The OTree: Multidimensional Indexing with efficient data Sampling for HPC

Cesare Cugnasco
Barcelona Supercomputing Center
cesare.cugnasco@bsc.es

Hadrien Calmet
Barcelona Supercomputing Center
hadrien.calmet@bsc.es

Pol Santamaria
Barcelona Supercomputing Center
pol.santamaria@bsc.es

Raül Sirvent
Barcelona Supercomputing Center
raul.sirvent@bsc.es

Ane Beatriz Eguzkitza
Barcelona Supercomputing Center
beatriz.eguzkitza@bsc.es

Guillaume Houzeaux
Barcelona Supercomputing Center
guillaume.houzeaux@bsc.es

Yolanda Becerra
Barcelona Supercomputing Center
yolanda.becerra@bsc.es

Jordi Torres
Barcelona Supercomputing Center
jordi.torres@bsc.es

Jesus Labarta
Barcelona Supercomputing Center
jesus.labarta@bsc.es

***Abstract*—Spatial big data is considered an essential trend in future scientific and business applications. Indeed, research instruments, medical devices, and social networks generate hundreds of petabytes of spatial data per year. However, many authors have pointed out that the lack of specialized frameworks for multidimensional Big Data is limiting possible applications and precluding many scientific breakthroughs. Paramount in achieving High-Performance Data Analytics is to optimize and reduce the I/O operations required to analyze large data sets. To do so, we need to organize and index the data according to its multidimensional attributes. At the same time, to enable fast and interactive exploratory analysis, it is vital to generate approximate representations of large datasets efficiently. In this paper, we propose the Outlook Tree (or OTree), a novel Multidimensional Indexing with efficient data Sampling (MIS) algorithm. The OTree enables exploratory analysis of large multidimensional datasets with arbitrary precision, a vital missing feature in current distributed data management solutions. Our algorithm reduces the indexing overhead and achieves high performance even for write-intensive HPC applications. Indeed, we use the OTree to store the scientific results of a study on the efficiency of drug inhalers. Then we compare the OTree implementation on Apache Cassandra, named Qbeast, with PostgreSQL and plain storage. Lastly, we demonstrate that our proposal delivers better performance and scalability.**

***Index Terms*—multidimensional indexing, distributed data store, High-performance computing**

I. Introduction

Many authors [1] [2] [3] [4] pointed out how multidimensional and spatial big data will be an essential part of future scientific and business applications. In particular, Eldawy et al. [1] described how we are entering the "*Era of Big Spatial Data*", with a single space telescope generating up to 150 GB of spatial data per week [5], medical devices producing spatial images at a rate of 50 PB per year and social networks managing billions of geo-tagged events per day. However, as described in Section V, the lack of specialized frameworks dealing with dimensional and spatial data limits applications, and probably, precludes many scientific breakthroughs, as most of the existing algorithms are designed for unidimensional problems and are suboptimal in high dimensional spaces.

Scientific simulations, IOT sensors, and various business applications generate complex data sets where multiple correlated characteristics describe each item. For instance, a particle might have a space position (x,y,z) at a given time (t). If we want to find all the elements within a particular area at a given time, we either have to scan the whole dataset, or we organize and group the items according to their space coordinates and time. The second approach is called Multidimensional Indexing (MI). While uni-dimensional indexing on large data sets is widely adopted in many sectors, MI differs because multidimensional points lack an intrinsic natural order, and therefore all indexing techniques which rely on ordering data cannot be directly applied. An alternative approach is to reduce the dimensions' granularity and combine them in a unique, distinct value. Many databases use this approach, but it only works well when the data distribution is mostly uniform and does not change with time. For instance, if we split a city map into quadrants of one km squared size, and we create a file for each quadrant containing the names of the restaurants and shops in that area, the data will be unbalanced, and some files will be larger, but still, none of them will be unmanageable. However, if we use the same approach to track the position of people, we will see that the files whose areas match with stadiums, concert halls, and shopping malls will be much larger than others, with a distribution that changes over time, or day and night. To overcome these limitations, Multidimensional Indexes take care of adapting the way data is partitioned following its statistical distribution, even when it changes over time.

On the other side, approximate analytics has often been indicated [6][7] as a smart and flexible way to interactively explore large data sets in a short period, as it allows to test and to try different hypothesizes rapidly. Still, if we want to reduce the number of I/O operations, we need *efficient data* *sampling*, which means that the index layout allows fetching data in incremental uniform samples until we gather the full dataset. In such a way, we can stop the query when we retrieve a large enough sample, or we achieve the desired accuracy. In contrast, traditional approaches require scanning the whole index in order to generate a random sample.

To our knowledge, none of the existing solutions combines scalable Multidimensional Indexing and efficient data Sampling (MIS). We consider such a feature fundamental when dealing with large data sets, as it enables more flexible data pipelines, and new types of interactive analysis and data exploration. For instance, in scientific computation, these two features combined enable interactive exploration and visualization with any arbitrary level of precision the outcomes of physics simulation, even when the simulation is still running,

This paper presents our work toward a novel peer-to-peer, distributed MIS indexing schema that can be applied to both HPC and data analytics workloads. First, Section II introduces the topic of multidimensional indexes and describes our previous contribution, the D8tree. We present the key concepts, the criticalities, and analyze its limits. As a result of the analysis, Section III proposes the OutlookTree and its implementation in Qbeast, our distributed indexing system built on top of Apache Cassandra. Section IV discusses how we tested the performance of our solution by using Qbeast to store and index in real-time the simulation results of Alya[8], an inhouse HPC-based multi-physics simulation code designed to simulate highly complex problems and efficiently run on high-end supercomputers. In this section, we also compare Qbeast against PostgreSQL and plain file storage on GPFS in write throughput and performance of exploratory queries. Finally, Section V, presents a summary of the existing related works, while in Section VI we present our conclusions.

II. Background

There are three main categories of multidimensional indexing algorithms: Space partitioning indexes like the QuadTree and the KD-Tree, Binary Tree evolutions such as the R-tree and its variant, and B+-trees that use space-filling curves to map the n-dimensional space to a scalar value. While the three approaches have their strengths and limitations, they all rely on a hierarchical tree structure, which is not trivial to distribute and to maintain across multiple machines without significant drawbacks. There are three main approaches to build a distributed index. Firstly, randomly partitioning the data and building a separate index in each machine, but then we need to broadcast each query to all nodes, nullifying the scalability of the system. Secondly, assigning a zone partition to each machine, leading to vulnerability to data hot-spots, and re-balancing when the distribution of the data changes. Thirdly, building a global index and randomly assigning each block of the index to a server. A drawback is that all queries need to start from the root node of the index; thus, all queries will question the same single server, which becomes a bottleneck.

Furthermore, we need expensive operations like distributed transactions and locks to preserve the consistency of the data when building the index dynamically. The Quadtree is a good example: first, create a space partition - a square -, and store data inside. When the number of elements stored reaches a threshold, split the partition into smaller equally-sized parts, and move the data into them. During this phase, we lock the partition, create the smaller squares in remote nodes, move all the data into them, and finally release the locks. These operations, while negligible in a multi-thread machine, are too expensive in multi-servers deployments. Distributed locks are not only an obstacle for system availability; they also increase response latency and diminish throughput.

In previous contributions [9] [10], we presented the integration of Alya [8] and the D8tree, demonstrating the advantages of interactive real-time exploration of large and long-running simulations. The D8tree employs de-normalization to avoid distributed transactions and to enable a uniform workload distribution between the cluster nodes. The idea is to build the index on a perfect 8-ary tree[1] with a configurable maximum height. Once they reach their maximum capacity, the nodes only keep a sample of the data in the node's domain. The sample is built using a random hash generated by the item identifier as the priority, so that when a node reaches its maximum size, we drop the elements with the lower priority. In this way, the children nodes contain a superset of the sample contained in the father. In other words, if α is a sample of 1% of the elements in a specific area, and β is a 2% sample in the same space, all items found in α will be present in β as well. This design ensures that the data is retrieved in incremental uniform samples that are used to compose a statistically valid preview of the final results and optimize the query execution.

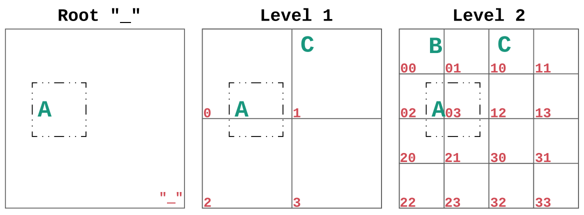

Fig. 1. A D8tree with depth 3 and partition max size = 1.

Figure 1 shows an example with three 2-dimensional points - A, B, C - with decreasing priority stored into a 3-levels D8tree with partitions that can contain one element at most. Every new item goes into the smallest corresponding area in the last level (e.g., "02" for A), and then it propagates to the "father" ("0" for A). If the father has reached its limit, we select the elements with a higher priority ($A > B > C$). We can see that larger partitions have smaller sampling fractions. For example, the root node "_" contains only one of the three that fall into is the domain(A). Similarly, partition "0" contains half of them and "02" all the elements in its domain. In this paper, we will alternately use the terms space partition and n-cube (or only cube) to identify the n-dimensional subset

[1]A k-ary tree with all leaf nodes at the same depth. All internal nodes have degree k.

of the domain. More formally, **X** is the domain of each n-cube representing all the elements that can potentially be stored in an n-cube, while the co-domain **Y** is the set of elements that the partition actually contains. Finally, f is the sampling fraction of each cube. Using the same example of Figure 1, we can see how **X**, **Y** and f are correlated and how f monotonically increases when the partition gets smaller.

$$f = \frac{|X|}{|Y|} : \qquad f_{root} = \frac{|\{A\}|}{|\{A,B,C\}|} = \frac{1}{3};$$
$$f_0 = \frac{|\{A\}|}{|\{A,B\}|} = \frac{1}{2}; \qquad f_{02} = \frac{|\{A\}|}{|\{A\}|} = 1$$

The rigid structure of the D8tree allows choosing different paths to complete a query. Let's suppose we are interested in all the data in the range $0.3 < x < 0.6$ and $0.15 < y < 0.40$ (the dashed blue rectangle in Figure 1). We can start from the root "_" and then proceed further down after analyzing what we found. Or we can directly go to level 1, reading "0" and "2", or to level 2 by issuing 4 requests for "02","03","20","21".

Such a design allows the D8tree to choose the right trade-off between the level of parallelism, the overhead of multiple requests, and the latency of multiple iterations. However, such a level of freedom comes at the cost of a high-overhead when building the index, making the D8-tree not feasible for write-intensive transactional workloads, as it adds numerous transactions and I/O requests for each insertion. As a result, the use of the D8tree is limited to read-intensive applications and slowly growing dataset, while it is unfit for write-intensive scenarios like HPC applications. Indeed, as the D8tree generates a perfect 8-ary tree, it means that a 3D index with 10 levels might have up to $\sum_{n=0}^{n=10} 8^n = \frac{8^{10+1}-1}{8-1} \approx 1.2 * 10^9$ n-cubes, and an element can be replicated up to 10 times. On a uniform distribution, the replication space overhead is about $\frac{1}{8}$ in a 3D index, and more generally $\frac{1}{2^D}$ for indexes of D dimensions. However, the cost can be much higher in real applications, as it strongly depends on the data distribution. While the user can decide to limit the overhead at the expense of the query speed by limiting the maximum index height, we observed replication overheads ranging from 60% to one order of magnitude [10].

In the D8tree, the replication benefits the query phase, but not all the additional copies bring sufficient advantages. Therefore, to understand how to change the D8tree without penalizing queries, we must study where the replication gives the most benefits. Replicating involves copying from one partition to multiple smaller and randomly distributed ones. Hence, when querying the index, we must find the optimal compromise between the number of groups and the size of their payload. Fewer but larger groups lead to higher latency, and they tend to distribute unevenly across servers, causing sub-optimal work distribution and parallelization. On the other hand, many small requests have a higher computation overhead. To this end, we used an analytical model we developed in a previous contribution [11] to estimate the performance of a distributed key-value database when varying the number of servers and the size of the data partitions. The model analyzes which architectural components limit the scalability of distributed databases. Given the data type, the number of nodes, the model predicts the maximum number of times that replicating an index partition gives a benefit. The model shows how the span-out of the D8tree's nodes in multidimensional spaces with three or more dimensions is sufficient to reach the optimal level of parallelism in clusters with up to 10 thousand nodes. As the fan-out correlates the density (f) of the area, we should favor replicating the areas with a smaller sampling size. To do so, we need a more general definition of the D8tree that allows not only to change the max-height dynamically but also to tune it for the different subparts of the dataset.

III. The OTree

Following the previous assumptions, we improved the original D8tree design by replacing the global "index max-height" with the idea that every single cube has its own "max-height", what we call the "outlook" (O). Instead of a single **K**-ary unbalanced rooted tree, the OuTree is an unbalanced tree composed of many locally balanced **O**-ary rooted trees. If cube "A" has outlook K then all descendants at a distance $\leq K$ exists, and each of them contains a copy of all the elements of "A"'. This information can be used to build query plans, as the outlook determines which nodes are directly visitable.

Definition III.1. The **OTree** is a **K**-ary unbalanced rooted tree **T(D)** where **D** is the number of dimensions and where each node can have up to $\mathbf{K} = 2^D$ children. Each node has a domain X, a co-domain Y and an overflow set F. The domain X is the set of all elements contained in the **D**th partition of the parent's domain, which is the K^Hth disjoint partition of the **D**-dimensional space, with **H** indicating the distance between the root and the given node. Each node contains $Y \subseteq X$ elements. The co-domain Y is a random uniform sample of X, where f is the sampling fraction between the cardinality of the domain X and the co-domain Y. The *OTree* guarantees that any node is the root of a perfect **K**-ary tree with a local height equal to the node's outlook **O**. When a cube has $O > 0$, all elements in its co-domain Y are also stored in the union of the co-domains of the descendants in each of the **O** levels downwards. On the other hand, when a cube has $O = 0$, the cube's co-domain might not be replicated. Thus, when the sampling fraction f changes, the elements removed from the domain X go in the overflow set F until they are forwarded to the descendants when the cube's outlook increases.

In particular, an OTree is said **regular** if every outlook is $O \geq \lceil \log_K \frac{1}{f} \rceil$, and all cubes have $F = \emptyset$. Note that the D8tree is a sub-case of the OTree where $O = max\_height - H$ and $F \neq \emptyset \implies L = max\_height$.

A. Querying the OTree

Querying the OTree is similar to querying the D8tree with the additional outlook constraint. Like the D8tree, every time we visit a cube, we use the f of each cube to determine at which level to jump, but we also need to ensure the jump is shorter than the outlook, and to consider the items in F.

We achieve a better distribution by keeping in an in-memory trie the outlooks of the top nodes so that queries can bypass them. Instead of starting from the root, Queries can start closer to the minimum bounding box (MBC) of the query, which is the smallest index partition that can contain all the searched information. Depending on the status of the index, the MBC could be directly visitable,if and only if all ancestors have $O > 0$, or not. Queries start from the MBC if it is visitable, and it has $O = 0$, while we can directly visit its descendants if 0 is greater than zero. Where to start depends on the percentage of data required by the query. If $f = 1$, the best approach is to follow the outlook, while if $f < 1$, it is better to visit the first ancestors that contain a large enough sample. For example, in a 3D OTree, if a cube has $f = \frac{1}{100}$, and $0 = 3$, but we need only 5% of the data, we will go one level down, instead of 3.

In case the MBC is not directly visitable, we must find the first ancestor that is visitable, and start from there.

Definition III.1 describes the characteristics that the index must follow to ensure fast and interactive query analysis, but it does not define how to build the index: how to decide the outlook and the co-domain size of each cube.

B. *A change of outlook*

Tuning the outlook of each cube, the *OTree* uses less disk space and transactions than the *D8tree*, as it "cuts" part of the tree. However, as the outlook changes, so does the index structure, and mutable structures are hard to keep both consistent and performant in a distributed environment. To increase the outlook of a cube, we must forward all its data to its descending nodes. A straightforward implementation would require distributed locks and transactions, similar to the ones needed for the Quadtree, limiting the system scalability and availability. For this reason, we designed and patented an architecture that allows implementing the index and ensuring data consistency without distributed locks while optimizing its structure asynchronously.

The OTree can use various policies to update the outlooks, but in this paper, we will focus on a strategy that we found better fitting for scientific applications. Since the the analysis of simulations often focuses on a few specific regions or timestamps, we opted for a strategy that optimizes a part of the index right after it has been queried. This straightforward approach has two main advantages: it fits well with interactive analysis, as we optimize only the parts of the index that has interested the user; and it makes index optimizations cheaper as the data is already in primary memory. We call this process *ReadOptimization* (RO). Once an RO completes forwarding the data of a cube to its descendants, we can increase the outlook. Besides the component that manages ROs, a major element of our indexing schema is the *RangeEstimator*, which has three main duties. Firstly, it reduces the number of transactions by avoiding to send a copy of the data to n-cubes where it does not fit. Secondly, it ensures that the outlooks are respected. Lastly, it avoids the loss of data during ROs. To achieve its goals, the RE uses an in-memory data structure to estimate in which nodes to insert the new items. At a higher level, the *RangeEstimator* is a function that calculates from which (r_{from}) level and to which (r_{to}) level new inserts should propagate. The RE uses the unique identifier of each element to generate a random priority that is used to estimate where a new element can fit. It starts from the root comparing f and the priority of the element. If f if smaller, the update does not fit, and the process iteratively continues until it finds the first child that can contain it (the r_{from}). In the meantime, the *RangeEstimator* calculates up to which level it should propagate the insertion; the value the r_{to}.

To correctly calculate r_{to} we must respect the outlooks of all nodes and ensure that no update is lost during ROs. Figure 2 shows an example of a Lost Update that can occur in any tree-based indexing algorithm when we have a node that has reached its maximum size, and we have to break it into new sub-partitions. The problem is that we risk losing data when concurrently reading and updating the index without a lock. In the image, the "splitter" process reads items from "..212" and assigns them to either "..2121" or "..2122", but if a concurrent insertion goes into node "..212", it will be propagated to the children nodes, resulting in an inconsistency.

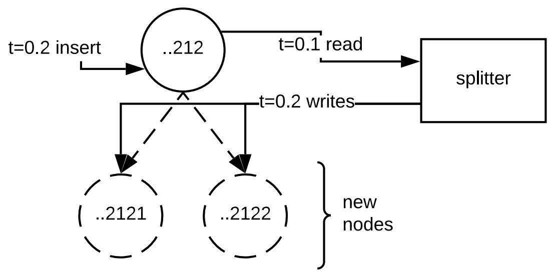

Fig. 2. Possible Lost Update during copy.

Using exclusive locks avoids inconsistency, but it puts on hold all operations on part of the index. To prevent such unsustainable performance cost, we designed a protocol that allows lock-free data copy and index evolution. When ingesting new data, our system builds a *OTree* with $O = 0$ that it is faster to write but less efficient to query. Later, after each query, as we have already retrieved data from the disk, we perform a background *ReadOptimization* that redistributes the data, speeding up future queries.

During ROs, the *RangeEstimator* preserves the consistency of the whole system by timely updating the r_{to} so that all new items propagate to the new children. Using the example in Figure 2, we ensure that while the splitter is copying the data from "..212" to its offspring, the *RE* sends all concurrent new insertions also to "..2121" and "..2122". In the meantime, queries must be unaffected by running ROs, and the outlook must not increase prematurely. To do so, we "announce" to all database nodes that we are going to replicate an n-cube. If all nodes acknowledge the announcement before we read the cube, we can optimize it. To ensure this mechanism, we identify four evolutional states of n-cube. All nodes start in the

leaf state. Then, when a node reaches its maximum capacity, and it contains only a fraction of its domain ($f < 1$), it evolves to **full**. When we decide a cube could be optimized, its state evolves to **announced**. Finally, once an RO eventually redistributes the data of a cube, it becomes **replicated**. The state is defined by three variables: the sampling fraction f; the time of the announcement acknowledgment (**announcement_time**); the outlook 0 tracking the number of *R0*s occurred.

C. Metadata consistency

Databases achieve high-performance aggregating disk accesses and keeping metadata about the data distribution and indexes in memory so that each query requires less than one I/O operation on average. In the case of the OTree, all nodes need to know the global status of the index, so we must ensure the metadata does not outgrow the memory of a single machine. At the same time, strict metadata consistency is expensive in distributed environments as it requires master-slave architecture, or guaranteed message delivery or consensus mechanisms like Paxos [12] to coordinate and propagate updates. To alleviate these problems, we studied how to reduce the requirements of precision and consistency of the metadata so we can use approximate data structures and unreliable communication to reduce the memory footprint and latency.

To preserve the index consistency, the Range Estimator must ensure that the estimated values (the ones with the $\hat{}$) obey the following inequalities with the real values:

$$\hat{r_{from}}(u) \leq r_{from}(u) \tag{1}$$

$$\hat{r_{to}}(u) \geq r_{to}(u) \tag{2}$$

Indeed, a smaller $\hat{r_{from}}$ causes data to propagate to a cube that cannot accommodate it, and it will be filtered out eventually. A larger $\hat{r_{to}}$ will insert the element in a node where it is not reachable by any query yet, a temporary waste of space that a background process will eventually fix. In both cases, no data is lost, and consistency is guaranteed.

We use two different functions to calculate $\hat{r_{from}}$ and $\hat{r_{to}}$. We define the estimator of r_{from}, as:

$$\hat{r_{from}}(u) = \min\{\mathbf{H_c} : \forall c | X_c \ni u \wedge \mathbf{p}(u) \leq \hat{f_{c_j}}\} \tag{3}$$

$$\hat{f_c} \geq f_c \tag{4}$$

where $\mathbf{H_c}$ is the height of the cube c and $\mathbf{p}(u)$ is the priority of item u. On the other hand, the estimator of r_{to} is:

$$\hat{r_{to}}(u) = \max\{\mathbf{H_c} : \forall c \ni u \wedge c \in \hat{R})\}, R \subseteq \hat{R} \tag{5}$$

$$R = \{c : O_c > 0 \vee c \in A\} \tag{6}$$

where R is the replication set containing all cubes where updates must propagate to respect the outlooks. A is the set of all announced cubes that we might optimize in the near future.

Thanks to this formulation we can implement $\hat{f}$ by keeping in memory only an arbitrary subset of the f values and defaulting to 1 in case of a miss. For instance, we can keep the smallest f values, as they ensure the greatest I/O savings.

Similarly, we can use for $\hat{R}$ any approximate membership structure like the Bloom Filters so that we can arbitrarily reduce the memory footprint at the cost of a higher indexing overhead. Differently, we might violate Inequality (5) if we miss a cube announcement in this case. Thus we must ensure all announcements are delivered.

At the same time, to speed up queries and to achieve uniform workloads, we need the outlooks so that we can jump directly to the required part of the index. In this case, the estimated $\hat{O}$ must be smaller than the real one so that a query might require more iterations, but it will never miss an update ($\hat{O} \leq O$). Similarly to f, also O has a monotonical (but increasing) tendency. Therefore, we can keep in memory an arbitrary large subset of the committed outlook to improve query performance. Furthermore, the metadata required by R and $\hat{R}$ can partially overlap (6), thus reducing the overall memory footprint. For instance, if cube “012” has $O_{012} = 5$, we can avoid saving all cubes “012*****” in the approximated membership structure used for the $\hat{R}$. In a 3D index, this could save $8^5 = 32768$ entries.

In case of repeated or lost messages, the monotonical tendency of both f and O makes trivial to rule out which is the most updated value, enabling the use of faster but less reliable communication protocols. On the other hand, we must reliably ensure that all nodes agree that a node is announced. At the moment, we use a naive implementation where we broadcast the information to all peers, and we require all peers to acknowledge. In case of loss messages or unresponsive peers, the operation is dropped, and it is retried in the future. In any case, there is no risk of inconsistency or system unavailability. As future work, we will consider a more efficient epoch-based approach, where announcements organize in timeslots so that peers can aggregate multiple updates in a single communication.

IV. OTree testing

This section contains the tests we ran to validate the performance of the OTree implementation of Qbeast. At first, we will introduce the scalability results generated by an open-source benchmarking tool. Secondly, we will discuss the performance of a real HPC application using Qbeast, focusing on its performance profile and the issues involved in integrating an MPI based code with a TCP based database. Lastly, we will propose a performance comparison of the time required to run the HPC application using as storage Qbeast, Cassandra, PostgreSQL, and a single file on GPFS.

We ran our tests at the Barcelona Supercomputing Center, in MareNostrum IV supercomputer. Each server contains two sockets with an Intel Xeon Platinum 8160 24C for a total of 48 cores and 96GB of ram for each server. Nodes are interconnected by a 100Gb Intel Omni-Path and a 10Gb Ethernet [13]. We use the local SATA 240GB Intel s3520 SSD scratch disk to store data. The disks are rated for sequential reads and write up to 320, and 300 MB/s respectively, while for random reads and writes up to 65000 and 16000 IOPS. We used fio [14] to benchmark the IOPS of GPFS when writing blocks of different sizes. The SSDs are 20 times faster for blocks of 4KB, 15 times for 64KB, while the GPFS is more than 5 times faster for large writes of 64MB.

Our first test aims to estimate the lower and upper bound performance of our system. We configured database clusters of increasing size, and we use numerous clients to perform random insertions. We used a stress tool shipped with Cassandra to benchmark the system, using twice as many machines for the stress tool than the database. We performed random insertions with a Gaussian distribution. We used the data model of Alya, which consists of a particle identifier as the partition key and the time as the clustering key. The rest of the values are the x, y, z positions, speed, acceleration, and other physical characteristics of the particles, for a total of 15 doubles and 3 integer numbers.

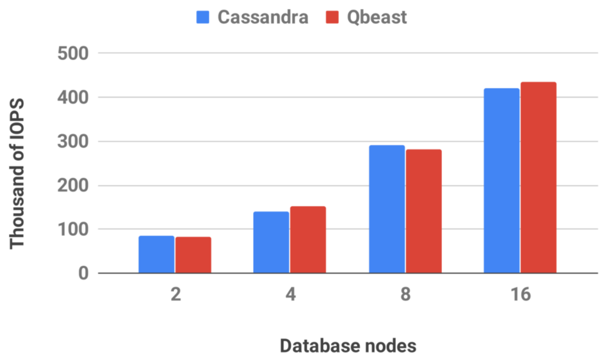

Fig. 3. Thousand of IOPS of Cassandra with 2 replicas vs Qbeast

Figure 3 shows the increase of performance when doubling the number of nodes in a cluster of Cassandra with replication set to 2, and with Qbeast storing data in the OTree and the original table. The level of availability of Cassandra and Qbeast in these settings is comparable. Indeed, the same hash value that determines in which server to store an element in the original Cassandra table, also represents the random priority in the OTree. As each node has an assigned hash range, we can query the OTree for the corresponding priority range to recover the missing information if a node is not available. As future work, we will modify the database partitioner to ensure the original and the Otree tables are not co-allocated so that we can achieve high-availability without additional replicas.

With two nodes, Cassandra and Qbeast perform very similarly, achieving respectively $\approx 84K$ and $\approx 83K$ IOPS. Cassandra and Qbeast approximately improve 80% when doubling the nodes. The scalability is not linear as the replica is synchronous, which adds latency and increases resource usage. When using a fire-and-forget approach for the replica, we have better scalability. However, to achieve linear scalability, we need a smarter client that directly forwards the requests to the correct node in the cluster, but that would require the client to be aware (at least approximately) of the current index status, which is an improvement that we plan as future work.

A. HPC integration

We use the OTree for a scientific use case that studies how to improve the assumption of drugs with inhalers by using Alya to simulate Lagrangian particles transported by fluids. A nontrivial task is integrating an MPI based application with an asynchronous TCP-based protocol. There are two main problems. The first is handling the asynchronous communication with a high enough level of parallelism that can exploit the distributed database and thus achieve excellent performance. To this end, we used the C version of Hecuba [15] an HPC oriented library that we develop in our research group. Hecuba allows efficient use of NoSQL databases in MPI oriented applications by taking care of all the callback and asynchronous management of messages.

In a physics simulation, it is common to split the space into smaller parts so that each worker can focus on its domain. After each timestamp, workers share information regarding the particles that moved from a domain to another. The downside of such an approach is that particles may concentrate on the specific area during part of the simulation. In our drug inhalers study, we used Alya to simulate the flow of particles from the inhaler's nose to the human bronchi. Therefore, the experiment starts with all drug particles residing in a limited area with consequentially an initial unbalanced workload between nodes.

To improve I/O without penalizing the full execution, we used a hybrid approach introducing an additional data shuffling step between workers on the same node, so that each worker participates equally in the writing process, taking better advantage of all available CPU resources. Using shared memory is a sub-optimal solution, but it serves the scope of our tests as the general goal is to reduce the number of synchronizations required for I/O. In the future, we will investigate more flexible solutions such as the integration with dynamic scheduling framework or more CPU friendly communication protocols.

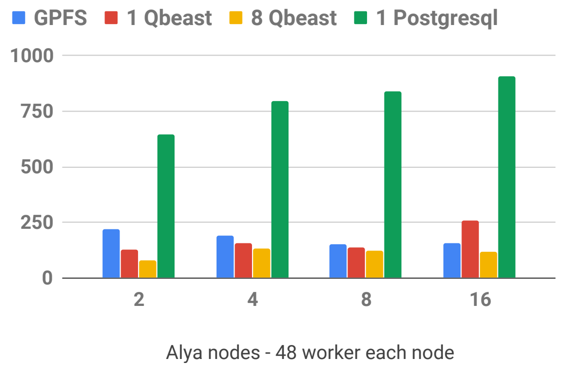

Fig. 4. Net I/O time for 1000 steps with different backends.

Figure 4 reports the net time Alya spent performing I/O with different backends when increasing the number of workers. There are several insights we can gather from these results. First, we shall note that Qbeast can store and index data faster than the GPFS can write into a not-indexed CSV file. As the reader may be surprised by such a result, we should clarify that

Alya uses a master-slave approach to output data into an ASCII file, which is arguably not the most efficient format. However, alternatives such as MPI/IO are not a perfect solution either. Indeed, the number of particles changes during the simulation as they might either deposit or move to another domain, thus making infeasible to use of Hyperslabing. Alternatively, each worker could write independently in a different file, but then a second phase of merging and reassembling the results is required. In any case, the point is not that our system is generally faster than file storage, but that when applications require specific file structures to facilitate analysis, our system can compete, if not be faster, then mere files. Another notable result is that time required for I/O for one Qbeast node or eight is not proportional as the I/O time of Alya remains approximately constant when varying the number of workers. Such behaviour suggests that either in the MPI3 shuffling or in the database communication, there is a performance bottleneck that we will investigate in future works.

Figure 4 also shows that PostgreSQL is considerably slower while ingesting writes and that its speed decreases when increasing the number of concurrent actors. For a fair comparison, we used the same MPI3 shared memory approach, scratch SSD and prepared statements for PostgreSQL. To improve the throughput, each worker commits only after storing the full timestamp, not after each insertion as in Cassandra. In such a way, the PostgreSQL driver can optimize the writing of the single particles.

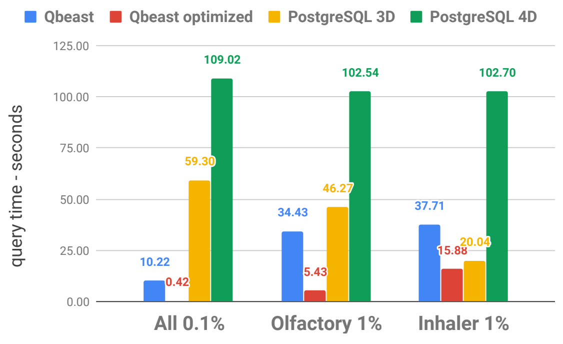

Fig. 5. Response time: Qbeast vs. PostgreSQL.

To evaluate the read performance, we selected three typical queries that scientists use when exploring the result of our particle inhalation problem. At first, scientists need to have an overall view of the whole simulation. Secondly, a relevant query is to see which particles deposited in a specific area of the nasal cavity, the olfactory region, where drugs get absorbed faster. Lastly, it is interesting to check how the particles get expedited from the nozzle of the inhaler.

Given the size of the areas of interest, we will gather only a sample of the data. More precisely, a sample of 0.01% of the whole simulation, while 1% of data in the other two queries. In the following, we will identify the queries as "all 0.01%", "olfactory 1%", "inhaler 1%". With such configuration, the three queries return 20, 27, 200 thousand results respectively.

Figure 5 compares the response time of different configurations of Qbeast and PostgreSQL. We analyze two status of the OTree: when all nodes of the OTree have $O = 0$, and when the OTree is optimized, and thus it has the same query performance of the D8tree. In the case of PostgreSQL, we can either use a 3D or a 4D secondary index. We can see how Qbeast always outperforms PostgreSQL with the optimized OTree. Also, even with the not-optimized OTree, Qbeast is faster than the PostgreSQL on two queries out of 3.

	speedup	RO runs	iterations	cube visited
All 0.01%	24.51	6	2	10
Olfactory 1%	6.34	10	19	19
Inhaler 1%	2.37	8	61	61

TABLE I
QBEAST SPEEDUP AFTER FEW *ReadOptimizations*.

Table I shows how the OTree improves after multiple *ReadOptimizations*. The table shows how many ReadOptimizations run before increasing the outlook of the part of the index interested in the three queries. It is crucial to note an RO execution for one query most likely benefits also others, thus reducing the overall number of RO required to achieve optimal performance. The table also reports the different speedup we can achieve in the three queries, ranging from 24.51 X improvement to a "mere" factor 2.37. In query "All 0.01%" we have the highest speedup as we benefit the most from the efficient sampling of the OTree. In terms of disk usage, a D8tee built on this dataset requires to replicate each item 5.29 times on average, while the optimized OTree only 1.14.

V. Related work

Simion et al. [16] discussed in their work *"The Price of Generality in Spatial Indexing"* how re-using existing solutions for one-dimensional indexing in spatial applications leads to sub-optimal performance in PostgreSQL. Kornacker et al. [17] reached a similar conclusion analyzing the use of generalized indexes in DB2/Common Server. Again, Eldawhy and Mokbel[1] elaborated a comprehensive survey of the existing solutions for big spatial data, and they described all existing approaches and their relative limitations. In particular, they showed that few solutions target dynamic indexing, and they only work for small point queries. To our knowledge, our system is the first that combines multidimensional indexing and efficient data sampling, but there are related works that target either the first or the second goal. Typically, approximate analytics achieves speed by relaxing the precision of the results within a specific interval of confidence, either via statistical "synopses" descriptors (e.g., wavelets, histograms, sketches...) or analyzing uniform random samples of the data. The second approach is preferable, as it enables complex queries such as joins, filters, and all types of aggregations. An example of a query engine with efficient sampling is BlinkDB [7], which extends Apache hive to build samples of large data sets in a batch fashion; thus, it does not support real-time indexing as Qbeast does.

Several works as HGRID[18], MD-HBASE[19] and the KR^{+}-index[20], have proposed different alternatives on how to combine both Quadtrees, Kd-trees, and R-trees with hybrid approaches where different indexes are used globally and locally, but none of these works support efficient sampling, and they do not solve the issue related to the change of data distribution and item popularity over time. Alternatives approaches, like the Quadboost [21], focus on multi-thread parallelism but they do not target distributed system. Regarding sampling geographic data sets, Sharma et al. [22] address how a randomized thinning algorithm for sets of points can respect the constraints of Visibility, Zoom Consistency, and Adjacency if we assign to each item a number - a priority - independently and uniformly at random. We use a similar approach, but we improve the index creation allowing the update of the index and query it in real-time while they use a batch approach.

VI. Conclusion

In this paper, we presented the OTree, a novel multidimensional index with efficient data sampling that runs on distributed key-value databases. We described our previous solution, the D8tree, and its limits dealing with write-intensive applications, and we studied how to reduce its transactional and storage requirements without compromising query performance. As a result, we proposed the OTree, which achieves high indexing speed by building at first a sub-optimal structure that gets opportunistically optimized in the background. We tested the performance and the scalability of the OTree, and we described its use in HPC. In particular, we described its integration with a medical use case simulation where we demonstrated that the OTree is not only convenient for users, but it also speeds up the execution, outperforming alternative databases, and files stored on a parallel file system.

As future work, we will study how to improve the performance of our system using adaptive query algorithms, predictive index optimization, and locality-aware clients. Finally, we believe a promising line of research is machine learning algorithms that use the indexing and sampling capability of the OTree to reduce the I/O requirements and speed up convergence.

Acknowledgements

This work has been supported by the Spanish Government (SEV2015-0493), by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), by the Generalitat de Catalunya (contract 2017-SGR-1414) and by the European Commission through the Horizon 2020 Research and Innovation program under contracts 720270 (Human Brain Project SGA1), 785907 (Human Brain Project SGA2), 780787 (I-BiDaaS) and 801331 (Quake).

References

[1] Ahmed Eldawy and Mohamed F. Mokbel. The era of Big Spatial Data. *2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016*, 10(12):1424–1427, 2016.

[2] Alfredo Cuzzocrea, Il-Yeol Song, and Karen C Davis. Analytics over large-scale multidimensional data: the big data revolution! *… 14th international workshop on Data …*, pages 101–104, 2011.

[3] Václav Snášel, Jana Nowaková, Fatos Xhafa, and Leonard Barolli. Geometrical and topological approaches to Big Data. *Future Generation Computer Systems*, 67:286–296, 2017.

[4] I. Arapakis, Y. Becerra, O. Boehm, G. Bravos, V. Chatzigiannakis, C. Cugnasco, G. Demetriou, I. Eleftheriou, J. Etienne Mascolo, L. Fodor, S. Ioannidis, D. Jakovetic, L. Kallipolitis, E. Kavakli, D. Kopanaki, N. Kourtellis, M. Maawad Marcos, R. Martin de Pozuelo, N. Milosevic, G. Morandi, E. Pages Montanera, G. Ristow, R. Sakellariou, R. Sirvent, S. Skrbic, I. Spais, G. Vasiliadis, and M. Vinov. Towards specification of a software architecture for cross-sectoral big data applications. In *2019 IEEE World Congress on Services (SERVICES)*, July 2019.

[5] NASA. Hubble Essentials: Quick Facts. bit.ly/2UPKQ5R.

[6] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. ApproxHadoop. *ACM SIGARCH Computer Architecture News*, 43(1):383–397, 2015.

[7] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. {BlinkDB}: Queries with bounded errors and bounded response times on very large data. *EuroSys*, pages 29–42, 2013.

[8] Seid Koric, Herbert Owen, Antoni Artigues, Ruth Arís, Guillaume Houzeaux, Daniel Mira, Fernando Cucchietti, Mariano Vázquez, Ahmed Taha, Hadrien Calmet, Jazmin Aguado-Sierra, Evan Dering Burness, José María Cela, and Mateo Valero. Alya: Multiphysics engineering simulation toward exascale. *Journal of Computational Science*, 14:15–27, 2016.

[9] Cesare Cugnasco, Yolanda Becerra, Jordi Torres, and Eduard Ayguadé. D8-tree. In *Proceedings of the 17th International Conference on Distributed Computing and Networking - ICDCN '16*, 2016.

[10] Antoni Artigues, Cesare Cugnasco, Yolanda Becerra, Fernando Cucchietti, Guillaume Houzeaux, Mariano Vazquez, Jordi Torres, Eduard Ayguadé, and Jesus Labarta. ParaView + Alya + D8tree: Integrating High Performance Computing and High Performance Data Analytics. *Procedia Computer Science*, 108:465–474, 2017.

[11] Cesare Cugnasco, Yolanda Becerra, Jordi Torres, and Eduard Ayguade. Exploiting Key-Value Data Stores Scalability for HPC. In *Proceedings of the International Conference on Parallel Processing Workshops*, pages 85–94. IEEE, 8 2017.

[12] Leslie Lamport and others. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[13] Barcelona Supercomputing Center. MN4 specs. (http://bit.ly/2Vm1A4c).

[14] Jens Axboe. Fio, flexible I/O tester. (https://linux.die.net/man/1/fio).

[15] Pol Santamaria, Lena Oden, Eloy Gil, Yolanda Becerra, Raül Sirvent, Philipp Glock, and Jordi Torres. Evaluating the benefits of key-value databases for scientific applications. In *Computational Science – ICCS 2019*. Springer International Publishing, 2019.

[16] Bogdan Simion, Daniel N Ilha, Angela Demke Brown, and Ryan Johnson. The Price of Generality in Spatial Indexing. *BigSpatial '13 Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 8–12, 2013.

[17] Marcel Kornacker, C Mohan, and Joseph M Hellerstein. Concurrency and recovery in generalized search trees. *SIGMOD '97 Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, 26(2):62–72, 1997.

[18] Dan Han and Eleni Stroulia. HGrid: A Data Model for Large Geospatial Data Sets in HBase. *2013 IEEE Sixth International Conference on Cloud Computing*, 2013.

[19] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. MD -HBase: Design and implementation of an elastic data infrastructure for cloud-scale location services. *Distributed and Parallel Databases*, 31, 2013.

[20] Ling-Yin Wei, Ya-Ting Hsu, Wen-Chih Peng, and Wang-Chien Lee. Indexing spatial data in cloud data managements. *Pervasive and Mobile Computing*, 2013.

[21] K. Zhou, G. Tan, and W. Zhou. Quadboost: A scalable concurrent quadtree. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[22] Anish Das Sarma, Hongrae Lee, Hector Gonzalez, Jayant Madhavan, and Alon Halevy. Consistent thinning of large geographical data for map visualization. *ACM Transactions on Database Systems*, 38(4), nov 2013.