The TileDB Array Data Storage Manager

Manish Patel

CS 561 - Class 21

April 13, 2021

Motivation

- Scientific and engineering data → multi-dimensional arrays
- Either <u>dense</u> or <u>sparse</u>

Dense Matrix

1	2	31	2	9	7	34	22	11	5
11	92	4	3	2	2	3	3	2	1
3	9	13	8	21	17	4	2	1	4
8	32	1	2	34	18	7	78	10	7
9	22	3	9	8	71	12	22	17	3
13	21	21	9	2	47	1	81	21	9
21	12	53	12	91	24	81	8	91	2
61	8	33	82	19	87	16	3	1	55
54	4	78	24	18	11	4	2	99	5
13	22	32	42	9	15	9	22	1	21

Sparse Matrix

1	.	3	.	9	.	3	.	.	.
11	.	4	.	.	.	.	.	2	1
.	.	1	.	.	.	4	.	1	.
8	.	.	.	3	1	.	.	.	.
.	.	.	9	.	.	1	.	17	.
13	21	.	9	2	47	1	81	21	9
.	.	.	.	.	.	.	.	.	.
.	.	.	.	19	8	16	.	.	55
54	4	.	.	.	11	.	.	.	.
.	.	2	.	.	.	.	22	.	21

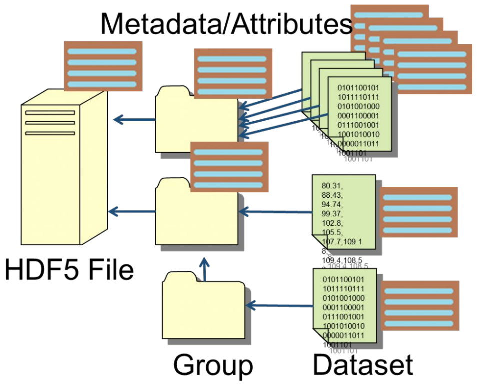
https://i1.wp.com/cmdlinetips.com/wp-content/uploads/2018/03/Sparse_Matrix.png?resize=576%2C324

Motivation (continued)

- Difficulty in storing the expansive array data
  - Maintaining efficient read and writes
- Need for array data storage management systems → Efficient data access primitives

Current Approaches

- HDF5: dense array format, grouped into chunks
  - Library in C for storage management tasks
  - Datasets: array elements and metadata
  - Groups: multiple datasets with their own metadata



https://icmplus.neurosurg.cam.ac.uk/wp-content/uploads/2017/10/HDF5_2-1024x0-c-default.png

Current Approaches (continued)

- Drawbacks of HDF5
  - Inefficient for sparse arrays
  - Small, random in-place writes/updates
- Drawbacks of Parallel HDF5
  - No concurrent writes to compressed data
  - No variable-length elements

https://upload.wikimedia.org/wikipedia/commons/thumb/a/a0/HDF_logo.svg/1200px-HDF_logo.svg.png

Current Approaches (continued)

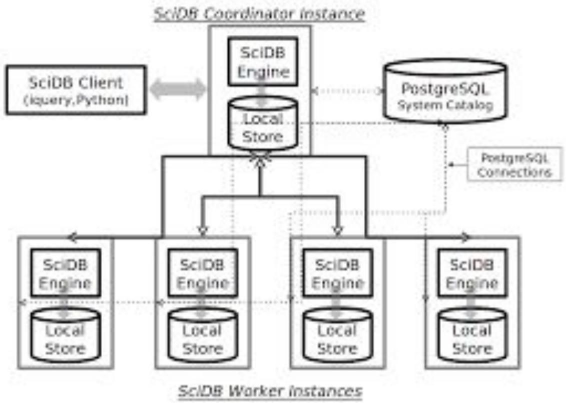
- Need for optimization for random updates of small blocks
- SciDB: array database
  - Similar chunking as HDF5
  - Reading and updating entire chunks
- ArrayStore
  - Optimizing for sparse arrays
  - Persisting issues

SciDB

https://dbdb.io/media/logos/scidb.png.280x250_q85.jpg

Current Approaches (continued)

- SciDB
  - Shared-nothing architecture
    - Parallelized and distributed
  - Vertically partitioned chunks
  - “No-overwrite” storage
  - ACID transactions
    - Array-level locking



https://www.nersc.gov/assets/Uploads/scidb-userguide-12.3.pdf

Current Approaches (continued)

- Relational databases
  - Store non-null elements as records
  - Maintaining element indices as columns
  - Inefficient for dense arrays

https://upload.wikimedia.org/wikipedia/en/b/b9/Monetdb-logo.png

https://dbdb.io/media/logos/vertica.png

TileDB - Overview

- First array storage manager optimized for dense *and* sparse arrays
- Elements of arrays organized into <u>fragments</u>

https://dbdb.io/media/logos/tiledb.png

A Look at Arrays

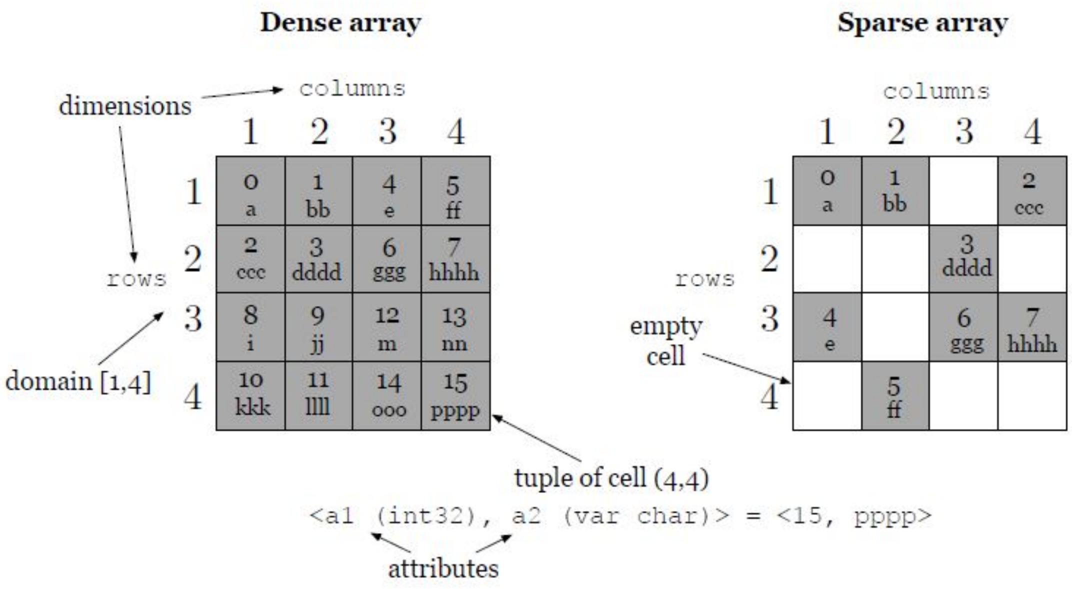


Figure 1: The logical array view

- Stored in sparse format if some threshold of the cells are empty/null

Examples of Uses of Arrays

- Imaging Application:
  - Dense 2-D array
  - Each cell with RGB attributes
- Geo-tagged Tweets:
  - Sparse 2-D array
  - Geographical coordinates as floats
  - Tweets as variable-length char

Global Cell Order for TileDB

- Mapping from multiple dimensions to linear
- Dependent on how each application would use the data

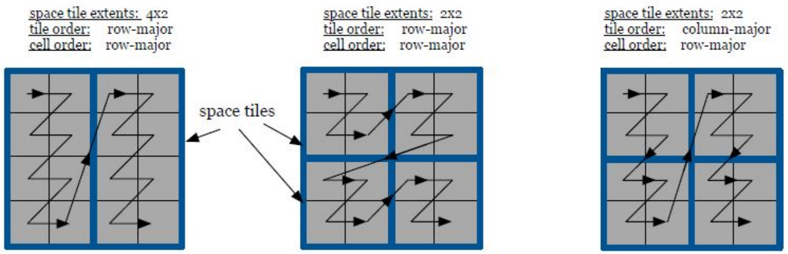


Figure 2: Global cell orders in dense arrays

← Define tile extent, cell order within space tile, and tile order

Data Tiles

- Sparse arrays in the same way → empty tiles
- Instead, group the non-empty cells
- Traverse in the global cell order

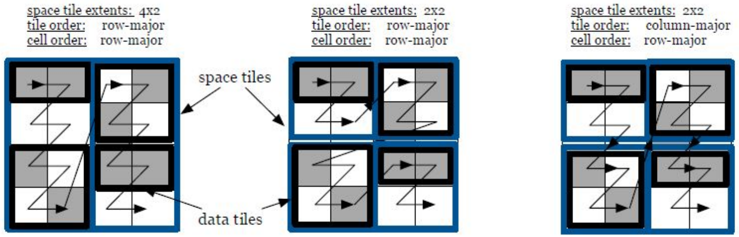


Figure 3: Data tiles in sparse arrays

← Specify a data tile capacity (e.g. 2 here), form minimum bounding rectangles (MBRs)

Fragments

- Snapshot of batched array updates at given time
- Collectively form the current logical array
- Allow for efficient writes
- If reads are affected, consolidation is performed
    - Merge fragments into one

Fragments (continued)

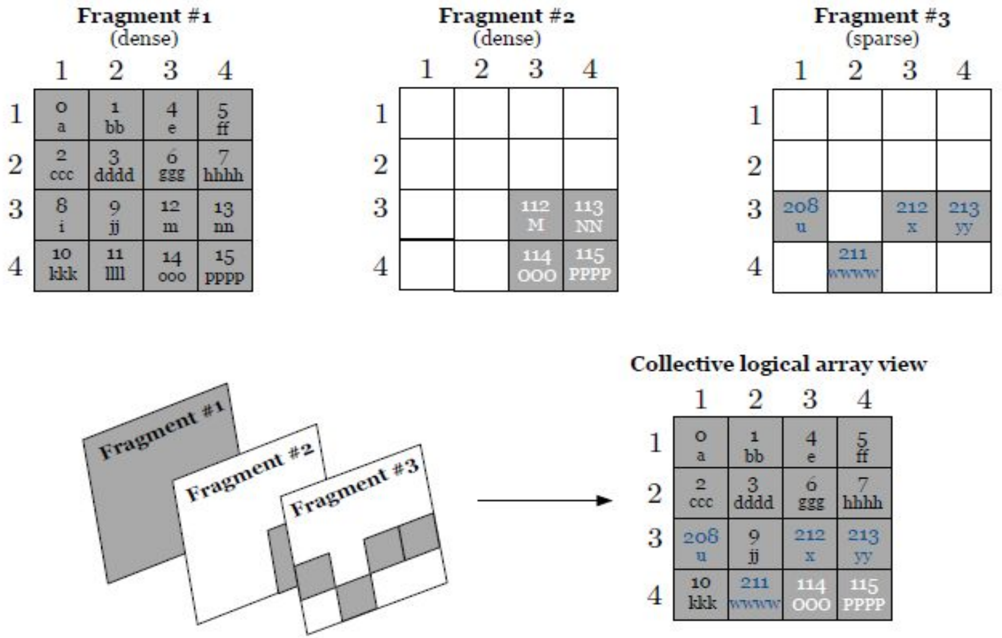


Figure 4: Fragment examples

Physical Organization

- Array stored as a directory with subdirectories for each fragment with files for each attribute (in global cell order)
- Bookkeeping metadata about MBRs and bounding coordinates (useful for reads)

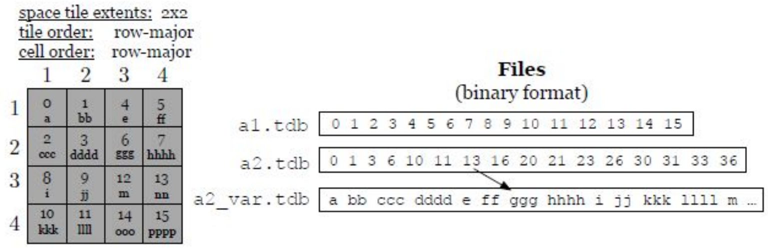


Figure 6: Physical organization of dense fragments

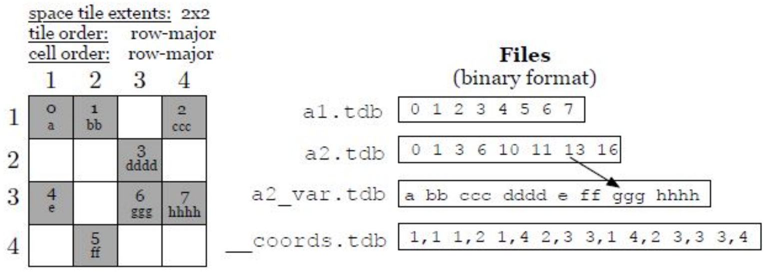


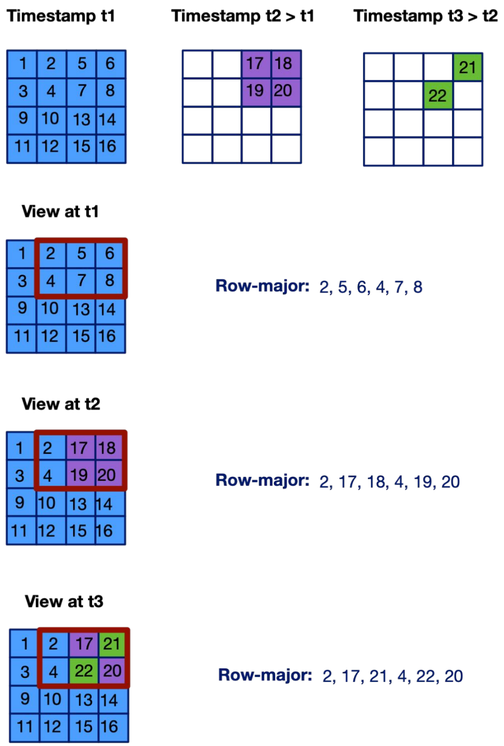
Figure 7: Physical organization of sparse fragments

READ Operations

- Buffers allocated to store results
- Challenge of having multiple fragments
- Importance of global cell order
    - More efficient operation on single-dimension

READ Operations - Dense arrays

- 1) Compute sorted list of tuples containing:
    - start coordinates and end coordinates
    - A fragment ID
  - Iterate through the space tiles
- 2) Retrieve the attribute values from the fragment files



https://docs.tiledb.com/main/solutions/tiledb-embedded/internal-mechanics/reading

READ Operations - Sparse arrays

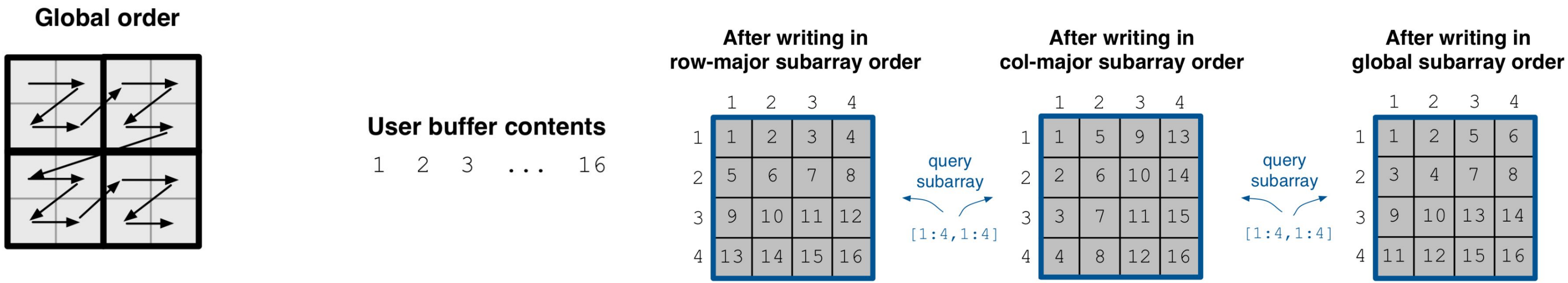
- Differences in step 1:
  - Iterations on ranges involving minimum bounding coordinate of a data tile in a fragment, instead of space tiles
  - One of the overlap cases never occurs

WRITE Operations

- Loading and updating data
- Done in batches
- Forming a new fragment
    - Can be initialized as dense or sparse

WRITE Operations - Dense Fragments

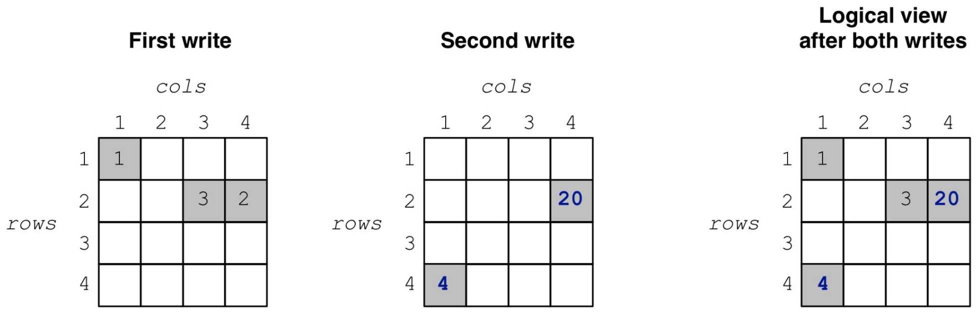
- Specify subarray region for fragment
- User fills a buffer for each array attribute in global cell order
- Appends buffer values into attribute files



https://tiledb-inc-tiledb.readthedocs-hosted.com/en/1.6.3/_images/writing_dense_layout.png

WRITE Operations - Sparse Fragments

- 1) Filling buffers with values for non-empty cells only
  - Extra buffer for coordinates of non-empty cells
- 2) Random updates with unsorted cell buffers
  - Separate fragments for each write
- Deletions by inserting empty cells

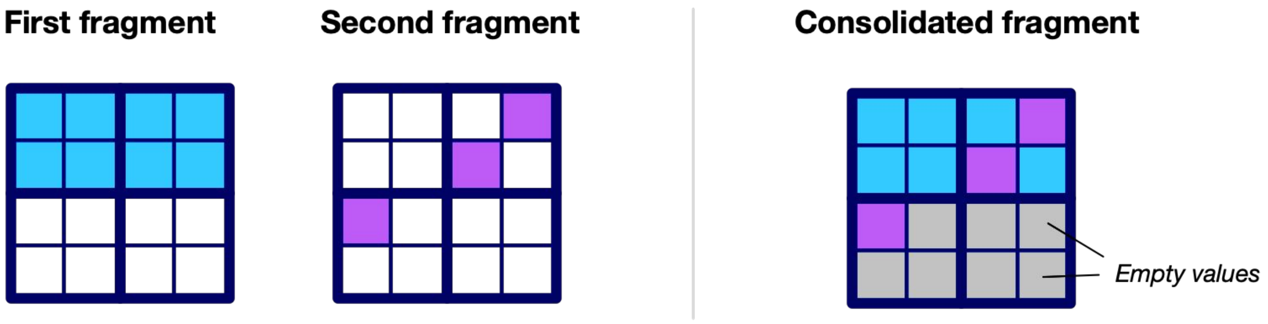


https://tiledb-inc-tiledb.readthedocs-hosted.com/en/1.6.3/_images/writing_sparse_multiple.png

CONSOLIDATE Operation

- Forming a single fragment from multiple
- Performed with repeated READ operations and writing into the output fragment
- TileDB allows for consolidation on only a subset of fragments

CONSOLIDATE Operation (continued)



https://docs.tiledb.com/main/solutions/tiledb-embedded/internal-mechanics/consolidation

Parallel Programming

- Concurrent reads and writes
  - No locking necessary
- Thread/process-safety
- Atomic reads and writes
- Background consolidation
  - Locking only needed upon completion

Experimental Performance

- Competitors:
  - HDF5/Parallel HDF5, SciDB, and Vertica
- System configuration:
  - Intel x86_64 platform with a 2.3 GHz 36-core CPU and 128 GB of RAM, running CentOS6
  - 4TB, 7200 rpm Western Digital HDD
  - 480GB Intel SSD

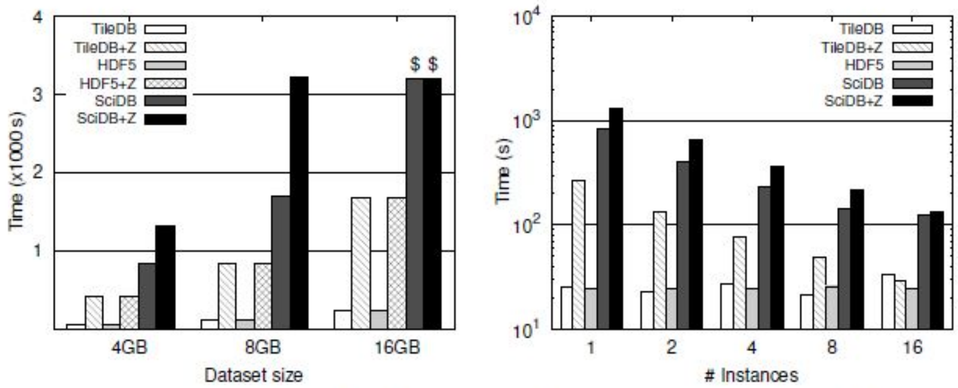
Experimental Performance (continued)

- Datasets Used
  - Dense arrays:
    - Synthetic 2-D arrays with an int attribute
  - Sparse arrays:
    - Data collected by National Oceanic and Atmospheric Administration for ships
    - Geographical coordinates as dimensions

Experimental Performance (continued)

Loading Dense Arrays

- TileDB matches HDF5 and outperforms SciDB by several orders of magnitude



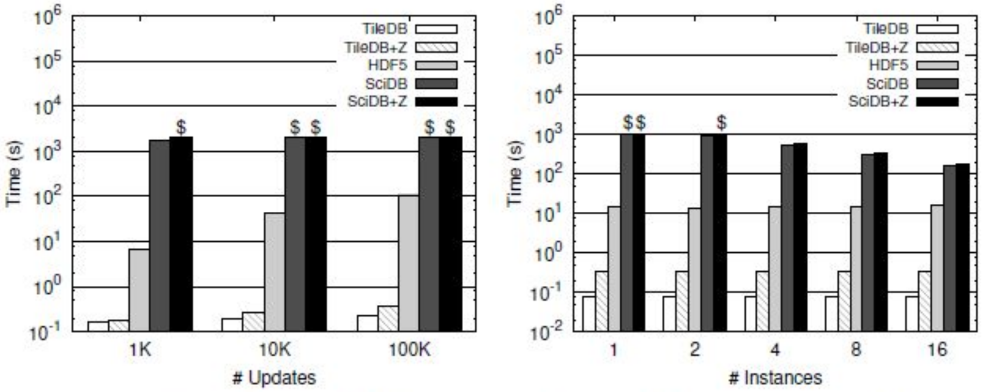
(a) vs. dataset size (HDD) (b) vs. # instances (SSD)

Figure 9: Load performance of dense arrays

Experimental Performance (continued)

Updating Dense Arrays

- TileDB performs > 2x faster than HDF5 and > 4x faster than SciDB
- Sequential, fragment-based writes



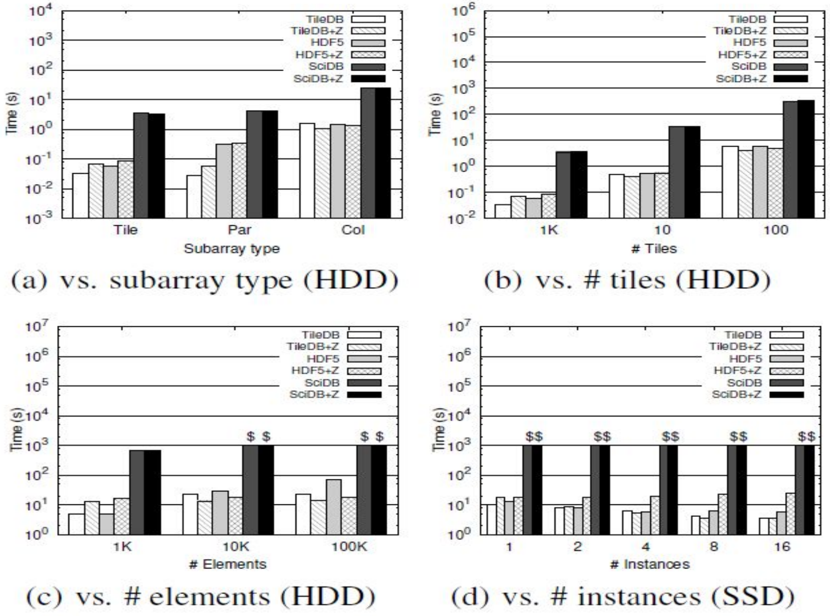
(a) vs. # updates (HDD) (b) vs. # instances (SSD)

Figure 10: Random update performance of dense arrays

Experimental Performance (continued)

Reading Dense (Sub)Arrays

- TileDB either matches or outperforms HDF5 and outperforms SciDB
- Scaling with # tiles
- Unaffected by array size



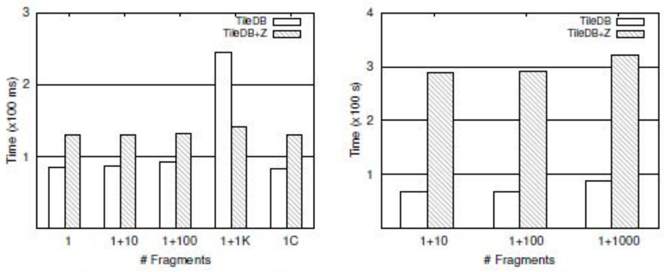
(a) vs. subarray type (HDD) (b) vs. # tiles (HDD)

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

Figure 11: Subarray performance for dense arrays

Experimental Performance (continued)

- Number of fragments → consolidation
  - Read performance worsens as more fragments are created
  - Efficiency returns after consolidation
  - Consolidation time is largely the same



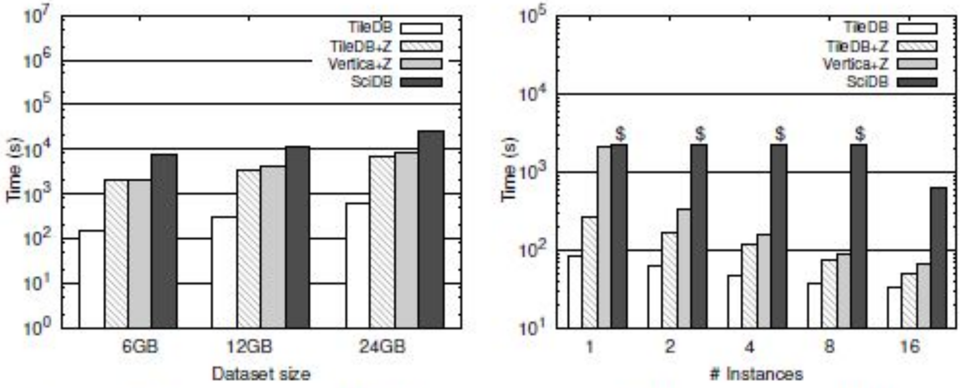
(a) Subarray time (HDD) (b) Consolidation time (HDD)

Figure 12: Effect of # fragments in dense arrays

Experimental Performance (continued)

Loading Sparse Arrays

- TileDB outperforms SciDB by more than an order of magnitude



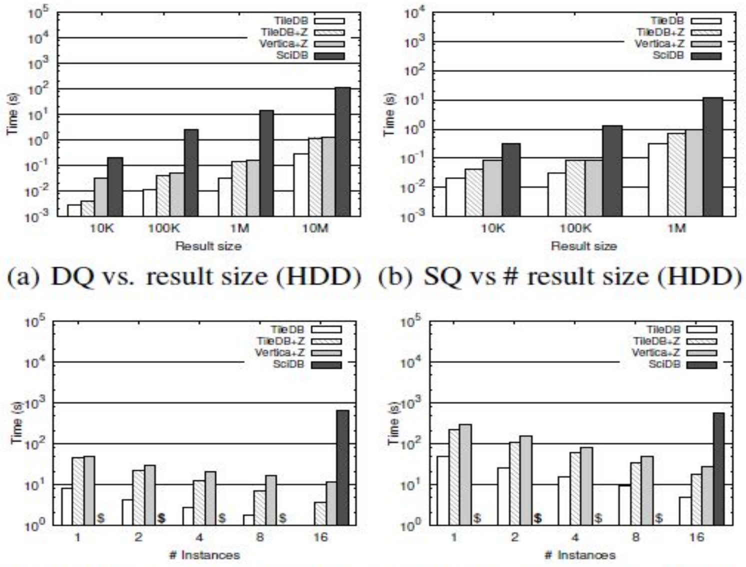
(a) vs. dataset size (HDD) (b) vs. # instances (SSD)

Figure 13: Load performance of sparse arrays

Experimental Performance (continued)

Reading Sparse (Sub)Arrays

- TileDB is 1-2 orders of magnitude faster than SciDB and essentially matches Vertica
- Favorable scaling



(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

Conclusion

- TileDB optimized for dense and sparse arrays
- Much more efficient random writes than HDF5, and similar read performance (dense)
- Far outperforming SciDB for both types
- Similar performance as Vertica (sparse)
- Optimal scaling for dataset size and level of parallel programming

Strengths and Weaknesses of the Paper

Strengths:

- Very thorough experimentation on all types of operations and both types of arrays
- Useful implementation of visuals for characteristics of TileDB set-up

Weaknesses:

- Lacking in visual depictions for the operations
  - Hard to comprehend from the lengthy written explanations

Future Work/Improvements

- Still an active project → www.tiledb.com
  - Implemented in C++
- Possible implementation for storing matrices and performing matrix operations
  - Array computations

TileDB GitHub Repo

README.md

[tile] DB

Azure Pipelines succeeded | Anaconda downloads 1M total

The Universal Storage Engine

TileDB is a powerful engine for storing and accessing **dense and sparse multi-dimensional arrays**, which can help you model any complex data efficiently. It is an embeddable C++ library that works on Linux, macOS, and Windows. It is open-sourced under the permissive MIT License, developed and maintained by TileDB, Inc. To distinguish this project from other TileDB offerings, we often refer to it as *TileDB Embedded*.

TileDB includes the following features:

- Support for both **dense** and **sparse arrays**
- Support for **dataframes** and **key-value stores** (via sparse arrays)
- **Cloud storage** (AWS S3, Google Cloud Storage, Azure Blob Storage)
- **Chunked** (tiled) arrays
- Multiple **compression**, **encryption** and **checksum** filters
- Fully **multi-threaded** implementation
- **Parallel IO**
- **Data versioning** (rapid updates, time traveling)
- Array **metadata**
- Array **groups**
- Numerous **APIs** on top of the C++ library
- Numerous **integrations** (Spark, Dask, MariaDB, GDAL, etc.)

You can use TileDB to store data in a variety of applications, such as Genomics, Geospatial, Finance and more. The power of TileDB stems from the fact that any data can be modeled efficiently as either a dense or a sparse multi-dimensional array, which is the format used internally by most data science tooling. By storing your data and metadata

https://github.com/TileDB-Inc/TileDB

References

Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. 2016. The TileDB array data storage manager. *Proc. VLDB Endow.* 10, 4 (November 2016), 349–360. DOI: https://doi.org/10.14778/3025111.3025117