Performance of Null Handling in Array Databases

1st Dimitar Mišev
School of Computer Science & Engineering
Constructor University
Bremen, Germany
dmisev@constructor.university

2nd Mikhail Rodionychev
School of Computer Science & Engineering
Constructor University
Bremen, Germany
mrodionych@constructor.university

3rd Peter Baumann
School of Computer Science & Engineering
Constructor University Bremen, Germany
pbaumann@constructor.university

2023 IEEE International Conference on Big Data (BigData) | 979-8-3503-2445-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/BigData59044.2023.10386100

***Abstract*—Array databases specialize in storage, management, and query processing on massive multidimensional array data such as satellite image timeseries, weather forecast models, IoT sensor measurements, medical imaging data, etc. Some of the values in this data may be "null" for a variety of reasons, such as unknown, known false, etc. The DBMS must handle null values correctly and efficiently.**

As of today, the effects of different data structures for representing null values on query processing in array databases have not been systematically studied. As a consequence, it is not clear what the optimal way for handling null values is.

In this paper, we measure how four different methods for encoding null values perform across most common categories of array operations. The result is a comprehensive overview, publicly available in open source, providing relevant insights into the performance characteristics of these data structures allowing to discern which among them offers the most optimal approach for representing null values in an array processing context.

***Index Terms*—null values, array databases, performance benchmark**

I. Introduction

Null is a special marker commonly used in databases to indicate non-existing data values or, more generally, data that should not be considered in computations such as aggregations. A variety of reasons exist for such missing values. In multidimensional geo data, for example, they can arise when a sensor detects a faulty reading, no value is delivered at all (maybe due to transmission problems), no value is applicable to a position (such as sea surface temperature measurements on dry land positions), or introduced through some common operations such as reprojection which warps the data grid.

While in many data structures, such as sets, it is sometimes enough to not store such data, they form a particular problem in arrays where, by definition, every cell inside the array has a value. Simply dropping that cell from the array could shift a lot of values into a wrong position. On implementation level this is also true most of the time, with some exceptions as we will see in the next section.

The term *dense array* characterizes arrays with relatively few or no null value within its spatial domain; in a *sparse array*, conversely, the null values dominate. For example, OLAP datacubes are generally sparse with some three to five percent non-nulls, whereas satellite images are usually dense as most pixels hold sensor values. Dense array have *some* value for every coordinate within the spatial domain, because for efficiency the coordinates are implicit and not separately materialized. Hence, even values which are null are present in the array data, and some mechanism for marking these values as null is necessary.

Operations on arrays, therefore, must take into account the null status of each value. The effects of null values on the result varies depending on the operation on hand. Based on that, operations can be aggregated into several more general categories, which form the basis for our benchmark.

Inevitably, the system performance additionally depends on the data properties and mode of operation execution, and not just what operations are performed. Data sparsity, i.e. the percentage of null values in the array, can have significant effects on performance characteristics depending on how the null values are encoded. Another aspect to consider is the data size, which can reveal differences between more and less compact null value representations. Finally, it is necessary to take into account how the execution of operations (serial vs. parallel) affects performance.

In this paper we present a systematic, scientifically rigorous benchmark of four methods for null handling: plain boolean arrays which are widely used [1]–[7], bit-packed version of boolean arrays with the standard C++ implementation and the CRoaring `bitset_t`, and Roaring compressed bitmaps [8]. All of the benchmark code has been published as open-source [9].

The paper is organized as follows. In the next section we inspect the state of the art of null value handling. Section III presents a comparative benchmark of several null encoding schemes under typical array operations. Results are discussed in Section IV. The plot concludes with Section V.

II. Related Work

A. Tools

A plethora of different specifications, standards and software implementations need to handle null values: from

databases, data processing tools and frameworks, to programming languages. In this paper we are particularly interested in null values in the context of array data, so we will focus on software in this space, especially array databases.

The category of array databases was pioneered by rasdaman ("raster data manager") being the first database system with a comprehensive archtecture for storage, management, and declaratively querying of massive multidimensional arrays [10], [11]. It is available as an open-source edition as well as an enterprise edition suitable for large-scale, federated commercial deployments. Both have generally the same interfaces, with the query languages based on the ISO SQL/MDA [12] and OGC WCPS [13] standards; internally, the two versions differ with respect to null value handling. In rasdaman community a null value set can be associated with a persistent array, which is then propagated through operations and checked while processing each of the operand(s) array values to determine the null values [14]. In rasdaman enterprise when an array is loaded from disk, a null mask is generated as a boolean array of same shape and size as the array [15].

SciDB is another major array DBMS with a full-stack implementation similar to rasdaman [16]. SciDB encodes the null status embedded in the array itself. The array data is split into variable segments which may be runs of the same value (a compression technique known as Run-Length Encoding, or RLE), non-RLE literal list of values, or runs of one or more null values [17].

TileDB is a library for storage and management of dense and sparse arrays [18], which encodes null status of array values as a separate validity array persisted alongside the array itself [2].

Across open-source relational databases there are two general variations depending on how they organize data, row and column-oriented architectures. PostgreSQL stores the null status in a separate bitmap per row of values [19], [20], and MySQL does the same in its InnoDB storage engine [21]. SQLite is only slightly different: a separate list of values encode the type of each field in a row, and one of the possible values is reserved for null [22]. Column-stores on the other hand are closer to array databases as they store table columns contiguously; all values in a column are of the same type, so the whole column can be thought of as an array. Vectorwise encodes null status of nullable columns as a separate boolean column, and rewrites queries to consider both columns during evaluation [6]. ClickHouse also stores the null status as a boolean array [7]. Apache Druid is similar, except it uses a bitmap corresponding to a particular column segment [23], as does DuckDB [24].

Support for nulls across data formats varies depending on how general they are. Apache Arrow is, at its core, an in-memory columnar format specifically designed for use in analytic database systems and data frame libraries (i.e. very applicable for array processing applications); nulls are encoded in bit vectors separately from the data columns [25], [26]. Apache Parquet, a columnar-oriented storage format, has native support for nulls which are stored separately from the data and RLE-compressed [27]. A similar storage format is Apache ORC, which stores and compresses a null bitmap separately from the data [28]. The NetCDF format for array-oriented scientific data allows specifying a _FillValue attribute which is used for unwritten data, and a valid_range interval bounding the valid values [29]. Other more image-oriented formats, such as TIFF, JPEG, or PNG, do not generally have a mechanism for embedding information about nulls. Software tools, such as GDAL, typically try to add a nodata status within the metadata or as a companion mask file [30]. ArcGIS Pro is popular GIS software that supports either storing a separate null mask, or adding a special NoData value which may trigger type promotion if the array contains all values in its type range [31].

In NumPy arrays that may have null values are called masked arrays. The null mask is a separate boolean array in which a true value indicates that the corresponding array value is a null value [32]. A MaskedTensor works in the same way in PyTorch [4]. TensorFlow's boolean_mask is similar, with the caveat that it can be of lower dimension than the corresponding tensor, which allows for encoding the null status of whole slices as well [5].

In MATLAB missing data are encoded inline, either as NaN values for floating-point arrays, or a special keyword for other data types [33]. NA (not available) values are similarly encoded inline in R, with a special double value for floating-point values, and C's INT_MIN for integer values [34].

B. Benchmarks

Multiple benchmarks of array databases and related systems have been published including [35]–[42], but none addresses null values. To the best of our knowledge this work represents the first benchmark focusing on null values in array databases. Therefore, we reach out further and inspect null benchmarking in relational databases and further tools.

The authors in [19] propose a more efficient storage format for very sparse relational data, and benchmark it against vanilla PostgreSQL null value encoding and a PostgreSQL modified with "positional" null value scheme (all fixed-size fields are materialized, including the null ones). This work is specific to row-oriented relational databases and highly sparse data.

Treatment of null values in the context of sparse data in column-store databases has been considered in [43]. Abadi extended C-Store with three methods for handling nulls depending on the column sparsity. In all cases the positions of non-NULL values are separately encoded, as a list of positions for very sparse data, a bitmap for intermediate sparsity, or position ranges for low sparsity. A comparison to naive C-Store implementation that materializes null values inline with the column data shows linear improvement as the sparsity increases. The benchmark is limited to a single query type that focuses on selecting data from disk.

In [44] the authors perform a performance and space-efficiency evaluation of two popular column-oriented storage formats, Parquet and ORC. The outcome is a set of recommendations on how future such formats can be designed to fit

better on modern hardware and real-world datasets. Sparsity is considered in designing the benchmark workloads, but null handling is not benchmarked nor discussed. More relevant is the work done in [45] which evaluates suitability of the column-oriented formats Apache Arrow, Parquet and ORC for direct use in an analytical DBMS. Arrow is especially close to our use case as it is tailored for in-memory data encoding and processing. Null handling is not specifically considered, however.

III. Null Handling Benchmark

A. Null Encodings

Based on the survey in Section II-A and research on related work in Section II-B, we narrowed down the candidates for benchmarking to the following data structures for encoding null status of array values.

BoolArray: A boolean array of same size as the associated array where a true value means the corresponding data value is null. This is in use by many of the tools covered in Section II-A, including rasdaman enterprise, TileDB, NumPy, PyTorch and TensorFlow, and Vectorwise and ClickHouse. The key to its widespread use is that it is the most straightforward implementation-wise. It may also be advantageous to more streamlined, vectorized operation processing. The main disadvantage is that it takes up a significant amount of space (null status of one value per byte), as much as the data itself in case of 8-bit integer arrays.

BitVector: A bit vector improves on BoolArray by packing 8 indices per byte to reduce space requirements by 8x. Considering that much of the software dealing with arrays is implemented in C++, we specifically selected the C++ `std::vector<bool>` in this case, which is readily available in the C++ standard library.

RoaringU: Roaring bitmaps is a well-optimized library for bitmaps for 32-bit and 64-bit integers [8]. This case refers to CRoaring's `bitset_t`, which is a conventional uncompressed bit vector like BitVector. The implementation is different enough from BitVector to warrant its inclusion. Most significantly, the bitset is stored in contiguous memory, unlike BitVector which makes no such guarantees.

RoaringC: Bitmap compression offers further space reduction, with generally little performance penalty. Some of the software covered in Section II-B uses standard RLE for compressing bitmaps, which is already a good solution especially in very sparse or very dense data. However, Roaring compressed bitmaps are overall considered the best specification for bitmap compression [46], so we decided to focus on Roaring. Specifically, we benchmark CRoaring's C++ `Roaring` data structure which is specifically tailored for 32-bit integers. As virtually all software for array management and processing splits arrays internally into smaller array tiles, a 32-bit index space is sufficient; support for 64-bit integers is nevertheless available as a separate `Roaring64Map` type if needed.

We cover only solutions that explicitly encode the positions in the array which are null. Alternatives which in every operation verify null status by checking membership in a null set, such as in rasdaman community (flexible), or MATLAB and R (fixed null sets), are left out for several reasons:

1) One or more values from the valid type range of values must be allocated as null values, which is not always possible and may require expensive type promotion.
2) It is possible to incorrectly convert non-null values into null values, e.g. if the null set is 0 then $1 - 1$ will incorrectly result in a null value in the result.
3) It is difficult to correctly compare against the other encodings as they do not require any reference to the array values to calculate the result null masks.

B. Null Operations

Several operation categories can be defined based on how null masks are used and propagated into the result.

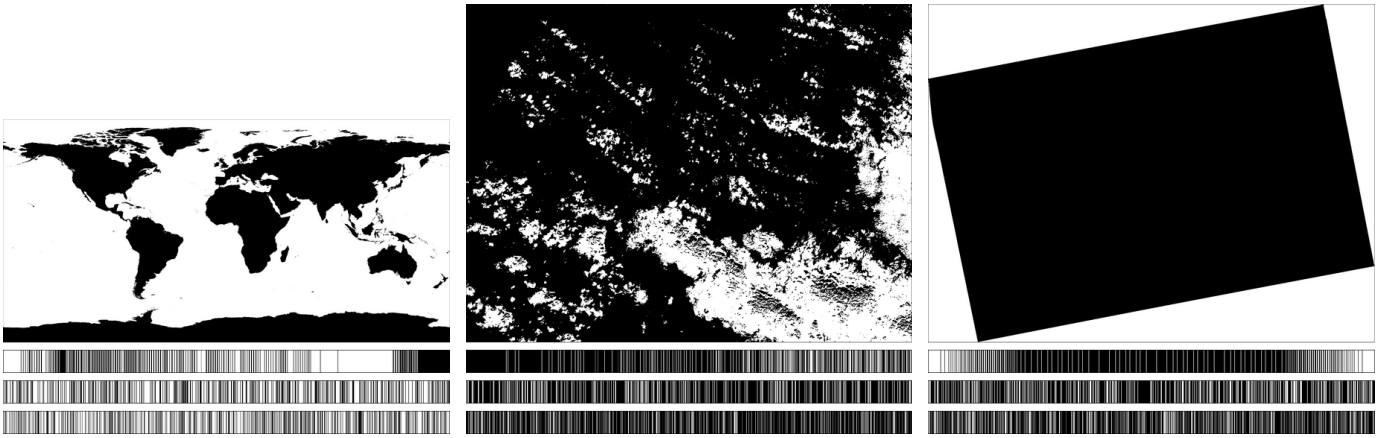
Import: To handle *external* data, a DBMS or other tools have to import it from the original format into its internal data structures. As it is not practically feasible to consider all possible data formats, we use a fixed representation of boolean array for external null values (i.e. same as BoolArray) and measure the time to convert into each of the null encodings. The first three cases are implemented by iterating through the external null mask and copying the set bits into the internal null mask. For RoaringC, however, it was more efficient to first create an intermediate array of null value position indexes, which is then bulk-loaded into the Roaring bitmap.

Export: This is the opposite operation: data is exported from the internal datastructures into a data format understandable by other software. We measure converting each null encoding representation to a boolean array. This is especially efficient for BoolArray, as it is a fast memory copy. BitVector and RoaringU require extracting the null statuses index-by-index, while RoaringC supports iteration over the null value positions.

Serialize: This operation is needed to store null values on disk or send over the network in the context of distributed applications. RoaringU stores the data in contiguous storage, so it can be serialized efficiently. RoaringC provides specific API for this purpose, which we utilize in the benchmark. Size has large impact, so in the other two cases we implemented efficient packing of eight null statuses into a single byte. In all cases the result is written to a binary file on disk.

Deserialize: This is the inverse operation of Serialize, allowing to read back data from disk or received over the network. The benchmark tests reading the serialized file from disk and reconstructing the null mask.

Unary: Unary induced operations are overloaded on array arguments, in which case they apply to each array cell in turn. For example, $\sqrt{A}$ is evaluated as $\sqrt{A[0]}$, $\sqrt{A[1]}$, and so on. Typically the most efficient implementation of such operations is to disregard the null statuses to avoid costly branches, and simply apply the operation to all values equally. The null mask is usually copied in array databases, where in complex query trees data is shared across multiple operators, so we measure copying a null mask in the benchmark.



(a) ERA5-Land climate data with valid data only over land mass (65% sparsity).

(b) Clouds on a Sentinel-2 satellite image are null values (22% sparsity).

(c) Sentinel-1 satellite scene has nulls as a result of orthorectification (28% sparsity).

Fig. 1: Sample null distributions: At the bottom of each null mask a distribution is shown of the nulls created by (top) 1D row-major linearization, (middle) RANDOMRUNS distribution, and (bottom) RANDOMNULLS distribution.

Binary: Binary induced operations are similar, but they are applied on two array operands (or an array/scalar combination). For example, $A+B$ sums all corresponding elements of A and B, while $A/2$ divides each value of A by 2. Besides applying on the operands data, in this case it is necessary to also do a set union on their null masks. Any null cell value in one of the arguments leads to a null value in the result array. The Roaring datastructures provide API for fast calculation of union, while for BOOLARRAY and BITVECTOR we implemented it manually.

Aggregate: An important class of operations summarize the values of an array into a single scalar value, e.g. a sum, average, minimum/maximum, etc. Null statuses must be considered while computing the result, because generally the null values have to be ignored. Fast querying of the null status at an individual position is particularly important in this case.

Subset: Datacubes are Big Data in volume, and support for selecting rectangular subsets (or cutouts) out of the whole dataset is essential. Subsetting generally requires multiple copy operations of some length, depending on how the array is linearized in memory. In the benchmark we approximate it to 1-D by copying the null statuses at positions in the interval $[size/4, size \cdot 3/4]$ into the result. This is representative of what is performed many times over for higher-dimensional data. For BOOLARRAY it can be implemented efficiently with copying memory directly, while the other cases require less efficient probing for the null status of each value.

Extend: Growing an array requires filling the extended empty parts with null values. The benchmark tests how efficient it is to initialize a nullmask to all nulls within the specified spatial domain.

Scale: Data fusion often requires matching arrays with different grid resolutions, and here a scale, also known as resample, is an essential operation needed to bring two or more inputs into a common resolution. We measure downscaling by a factor of 1.5x with nearest-neighbor interpolation. The implementation iterates through each coordinate c in the scaled spatial domain of the result, and copies the null status from coordinate $\lceil c \cdot (1/factor) \rceil$ in the input array.

Clip: Selecting a subset defined by a more complex geometry, such as a polygon, is a common array operation. Areas outside of the clipped region are set to null, while those inside are copied to the result. In the benchmark we approximate this behavior by doing a similar operation to Subset, except that positions in the intervals $[0, size \cdot 1/4)$ and $(3/4, size)$ are additionally set to null.

Mosaic: This is usually an internal operation not exposed to users. Array processing is applied on tiles (also known as chunks) of the array, which facilitates concurrent processing and lower memory overhead. At some points in the processing pipeline (known as blocking operations) the tiles have to be merged into a larger tile / array. We implement this by concatenating two input null masks.

C. Data

All benchmarked null operations are implemented specifically for 1-dimensional array data. Insights from the results are fundamentally valid to higher-dimensional data as well, and implementations remain straightforward.

Selecting representative real-life data is difficult and would inevitably lead to some biases, therefore we generate syntetic data in the benchmark. Randomly generated null distribution, depending on the sparsity, does not result in very realistic data aside of nulls due to spurious sensor error for example. Such random errors are very rare, so the benchmark still supports nulls randomly generated with the standard C++ `std::discrete_distribution` method, which could be used for very low sparsity tests.

In cases of higher sparsity, usually real data has connected areas of null values, e.g. areas over the Atlantic Ocean on a dataset with temperatures over land (see Figure 1). Here the generated distribution of nulls is relevant especially for

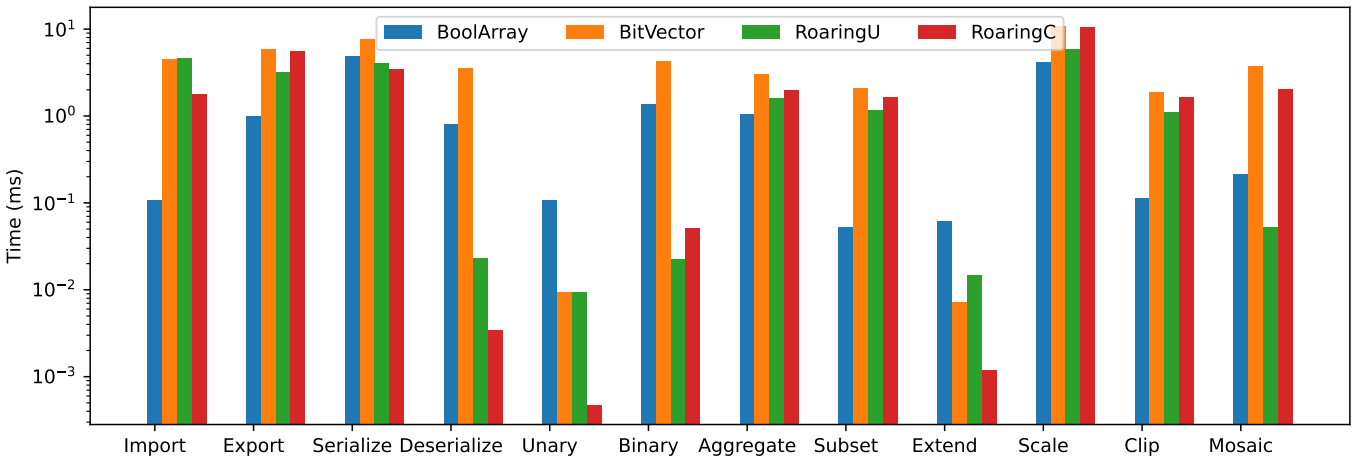


Fig. 2: Comparison of the null encoding schemes across each operation at a fixed mask size of 10^6 and sparsity of 25%. The y axis is logarithmic and shows the runtime in milliseconds.

benchmarks on compressed encodings, as purely random data is hardly compressible. For this reason we devised a flexible method for generating distributions that primarily consist of connected null areas (Algorithm 1).

Algorithm 1 RANDOMRUNS

$mask[0 : SIZE - 1] \leftarrow 0$ ▷ Initialize mask
$H \leftarrow \sqrt{SIZE}$ ▷ Image height
$TN \leftarrow SIZE \cdot SPARSITY$ ▷ Total number of nulls
$TV \leftarrow SIZE - TN$ ▷ Total valid values
$TR \leftarrow \sum_{i=0}^{H} rand(0, 5)$ ▷ Total number of runs
$AN \leftarrow TN/TR$ ▷ Average number of nulls per run
$AV \leftarrow TV/TR$ ▷ Average number of valid values
$i \leftarrow 0$
while $i \leq SIZE$ **do**
  $rs \leftarrow rand(1, AN \cdot 2)$ ▷ Null run size
  $rs \leftarrow min(rs, SIZE - i)$ ▷ Clip to bounds
  if $rs > 0$ **then**
    $mask[ri : ri + rs - 1] \leftarrow 1$ ▷ Set nulls in mask
    $i+ = rs$
  end if
  $vs \leftarrow rand(1, AV \cdot 2)$ ▷ Non-null run size
  $vs \leftarrow min(vs, SIZE - i)$ ▷ Clip to bounds
  $i+ = rs$
end while

We consider the real data to be 2-dimensional in square format with height H. The algorithm generates between 0 and 5 runs of null values per each row; the 5 is configurable in the benchmark. Each run of null values has a size between 1 and $AN \cdot 2$, where AN is the average number of nulls per run as determined by a data sparsity parameter, and is followed by a run of valid values with size between 1 and $AV \cdot 2$, where AV is the average number of valid values per run. This technique results in distributions that are much closer to those of real-life data examples (Figure 1) with respect to compressibility. This is clearly observable through the file sizes of the DEFLATE-compressed PNG files (Table I).

Example	Original	RANDOMRUNS	RANDOMNULLS
ERA5 (Fig. 1a)	402	484	992
Sentinel-2 (Fig. 1b)	499	440	870
Sentinel-1 (Fig. 1c)	295	448	959

TABLE I: PNG file sizes (in bytes) of null distributions.

IV. RESULTS & DISCUSSION

The benchmark is implemented in C++ as a collection of micro-benchmarks with the Google Benchmark framework, and compiled with GCC 11.4.0 with compiler flags `-O3 -mavx2 -mfma`. It was executed on a machine running Ubuntu 22.04 OS with 2x Intel Xeon E5-2609v3 CPUs (2x6 cores @ 1.90GHz, 16MB L3 cache, 256kB L2, 32kB L1), 64GB DDR4 2133MHz RAM, and an SSD hard disk with read spead of 520 MB/sec. The CPU scaling governor was set to `performance`. Benchmark processes were isolated from other processes on the system with `cset shield`.

A. Defaults Benchmark

Figure 2 shows runtime per null mask operation at default fixed parameters. Sparsity is fixed at 25% as we want to give more weight to dense array applications in this benchmark; the full range of sparsities is benchmarked in Section IV-B. Array size is fixed to 10^6 values, as in our experience array tiles are most commonly 4-5 MB in practice, which, depending on the cell size, hold roughly 1 million cells.

BITVECTOR is pretty uniformly worse than ROARINGU; an exception is Extend which initializes a mask with all null values. ROARINGU exhibits a well balanced performance, and we believe that it may have potential to support better optimized implementations for the Subset and Clip, as the storage is contiguous like BOOLARRAY. ROARINGC appears particularly suitable for Serialize / Deserialize cases as well as

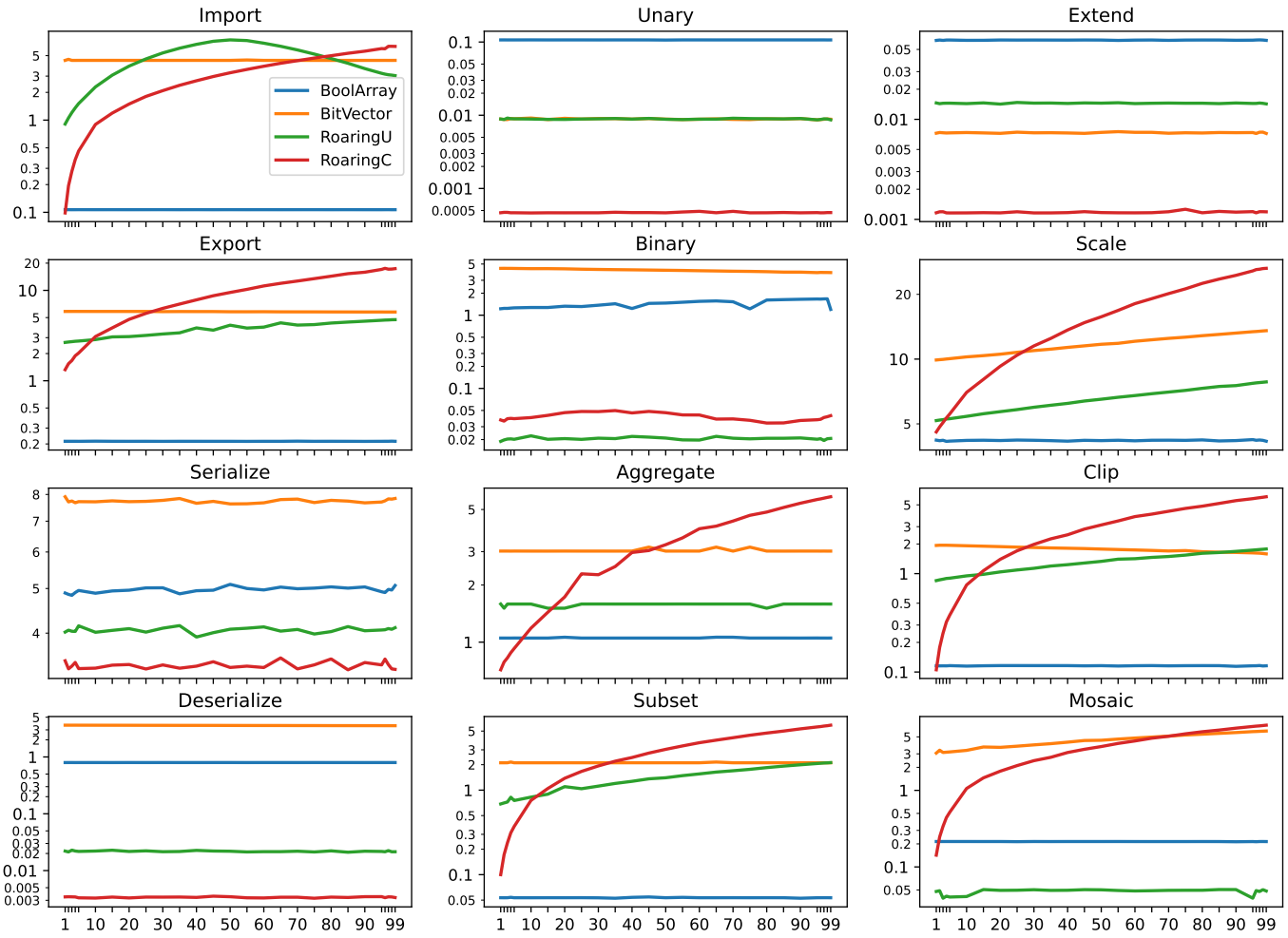


Fig. 3: Comparison of the null encoding schemes across each operation at 1-4% sparsity, then 5 to 95 in steps of 5, and finally 96-99%. Mask size is fixed to 10^6; the y axis is logarithmic and shows serial operation runtime in milliseconds.

when we just need to copy the mask like in a Unary operation. This is not surprising as it has compact compressed encoding that makes it an order of magnitude smaller in size than bit-packed BITVECTOR and ROARINGU (Section IV-C).

B. Specific Benchmarks

Varying Sparsity: The null sparsity makes a difference in how some datastructures behave with respect to performance. The benchmark results are shown on Figure 3. BOOLARRAY and BITVECTOR are not affected much by the data sparsity. ROARINGU is also mostly unaffected, with slight performance degradation at higher sparsities in the Subset, Scale, and Clip operations. It has an interesting performance curve for Import that initializes the CRoaring bitset from a boolean array, likely due to CPU branch prediction specifics. ROARINGC exhibits two types of behavior. When its performance is independent of the sparsity it tends to be the fastest performing on a given operation; these operations either do not involve probing for null status of individual elements, or are well optimized such as a union in Binary operations. In other cases, performance drops fast as the sparsity level increases, often by an order of magnitude between 1 and 10% sparsity.

Varying Size: Figure 4 shows how different null mask sizes affect the runtime of operations. ROARINGC scales well in the same cases where it had an edge in the previous benchmark. Due to the effective compression its real size grows at a slower rate (around 3.5x by empirical measurement) than the number of values (10x). The other datastructures behave more linearly with respect to the number of values. Between 10^7 and 10^8 we observe a drop in performance for BOOLARRAY, and similarly between 10^8 and 10^9 for BITVECTOR and ROARINGU. These correspond to the same physical size between 10 and 100 MB, as bitsets use almost 10x less space than BOOLARRAY. This performance drop is likely related to the size of CPU caches. For Serialize we observe that 10^6 tends to be the sweet spot, which confirms why it is a popular tile size for storage in array databases; 10^5 might be a little better, but it results in more tiles with negative effects on index performance.

Concurrency: Most array operations tend to be embarassingly parallelizable by processing multiple array tiles con-

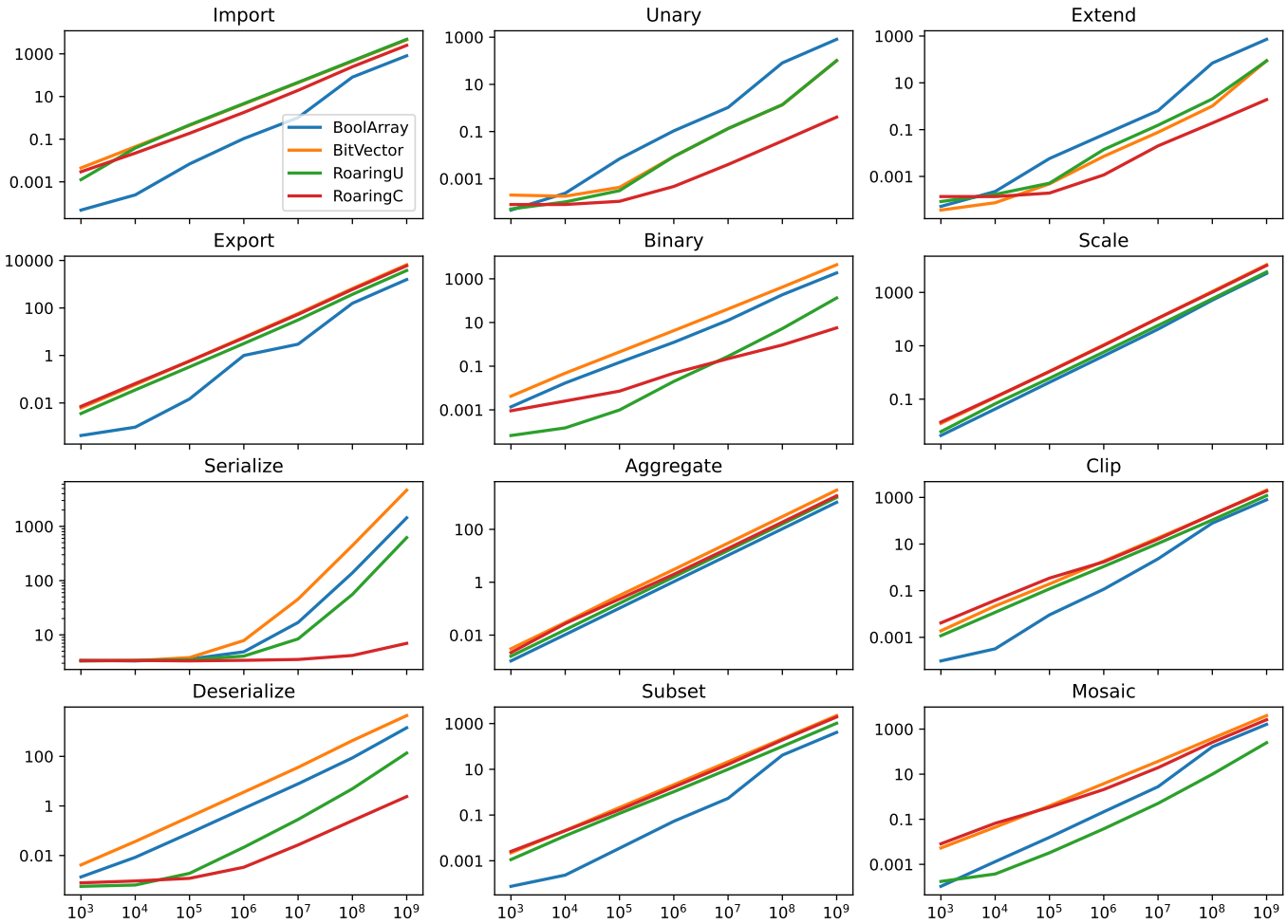


Fig. 4: Comparison of the null encoding schemes as the null mask size increases from 10^3 to 10^9 elements at a fixed sparsity of 25%. Both the x and y axes are logarithmic; the y axis shows serial operation runtime in milliseconds.

currently, so it is important to measure performance with a varying number of threads. The results from our benchmark of increasing concurrencty from 1 to 10 threads show that all methods scale similarly well as more cores are used.

C. Memory Usage

Reducing memory pressure during processing can have large effects on the overal runtime, as it allows more of the datasets to fit in fast main memory and CPU caches. For a null mask of N, the memory usage of BOOLARRAY is fixed to N bytes, and of BITVECTOR and ROARINGU to $N/8$ bytes. Determining the memory usage of ROARINGC is more complex as it is difficult to express with a simple formula. We implemented a benchmark that creates a Roaring bitmap (RANDOMNULLS or RANDOMRUNS) of size 10^6 at a configurable sparsity, and measured the memory usage empirically with the Valgrind Massif tool. The results showed that ROARINGC has a relatively constant memory usage of around 9-10 kB on RANDOMRUNS regardless of sparsity, which is an order of magnitude lower than uncompressed BITVECTOR/ ROARINGU. On RANDOMNULLS it was no better than uncompressed bitset, except at extreme 1-5% and 98-99% sparsities.

V. CONCLUSION

Proper null value handling in array services and tools is highly important in unsupervised Big Data Analytics, from both a correctness and a performance perspective. It will become even more critical once pretrained neural networks are going to operate autonomously on datacubes. This is being investigated, for example, in the AI-Cube [47] and FAIRiCUBE [48] projects. A commercial example is the IBM / NASA collaboration to create an open foundational AI model for geospatial datacubes [49].

Our contribution is the first rigorous benchmark of the most promissing techniques available for representing null values in datacubes. The authors have not been involved in the development of these methods. However, they have deep experience in array database architecting, which has helped crafting this benchmark in a practically relevant way. The authors hope that this research contributes to advancing the state of the art in the field of datacube research and development. To that end,

the complete benchmark code is publicly available, thereby allowing to reproduce the results presented in this paper, or extend to further techniques and operations [9].

ACKNOWLEDGMENT

This work was in part supported by NATO Science for Peace and Security under project Cube4EnvSec and EU project FAIRiCUBE, which is gratefully acknowledged.

REFERENCES

[1] Rasdaman, "The rasdaman Raster Array Database," http://rasdaman.org, accessed: 2018-feb-28.
[2] TileDB, Inc., "TileDB," https://tiledb.com, 2023.
[3] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science and Engg.*, vol. 13, no. 2, pp. 22–30, Mar. 2011. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2011.37
[4] PyTorch Contributors, *PyTorch Documentation, v2.0*, 2023. [Online]. Available: https://pytorch.org/docs/2.0/
[5] TensorFlow Developers, *TensorFlow API Documentation, v2.12.0*, 2023, https://tensorflow.org/versions/r2.12/api_docs.
[6] M. Zukowski, M. Van de Wiel, and P. Boncz, "Vectorwise: A vectorized analytical DBMS," in *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 2012, pp. 1349–1350.
[7] ClickHouse, Inc., "ClickHouse," https://github.com/ClickHouse/ClickHouse, 2023.
[8] D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. Ssi-Yan-Kai, "Roaring bitmaps: Implementation of an optimized software library," *Software: Practice and Experience*, vol. 48, no. 4, pp. 867–895, 2018.
[9] D. Misev, M. Rodionychev, and P. Baumann, "Benchmark Code for "Performance of Null Handling in Array Databases"," Sep. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8313077
[10] P. Baumann, "Language Support for Raster Image Manipulation in Databases," in *Proc. Int. Workshop on Graphics Modeling, Visualization in Science & Technology*, Darmstadt, Germany, 1992.
[11] ——, "Management of Multidimensional Discrete Data," *VLDB Journal*, vol. 3, no. 4, pp. 401–444, 1994. [Online]. Available: http://dl.acm.org/citation.cfm?id=615204.615207
[12] "ISO/IEC 9075-15:2019: Information technology database languages — SQL — Part 15: Multi-dimensional arrays (SQL/MDA)." *ISO*, 2019.
[13] P. Baumann, "OGC Web Coverage Processing Service (WCPS) Language Interface Standard," *OGC 08–068r2*, 2009.
[14] Rasdaman Community, *Rasdaman v10.2 Documentation*, 2023, https://doc.rasdaman.org/10.2/.
[15] rasdaman GmbH, *Rasdaman v10.2 Documentation*, 2023.
[16] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The Architecture of SciDB," in *Proc. 23rd International Conference on Scientific and Statistical Database Management*, ser. SSDBM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032397.2032399
[17] Paradigm4, Inc., "SciDB Release Repository," https://downloads.paradigm4.com/community/19.11/, 2023.
[18] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, "The TileDB Array Data Storage Manager," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 349–360, 2016.
[19] J. Beckmann, A. Halverson, R. Krishnamurthy, and J. Naughton, "Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format," in *22nd International Conference on Data Engineering (ICDE'06)*. Atlanta, GA, USA: IEEE, 2006, pp. 58–58.
[20] PostgreSQL Global Development Group, *PostgreSQL 15 Documentation – Table Row Layout*, 2023, https://postgresql.org/docs/15/.
[21] Oracle, *MySQL 8.1 Reference Manual – InnoDB Row Formats*, 2023, https://dev.mysql.com/doc/refman/8.1/en/.
[22] SQLite Consortium, *SQLite Documentation – Database File Format*, 2023, https://www.sqlite.org/docs.html.
[23] Apache Druid Community, *Apache Druid 27.0.0 Documentation – Segments*, 2023, https://druid.apache.org/docs/27.0.0/.
[24] M. Raasveldt and H. Mühleisen, "DuckDB: An Embeddable Analytical Database," in *Proceedings of the 2019 International Conference on Management of Data*. Amsterdam Netherlands: ACM, Jun. 2019, pp. 1981–1984.
[25] W. McKinney, *Apache Arrow and the "10 Things I Hate About pandas"*, 2017, https://wesmckinney.com/blog/apache-arrow-pandas-internals/.
[26] Apache Arrow Community, *Arrow Columnar Format version 1.3 – Validity bitmaps*, 2023, https://arrow.apache.org/docs/format/Columnar.html.
[27] Apache Parquet Community, *Apache Parquet Documentation – Nulls*, 2023, https://parquet.apache.org/docs/.
[28] Apache ORC Developers, *Apache ORC Specification v1 – Stripes*, 2023, https://orc.apache.org/specification/ORCv1/.
[29] R. Rew, G. Davis, S. Emmerson, H. Davies, E. Hartnett, and D. Heimbigner, "The NetCDF Users Guide – Data Model, Programming Interfaces, and Format for Self-Describing, Portable Data - NetCDF Version 4.1," March 2010.
[30] E. Rouault, F. Warmerdam, K. Schwehr, A. Kiselev, H. Butler, M. Łoskot, T. Szekeres, E. Tourigny, M. Landa, I. Miara, B. Elliston, K. Chaitanya, L. Plesea, D. Morissette, A. Jolma, N. Dawson, D. Baston, C. de Stigter, and H. Miura, "GDAL," May 2023. [Online]. Available: https://github.com/OSGeo/gdal/
[31] Environmental Systems Research Institute, Inc, *ArcGIS Pro help – NoData in raster datasets*, 2023, https://pro.arcgis.com/en/pro-app/latest/help.
[32] NumPy Developers, *NumPy reference, release 1.25*, 2023. [Online]. Available: https://numpy.org/doc/1.25/reference/
[33] The MathWorks Inc., *MATLAB Documentation – Missing Data in MATLAB*, https://mathworks.com/help/, 2023.
[34] R Developers, "R source code, v4.3.1," https://svn.r-project.org/R/tags/R-4-3-1/, 2023.
[35] P. Baumann, D. Misev, V. Merticariu, and B. P. Huu, "Array Databases: Concepts, Standards, Implementations," *Big Data*, 2021.
[36] G. Merticariu, D. Misev, and P. Baumann, "Towards a general array database benchmark: Measuring storage access," in *Big Data Benchmarking*, T. Rabl, R. Nambiar, C. Baru, M. Bhandarkar, M. Poess, and S. Pyne, Eds. Springer International Publishing, 2016, pp. 40–67.
[37] R. Wieruch, "Mongodb: Avoid large arrays - benchmark," http://www.robinwieruch.de/avoid-large-arrays-in-mongodb-benchmark/, 2014, accessed online on November 16, 2023.
[38] R. Taft, M. Vartak, N. R. Satish, N. Sundaram, S. Madden, and M. Stonebraker, "GenBase: A Complex Analytics Genomics Benchmark," in *Proc. ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 177–188. [Online]. Available: http://doi.acm.org/10.1145/2588555.2595633
[39] Y. Cheng and F. Rusu, "Formal Representation of the SS-DB Benchmark and Experimental Evaluation in EXTASCID," *Distributed and Parallel Databases*, pp. 1–41, 2013.
[40] P. Baumann and H. Stamerjohanns, "Benchmarking Large Arrays in Databases," in *Proc. Workshop on Big Data Benchmarking*, December 2012, pp. 94–102.
[41] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, S. Madden, M. Stonebraker, S. B. Zdonik, and P. G. Brown, "Ss-db: a standard science dbms benchmark," in *Proc. XLDB Workshop*, 2010.
[42] G. Merticariu, D. Misev, and P. Baumann, "ADBMS Storage Benchmark Framework," https://github.com/adbms-benchmark/storage, 2015, accessed online on November 16, 2023.
[43] D. J. Abadi, "Column stores for wide and sparse data," in *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 2007, pp. 292–297. [Online]. Available: http://cidrdb.org/cidr2007/papers/cidr07p33.pdf
[44] X. Zeng, Y. Hui, J. Shen, A. Pavlo, W. McKinney, and H. Zhang, "An Empirical Evaluation of Columnar Storage Formats," Apr. 2023.
[45] C. Liu, A. Pavlenko, M. Interlandi, and B. Haynes, "A Deep Dive into Common Open Formats for Analytical DBMSs," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, pp. 3044–3056, Jul. 2023.
[46] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "An Experimental Study of Bitmap Compression vs. Inverted List Compression," in *Proceedings of the 2017 ACM International Conference on Management of Data*. Chicago Illinois USA: ACM, May 2017, pp. 993–1008.
[47] AI-Cube, 2023, https://ai-cu.be.
[48] FAIRiCUBE, "F.a.i.r. datacubes," 2023, https://fairicube.nilu.no/.
[49] IBM, *IBM and NASA open source the largest geospatial AI foundation model on Hugging Face*, 2023. [Online]. Available: https://research.ibm.com/blog/nasa-hugging-face-ibm