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 hyperrectangles, 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
Copyright 2016 VLDB Endowment 2150-8097/16/12.