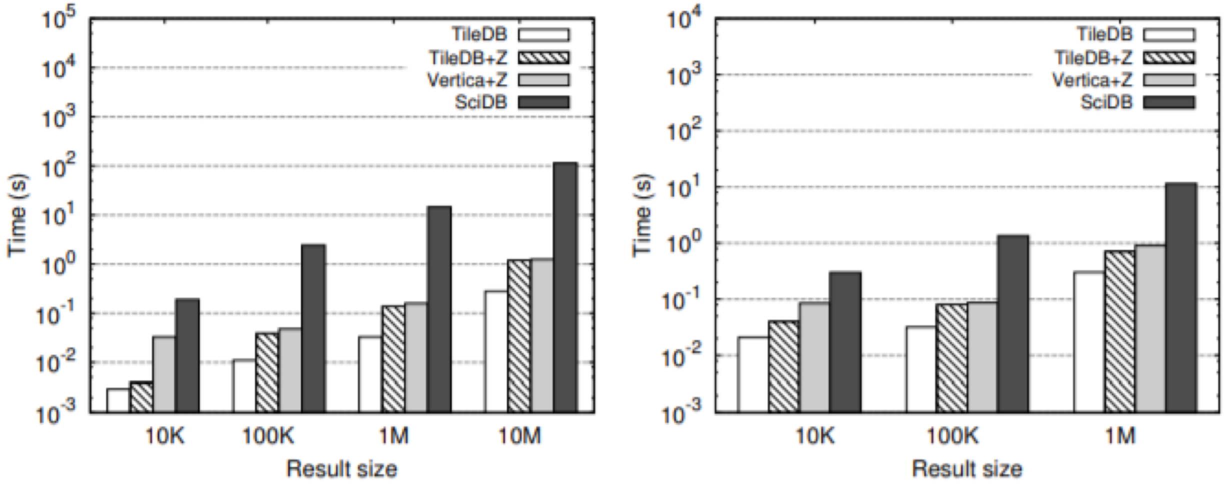


(a) vs. dataset size (HDD) (b) vs. # instances (SSD)

Figure 13: Load performance of sparse arrays

Load

Experiments

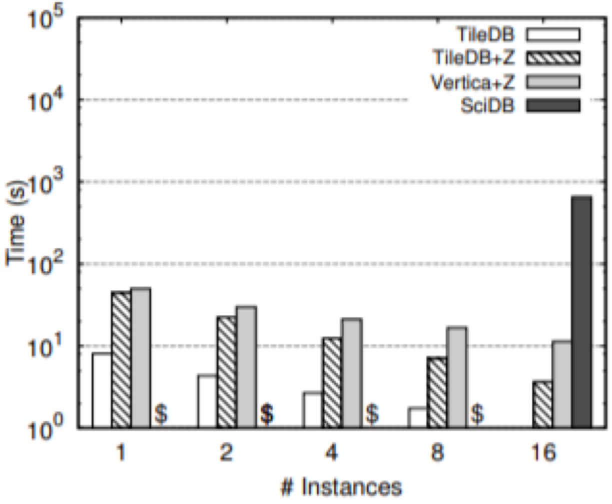


(a) DQ vs. result size (HDD) (b) SQ vs # result size (HDD)

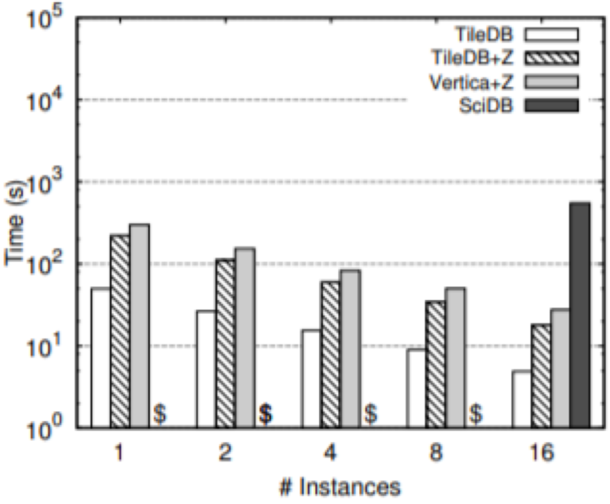
Subarray

2 array regions

Experiments



(c) DQ vs. # instances (SSD)



(d) SQ vs. # instances (SSD)

Subarray

Experiments

Consolidation random new cells

deteriorates 18% after inserting 100 fragments,

2x after 1000 fragments,

normal after consolidation

#Same as original Load

Conclusion

HDF5-- Better performance

SciDB-- Better in all settings

Vertica-- Equivalent performance on sparse arrays

More friendly API

Key Factors

Arrays → dense and sparse

Space tiles → shape and size MBR

Tile capacity→ number of cells

Dimensions → no subselection

Filtering (Compression)

Thanks for watching