Managing Big Interval Data with CINTIA: the Checkpoint INTerval Array

Ruslan Mavlyutov and Philippe Cudre-Mauroux
eXascale Infolab
U. of Fribourg—Switzerland
{firstname.lastname}@unifr.ch

Abstract—Intervals have become prominent in data management as they are the main data structure to represent a number of key data types such as temporal or genomic data. Yet, there exists no solution to compactly store and efficiently query big interval data. In this paper we introduce CINTIA—the Checkpoint INTerval Index Array—an efficient data structure to store and query interval data, which achieves high memory locality and outperforms state-of-the art solutions. We also propose a low-latency, Big Data system that implements CINTIA on top of a popular distributed file system and efficiently manages large interval data on clusters of commodity machines. Our system can easily be scaled-out and was designed to accommodate large delays between the various components of a distributed infrastructure. We experimentally evaluate the performance of our approach on several datasets and show that it outperforms current solutions by several orders of magnitude in distributed settings.

Index Terms—Interval Data; Low-Latency; Scalability; Distributed Data Management

1 Introduction

Infrastructures for managing sets of intervals have not evolved much in the last decades. Yet, intervals are increasingly prevalent in cloud and distributed information systems and are being created by an ever growing number of applications. Temporal intervals are fundamental to the management of time varying information [1]. They are for example essential for correctly capturing and handling online transactions. Several types of intervals can be defined in that context [2], such as transaction intervals (defining the time periods during which the tuples in the database are considered to be true) or validity intervals (capturing the period during which a database tuple is valid in reality). Intervals are also omnipresent when managing genomic data. With the commoditization of DNA sequencing technology, very large amounts of short DNA sequences can today be created at very low cost—the cost of sequencing a raw megabase (a million DNA bases) being less than 0.1$ in 2015[1]. Queries on those overlapping intervals have played a significant role in the development of modern genomics and are essential for many bioinformatics applications.

Beyond temporal information and genomics, intervals are often heavily used to efficiently manage spatial or multidimensional data, for example by taking advantage of space-filling curves (such as Hilbert [3] and Z-order [4] curves) to map a n-dimensional space onto a one-dimensional space while preserving some locality.

Despite this growing demand, interval infrastructures have been lagging behind. Relational database systems, with their bag (or set) semantics, are ill-designed to store ordered data and have to rely on secondary structures (such as interval trees, see Section 2 below) to handle interval data. A number of recent systems, such as column [5] or wide column [6] stores, have the ability to store ordered data natively; yet, they are unable to efficiently answer queries on top of interval data, as the complexity of answering queries on overlapping intervals grows linearly with the number of intervals without a dedicated index.

After having unsuccessfully tested out all the above solutions for one of our projects[2], we decided to develop a new solution to this problem. In this work, we introduce CINTIA, the Checkpoint INTerval Index Array, a new data structure to manage large sets of intervals in distributed settings. We designed our system from the ground up with two main goals in mind: i) to allow for low-latency, efficient query execution over large sets of overlapping intervals—even when the latency between some of the components of the infrastructure is high—and ii) to support graceful *scale-out*, by simply adding more nodes to the infrastructure when the system reaches capacity. We met those goals by developing a new interval index, which scales gracefully even in the worst-case scenario when all intervals overlap each other, and by integrating and optimizing our system directly on top of a modern distributed file system (i.e., HDFS).

In summary, the main contributions of this paper are:

- a new data structure to compactly encode and efficiently query interval data. Our new index has a construction complexity of $\mathcal{O}(N \log N)$ (where N is the number of intervals in a dataset), a query execution complexity of $O(log(N) + R)$ (where R is the number of intervals overlapping the query interval) and a space complexity of $O(N)$. Our index is cache-efficient, in the sense that cache-misses are minimized as much as

1. see http://www.genome.gov/sequencingcosts/

2. the 3D Genome Browser, see http://3dgb.cs.mcgill.ca/

possible thanks to data collocation;
- two implementations of our index, one for stand-alone, main-memory usage and another one built on top of a popular distributed filesystem (HDFS);
- an extensive evaluation of our system on several datasets showing that it is more efficient than state-of-the-art solutions both in centralized and in distributed settings.

The rest of this paper is organized as follows. We start below by discussing the related work in temporal, spatial and distributed systems in Section 2. Section 3 gives a high-level overview of our solution, including the main techniques we devised to insert, index and query interval data. We discuss a number of important implementation considerations in Section 4. Section 5 presents the results of an empirical evaluation of our system on several datasets, and how it compares to several state-of-the-art techniques and systems. Finally, we conclude in Section 6.

2 Related Work in Interval Data Management

A number of techniques have been proposed for managing intervals efficiently (see [7], [8], or [9] for surveys), mostly for relational database systems. To the best of our knowledge, however, we are the first to propose a solution for managing interval data based on two interrelated arrays: a main interval array and a checkpoint array.

We review below the most important pieces of work in our context, focusing on main-memory and multidimensional structures and their extensions to secondary storage and distributed settings.

2.1 Memory-Resident Structures

The trivial solution to resolve overlap queries (i.e., queries returning all intervals overlapping a chosen interval) on a collection of intervals is to do a full-scan on the set/list of intervals, which has a time complexity of $\mathcal{O}(N)$, where N denotes the number of intervals in the collection. Keeping the intervals ordered does not help to improve this query complexity.

A number of data structures were developed to execute overlap queries in logarithmic time.

The original Interval Tree by Edelsbrunner [10] indexes intervals on a line by splitting the tree recursively based on the median of the intervals. The result is a ternary tree, with each node storing i) pointers to intervals lying completely to the left and to the right of the point corresponding to the node and ii) two lists storing all the intervals overlapping the current point, one sorted on the intervals' starting points, the other sorted on their ending points. The Interval Tree has a space complexity of $\mathcal{O}(N)$, and a query complexity of $\mathcal{O}(M + \log N)$, where M is the number of intervals overlapping the query.

The Relational Interval Tree (RI-Tree) [11] is a related effort leveraging common database structures. In its core, it uses Edelsbrunner's Interval Tree, although intervals are internally managed by two relational indices. Another adaptation of the Interval Tree for external memory is made by Arge *et al.* [12]. Since this data structure is able to efficiently serve only stabbing queries (return all intervals overlapping a point), we do not consider it in our research.

A number of similar data structures were developed at the time in the context of computational geometry, among which the Priority Search Tree and the Segment Tree [13]. The Interval Binary Search tree [14] handles point queries efficiently and can be balanced easily although it incurs higher storage costs ($\mathcal{O}(N \log N)$). More recently, the Interval Skip List [15] allowed for efficient online searches, insertions, and deletions, yet was simpler to implement than previous structures.

Another recent approach proposed for genome alignment databases is the Nested-Containment List (NCList) [16]. The basic idea is to keep intervals fully enclosed by other intervals as their sublists. The result is a tree-like structure of lists, which yields a query complexity of $\mathcal{O}(M + \log N)$ and a construction time of $\mathcal{O}(N \log N)$.

We experimentally compare CINTIA to the most relevant data structures described above in Section 5.

2.2 Secondary Storage Structures

A number of secondary storage structures have been presented in the literature to overcome the limitations of memory-resident interval structures. Many of such structures extend secondary structures from relational database systems. The Time Index [17], for instance, leverages B+-trees to index time intervals. It maintains an ordered list of points in the B+-tree, each pointing to a set of intervals. As each interval can be indexed by several points, the space complexity is in this case $\mathcal{O}(N^2)$ [15].

Subsequently, Ang & Tan developed the Interval B-tree [18] to improve on the Time Index. The Interval B-tree implements Edelsbrunner's Interval Tree by extending a B+-tree. Hence, it retains the original properties of the Interval Tree while leveraging disk-efficient structures. However, the Interval B-tree keeps the complex tripartite structure of the original Interval Tree and adds an additional structure of its own on each level.

The Relational Interval Tree (RI-Tree) [11] is a related effort leveraging common database structures. In its core, it uses Edelsbrunner's Interval Tree, although intervals are internally managed by two relational indices. We do not include the RI-Tree or other structures extending relational engines in our experiments, since they are expected to show worse performance than the Interval Tree on which they build (e.g., due to the high number of seeks and cache misses caused by B+-trees and n-ary storage structures) and since we do not rely on relational systems for our own solution.

2.3 Multi-Dimensional Solutions

Intervals can also be indexed using general-purpose multidimensional indices. Guttman's R-tree [19] is often used to store one-dimensional intervals in practice. A number of variants were developed over the years, such as the R*-tree [20], which minimizes the overlap for leaf nodes and

which can efficiently index both point and spatial data at the cost of a more expensive insertion strategy.

The Segment Index [21] combines the main memory-based segment tree with the secondary structure of the R-tree and was specifically designed to improve query performance for intervals that have non-uniform length distributions. Its performance is pretty similar to the R-tree in practice.

More recently, Fenk *et al.* proposed a hybrid method to manage and query intervals efficiently, by transforming intervals into a two dimensional space and indexing that space with a UB-Tree [22]. Queries on intervals are then transformed into two-dimensional boxes, which are then handled by the UB-Tree range query algorithm [4]. It performs close to the RI-tree for point queries in practice.

Other multidimensional methods, such as our own previous work on indexing two-dimensional trajectories [23] could also be adapted to index one-dimensional intervals. We compare our solution to different R-trees configurations in Section 5.

2.4 Distributed Settings

Less focus has been given to the design of interval management techniques for distributed environments. Bisadi & Nickerson [24] proposed a distributed range search mechanism for sets of points distributed on several nodes based on rainbow skip graphs. This work leveraged message-passing interfaces and focused on reducing the number of messages, achieving a message complexity of $\mathcal{O}(N)$, where N is the number of computing nodes in the distributed setting.

The authors of [25] adapted the NClist to cloud environments and proposed optimization strategies to filter and query intervals. Unfortunately, we were not able to obtain the source code of their solution.

The SD-Rtree [26] is a generalization of the R-tree for distributed environment. Each machine in the cluster represents one leaf node, which keeps a collection of multidimensional objects. The SD-Rtree balances the servers load and yields low message costs for executing insertion and search queries. To the best of our knowledge, this structure was never implemented. In addition, the authors did not propose any efficient method for storing and searching objects in the leaf nodes, which makes this solution incomplete.

SpatialHadoop, proposed in [27], is a popular implementation of spatial data structures and algorithms on top of Hadoop. It segments a dataset into compact subsets of collocated objects and puts each subset into a serializable in-memory data structure. The serialized dictionaries are then distributed across the cluster nodes. Queries in SpatialHadoop are handled through distributed MapReduce tasks, where dictionaries are uploaded into memory and queried. We benchmark CINTIA against SpatialHadoop in Section 5.

3 METHOD

We designed our interval system with two challenging goals in mind: i) high scalability and ii) very low latency. We achieved i) by leveraging the fault-tolerance and data partitioning of a state-of-the-art distributed file system and by creating two complementary data structures to store and index intervals compactly while providing a high degree of spatial locality. We achieved ii) by taking advantage of high-throughput, buffered reads from the distributed file system while minimizing the number of data accesses to the bare minimum (most queries in CINTIA can be answered by only two accesses to the underlying filesystem).

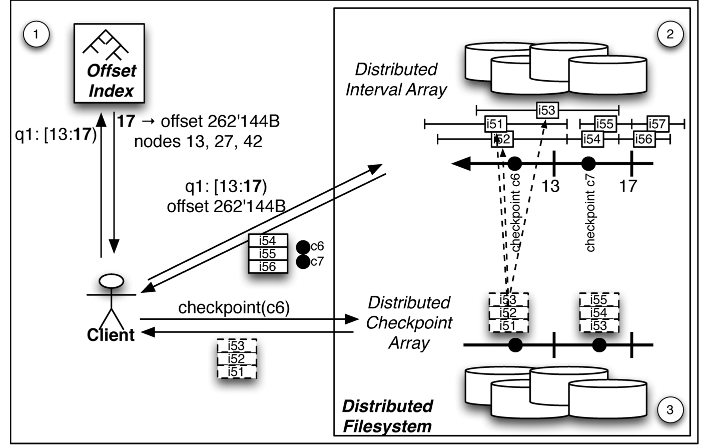

Fig. 1: CINTIA overview: querying starts by issuing a query to a segment index (1) returning, for any interval, the offset of the file segment corresponding to the right-bound of the interval. The client then buffer-reads all intervals whose left bounds fall between the two bounds of the query (2) from a distributed interval array. Finally, the client retrieves the remaining intervals that overlap the query but started before its left bound (3) from a corresponding checkpoint array.

Figure 1 gives a high-level overview of our system. Clients can send interval queries to our system over the Network. First, clients issue a query to an in-memory skip list to determine the segment in the *distributed interval array* responsible for indexing the right bound of the query in the file system. The skip list is small even for very large interval sets, and can either be placed on the directory (e.g., NameNode) of the underlying file system or be cached on the client-side. The client then locates the corresponding machine in the cluster and starts reading a distributed interval array sequentially, starting at the file segment corresponding to the right bound of the query and continuing backwards until it reaches the file segment corresponding to the left bound of the query. At this point, the client collected all intervals whose left bounds overlap the query. Then it continues backwards until it reaches a *checkpoint record*, pointing to lists of intervals overlapping the query but starting before its left bound. The client retrieves checkpoint data (if needed) from a second distributed array, the *checkpoint array* (3). At this point the client retrieved all intervals overlapping the query.

We describe our data structure as well as our construction and query processing algorithms in more detail below, after introducing some notations.

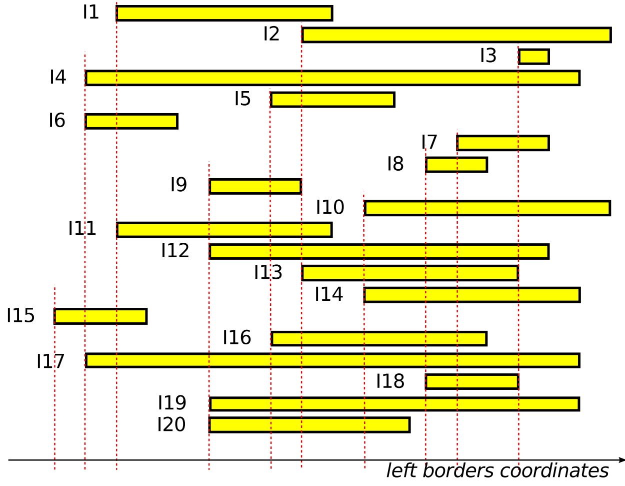

Fig. 2: Example interval set

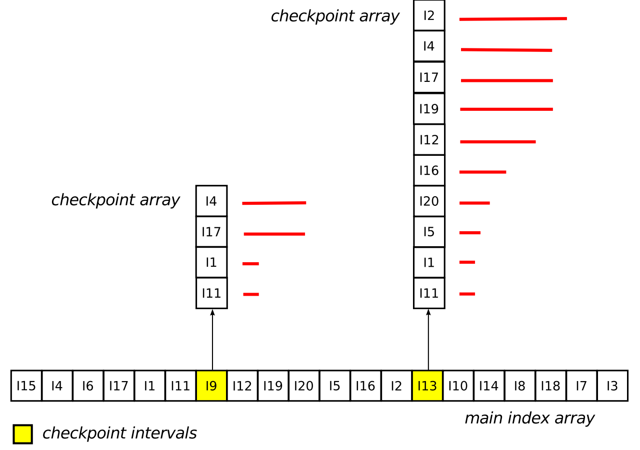

Fig. 3: Corresponding checkpoint interval index

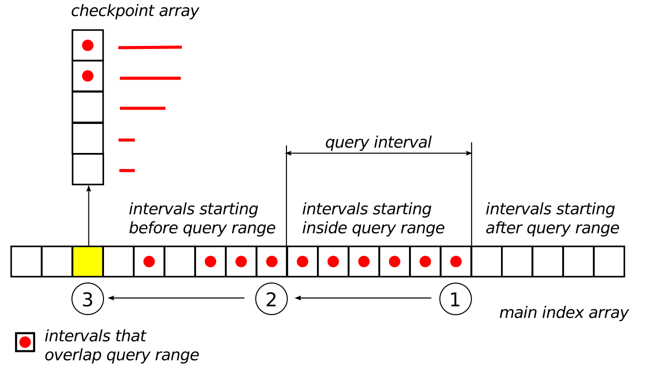

Fig. 4: Query example

3.1 Notations

We consider large sets of N intervals $\mathcal{I} = \{i_1, \ldots, i_N\}$. All the intervals we consider in the following are left-bounded, left-closed and right-open[3]. An interval is defined by its start and end points (left and right bounds) $[i_s, i_e)$, and by its associated payload or value i_v. We consider overlapping queries which take the form of interval queries $[q_s, q_e)$ retrieving all the intervals overlapping the region between their start and end points.

3.2 Data Structures

We introduce two main data structures:

1) an *interval array* (*main index array*) storing all intervals (along with their values) sorted by the value of their left bound in ascending order;
2) a sequence of *checkpoint arrays*, storing for every *k-th* record (called checkpoint) in the interval array copies of the records that overlap its interval, but that are located before it in the interval array. The records in the checkpoint arrays are sorted in descending order of the right bounds of their corresponding intervals.

Both the interval and the checkpoint arrays are cache-efficient, compact structures providing a high degree of spatial locality. They both can be easily distributed on several machines for fault-tolerance and scalability. We consider in the following that the value i_v attached to each interval is small and of fixed-size. In case of large values (e.g., for multimedia files indexed using intervals) or values varying in length, we store pointers to the values instead of the values themselves in both arrays in order to reduce the storage overhead.

We illustrate our two main data structures with an example. Figure 2 gives a set of 20 intervals $\mathcal{I} = \{i_1, \ldots, i_{20}\}$. Figure 3 illustrates how a checkpoint interval index would store and index those intervals. The 20 intervals are sorted by the value of their left bound in the interval array (bottom part of Figure 3). Each kth interval in that array is selected as a checkpoint (yellow record in Figure 3). There might be various strategies for selecting k, and thus the number of checkpoints, which directly influence the performance of our approach. We discuss how to find an optimal value of k below in Section 3.6.

For the first checkpoint corresponding to interval $I9$, the checkpoint array contains records $I1$, $I4$, $I11$ and $I17$. All of them overlap $I9$ (see Figure 2) and are located before it in the main interval array. For the same reason $I12$ is not in the checkpoint array, though it overlaps $I9$. The order in the checkpoint array is defined by the right bounds, so the intervals $I4$ and $I17$ occupy the first two positions.

3.3 Overlapping Profile

We now introduce the notion of *Overlapping Profile* OP, a function that for every record in the Main Index Array puts in correspondence the number of records that are located on its left-hand side (i.e., which have a lower index in the array) and overlap with it. We call the average value of OP *Average Overlapping* AO.

3.4 Space Factor

To quantify the space overhead caused by the checkpoint arrays, we also introduce the notion of *Space Factor* SF, which is defined as the relative space overhead of our indices to the original size of the data considered:

3. We note that extending our method to other types of intervals would be straightforward

$$SF = (size(Index) - size(Data))/size(Data). \quad (1)$$

SF can also be represented as the sum of all OP values for the checkpoint records divided by the total number of intervals stored in the index.

3.5 Query Execution

Given the two main structures described above, Algorithm 1 describes the query execution strategy to retrieve all intervals that overlap a query q. Figure 4 illustrates it. On the figure, the intervals that overlap the query—and thus must be returned—are denoted by red dots.

Algorithm 1: Query algorithm

```
Input: main interval array M, array of checkpoint
       arrays Ch, checkpoint step ChS, query [q_s, q_e)
Output: R - set of intervals overlapping [q_s, q_e)
/* via binary search or query to B-tree:
   */
POS ← Index of last interval starting before q_e
while POS ≥ 0 do
    if M[POS].end > q_s then
        R.add(M[POS])
    if M[POS].start < q_s and IsCheckpoint(POS)
     then
        ChIndex ← M[POS].checkpointArrayIndex
        ChPOS ← 0
        while ChPOS < Ch[ChIndex].size do
            if Ch[ChIndex][ChPOS].end ≤ q_s then
                break
            R.add(Ch[ChIndex][ChPOS])
            ChPOS ← ChPOS + 1
        break
                                        // Full stop
    POS ← POS − 1
return R
```

The example query q is processed as follows: First, the system determines the last interval in the interval array that starts before the right bound of the query q_e (by using binary search or a skip list). This record corresponds to point (1) in the figure. Next, the system moves back in the interval array and retrieves all intervals until it reaches an interval whose left bound starts before q_s (point (2) on the picture). At this point, the system has retrieved all intervals starting inside the query. The system continues reading the interval array backwards, until it reaches a checkpoint (point 3 on the picture). The intervals read in that phase (i.e., the intervals stored between (2) and (3)) may or may not overlap the query. Each of them is checked by the system and returned as a result only when it indeed overlaps the query. Finally, the systems reads the checkpoint array. Intervals in the checkpoint array, whose right bounds are larger than q_s, overlap the query and are also returned as results. As the intervals in the checkpoint array are sorted by their right bound, the algorithm scans only those records that overlap the query, and stops as soon as it comes across an interval whose right bound is smaller or equal to q_s. The main performance overhead of this procedure in incurred when walking from point (2) to point (3) in cases where there is a prevalence of intervals that do not overlap the query. The length of the walk between (2) and (3) is however limited by the size of the checkpoint step (ChS), which is discussed below in Section 3.6.

Soundness and Completeness. Algorithm 1 is sound, as it only retrieves intervals that overlap the query.

Proof. We have three groups of retrieved intervals:

(i) starting inside the query interval. They overlap the query by definition.

(ii) taken from the main array, and starting before the query interval.

(iii) taken from the checkpoint array, and starting before the query interval. They overlap q under the same condition. ■

Algorithm 1 is also complete, in the sense that it guarantees to retrieve all intervals that overlap the query.

Proof (by contradiction). Let us assume that there is an interval i in the index, that overlaps the query, but was not returned as a result by our algorithm. i's position in the interval array can either be:

i) after the right bound of query q_e (after point (1) in Figure 4); in that case, it cannot overlap the query since in that case i_s would be equal or greater than q_e.

ii) between q_e and the most recent checkpoint (between (1) and (3) in Figure 4). Algorithm 1 scans all entries in that range, so this is impossible also.

iii) before the checkpoint (before (3) in Figure 4). Since i starts before the checkpoint interval and overlaps q, it should also overlap the checkpoint. Consequently, i must be stored in the corresponding checkpoint array and would be considered as a result by Algorithm 1. ■

Query Complexity Analysis. Query execution consists of 3 phases:

1) searching for the last interval starting before q_e ((1) in Figure 4). This can be performed in logarithmic time $\mathcal{O}(\log N)$ (either using an index or by binary search since the main index array is sorted);

2) traversing the interval array backwards to reach the first checkpoint before q_s ((3) in Figure 4). The number of intervals that are linearly scanned is limited by the number of intervals in the response set $\mathcal{R}$ and the distance between two checkpoints ChS;

3) traversing the intervals in the checkpoint array. Here, the number of intervals to scan is limited by the number of remaining intervals in $\mathcal{R}$ that have not yet been retrieved.

The total complexity is hence $\mathcal{O}(\log N + R + ChS)$, which boils down to $\mathcal{O}(\log N + R)$. We provide a proof for that fact below; we restrict ourselves to the case where ChS is a power of 2 for brevity (it is however easy to extend this proof to arbitrary values).

Proof. The strategy of selecting the optimal value of $ChS = 2^n$:

1) In the beginning we consider every record in the main index array as a checkpoint. In that case the SF of the index will be:

$$SF_0 = N^{-1} \sum_{\forall i} OP_i = AO \tag{2}$$

2) On each iteration we start with an ordered group of checkpoints. The divide the group on two in the following manner: all checkpoints with an odd index go to the first group, and all the others go to the second group. Among the two subgroups we choose the one that yields the lowest value of the SF. After the first split we will get SF:

$$SF_1 \leq (2 * N)^{-1} \sum_{\forall i} OP_i = AO/2 \tag{3}$$

3) We repeat the operation 2 until we get a group of checkpoints with the $SF \leq SF_{max}$. Assume there were k such splits in total. Then we will have:

$$SF_k \leq AO/2^k \tag{4}$$

$$SF_k \leq SF_{max} < SF_{k-1} \leq AO/2^{k-1} \tag{5}$$

Since $ChS = 2^k$:

$$SF_{max} < (2 * AO)/ChS \tag{6}$$

$$ChS < AO * (2/SF_{max}) \tag{7}$$

At the same time AO is the expected value the amount of intervals that overlap the left bound of the query interval and located before it. So the value of AO is bounded by the expected amount of the query results R (intervals that overlap the query interval):

$$AO \leq E[R] \tag{8}$$

Groupping the last two equations:

$$ChS < AO * (2/SF_{max}) \leq R * (2/SF_{max}) \tag{9}$$

So the ChS is bounded by the value of R multiplied by the constant $2/SF_{max}$. The query complexity formula $\mathcal{O}(\log N + R + ChS)$ can be reduced to $\mathcal{O}(\log N + R)$ ∎

3.6 Optimizing the Checkpoint Step

There exist many potential strategies for selecting positions for the checkpoints. In the context of CINTIA, we decided that the very first record in the main index is a checkpoint, and subsequent checkpoints are positioned after every ChS records. In the following, we call ChS the *Checkpoint Step* as it denotes the number of records between two consecutive checkpoints. Our motivation behind this choice was to minimize space consumption (we do not need any additional field), to ease query process (no need to support a dedicated data structure signaling the position of the closest checkpoint) and to support optimized processes for the placement

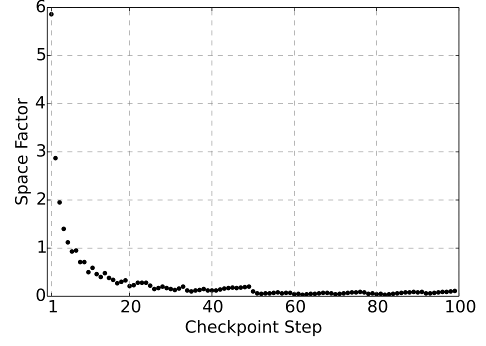

Fig. 5: Space Factor follows a decreasing hyperbolic trend

of the checkpoints (see below). We also implemented further techniques for setting the locations of the checkpoints (e.g., variable checkpoint steps), but do not describe them in this paper because of space constraints.

The value of the checkpoint step ChS has a strong influence both on the efficiency of our structures and on its memory consumption. On one hand, high values of ChS create some significant overhead when answering queries as discussed above in Section 3.5. The lower the degree of overlap between the intervals, the higher this overhead. In the worst case—when intervals do not intersect each other and queries are uniformly distributed—this overhead is equal to $ChS/2$.

On the other hand, lower values of ChS lead to a higher memory consumption (higher value of Space Factor) due to the creation of multiple checkpoint arrays.
In practice, the value of SF is limited by disk or memory constraints SF_{max}. The minimum value 0 corresponds to the case when we do not keep any additional information on top the original data set.

Hence, we are faced with a discrete optimization task under constraint, which can be summarized as follows:

$$\begin{aligned} \text{min} \quad & ChS \\ \text{subject to} \quad & SF(ChS) \leq SF_{max} \end{aligned} \tag{10}$$

Due to the very nature of our system, all space overhead are caused by the checkpoint arrays. The lower the value of ChS, the more arrays will be created, and (in general) the more memory will be consumed. The function $SF(ChS)$ is discrete and non-monotonic but follows a decreasing hyperbolic trend. Figure 5 shows an example of a Space Factor profile for a dataset of 10'000 intervals with fixed length and uniform distribution of left borders. The further the distribution of the intervals' borders moves from a uniform distribution, the less monotonic $SF(ChS)$ becomes.

The space factor SF can also be expressed as the ratio between all records stored in the checkpoint arrays and the size of the interval array N (assuming a record in the index array is of the same type as in the checkpoint arrays):

$$SF = N^{-1} \sum_{\forall\ checkpoint\ arrays\ ca_i} size(ca_i). \tag{11}$$

We introduce below a scan algorithm to find the optimal value of ChS. The complexity of this algorithm is

$\mathcal{O}(Nlog(N))$ (where N is the size of the dataset), such that the algorithm can be applied to very large data.

We start by introducing the notion of *overlapping profile*. The overlapping profile stores, for each record in the main index, the number of records that are located beforehand in the index and that overlap its left bound. We can build this profile in $\mathcal{O}(Nlog(N))$ time using the following algorithm (Algorithm 2).

Algorithm 2: Overlapping profile construction

```
Input: main interval array M (intervals are sorted by
       left bound)
Output: OP - overlapping profile
RBoundHeap ← []        // heap, storing right
 bounds
for POS ← 0 to M.size − 1 do
    while RBoundHeap.size > 0 and
     RBoundHeap.Min ≤ M[POS].left do
        RBoundHeap.PopMin()        // O(log(N))
    OP.Add(RightBoundHeap.size)
    RBoundHeap.Insert(M[POS].right)
     // O(log(N))
return OP
```

The algorithm swipes across the main index array and keeps a heap of the records' right bounds. At each step, we remove the right bounds that are smaller than the left bound of the current record. Subsequently, the size of the heap equals the number of already visited records that overlap the current record, which gives us the next value for the overlapping profile. Since each record will be pushed and popped from the heap once (pushing an element and popping the minimum from the head have logarithmic complexity) the final complexity of that step is $\mathcal{O}(Nlog(N))$.

Once the overlapping profile of the interval array is built, we can determine the optimal checkpoint step through Algorithm 3. The algorithm uses Equation 11 to estimate the value of the space factor. According to our optimization goal (10), the optimal value will be the minimum value of ChS that yields a space factor smaller or equal to SF_{max}.

Algorithm 3: Optimal checkpoint step

```
Input: overlapping profile OP
Output: ChSOpt - optimal checkpoint step
for ChSOpt ← 1 to M.size do
    ChArraysSizesSum ← 0
    for POS ← 0 to M.size − 1 step ChSOpt do
        ChArraysSizesSum ←
              ChArraysSizesSum + OP[POS]
    SpaceFactor ← ChArraysSizesSum/OP.size
    if SpaceFactor ≤ SF_max then
        return ChSOpt                 // Full stop
```

The algorithm considers different possible values of ChS starting with the lower values. For each value, it computes the sum of all elements in the overlapping profile that correspond to checkpoints. That sums constitute the total number of records stored in the checkpoint arrays for each possible value of ChS. By dividing the result by the size of the main index, we obtain the corresponding space factor (according to Equation 11). We stop as soon as the algorithm finds a value of ChS yielding an appropriate space factor, thus satisfying Equation 10. The complexity of this step is also $\mathcal{O}(Nlog(N))$ as shown below.

Proof. The number of checkpoints for a checkpoint step of size ChS is $\lfloor N/ChS \rfloor$. The total number of positions to be considered is a partial sum of a series:

$$\sum_{i=1}^{N} \lfloor N/i \rfloor < \sum_{i=1}^{N} N/i.$$

The last sum is a partial sum of a harmonic series; according to Euler's formula:

$$\sum_{i=1}^{N} \lfloor N/i \rfloor < ln(N) * N.$$

The final complexity of the algorithm is $O(Nlog(N))$. ■

3.7 Index Construction

Batch Construction. The two main structures of our system can be built efficiently when considering batch insertions. The index construction can be broken into three steps:

1) the construction of the interval array storing the intervals sorted by their left borders. Sorting this array has $O(NlogN)$ complexity;
2) the selection of an optimal value for the checkpoint step; the complexity of this step is also $O(NlogN)$, as described above in Section 3.6.
3) the construction of the checkpoint arrays, which is described below.

The construction of the checkpoint arrays is analogous to the calculation of the optimal value of the checkpoint step. We swipe across the interval array and use a heap to store already visited records. We use the right bound of the record's interval as a comparison key for the heap. For each new record, we pop all records from the heap that have the right bound smaller than the left bound of the current record's interval. After this step, the heap contains all records that are located to the left of the current record in the interval array and overlap it. Upon reaching the next checkpoint, we make a copy of the heap and pop all values in order to get the list of records to put in the checkpoint array, sorted by their right border. We just need to reverse that list to get a final checkpoint array.

Since each interval is pushed and popped once from the heap (pushing an element and popping the minimum have complexity $O(log(N))$), the complexity of this process is $\mathcal{O}(Nlog(N))$. The amount of times the initial intervals are copied to checkpoint arrays is limited by the Space Factor and is bounded by $SpaceFactor * N$. The final time complexity of the construction is hence $O((SpaceFactor + log(N)) * N)$.

The total space complexity is $O((SpaceFactor + 1) * N)$, which might be reduced to $O(SpaceFactor * N)$. During construction, the overlapping profile has N elements, and the heap never stores more than N records, so it does not affect the space complexity.

Append Operations. Appending new intervals that start after (or at the same time as) the last considered interval is straightforward. The append operation resembles the construction process described above, the main differences being that we directly add a new interval to the main array and that we maintain a heap of currently opened intervals to support online append operations.

Random Inserts. Random inserts in CINTIA are expensive, as entire portions of the arrays might have to be rewritten. This is a common issue of read-optimized structures. One standard way of coping with this issue would be to consider an in-memory, write-optimized store that handles all random inserts, and to periodically merge the write-optimized and the read-optimized stores as implemented in recent column stores [5].

4 IMPLEMENTATIONS

We implemented two fully-functioning prototypes of CINTIA, one is for testing our index in main memory, and one for large-scale deployments on top of the HDFS file system. Both versions are freely available online[4].

4.1 In-Memory Implementation

The in-memory version of CINTIA was developed in C++, and is a direct implementation of the structures and methods described above in Section 3. $std : vector$ is used in this version to implement both the interval and the checkpoint arrays. The first stage of query processing, i.e., finding the last interval in the interval array that starts before the right bound of the query, is simply resolved by binary search in this case.

4.2 Distributed Implementation

The distributed version was built on Hadoop 2.3. Like many recent distributed file systems, the Hadoop File System (HDFS) splits files in large, replicated blocks. The block size is adjustable and is equal to 128MB by default. When a read is initiated, the client first queries a *NameNode* to get the required block location. It then picks a *DataNode* storing one of the block replicas, creates a TCP connection and download the desired chunk of data from that replica. Information about block locations and TCP connections are subjects to caching, such that the number of calls to the NameNode are minimized over time.

The read proceeds by iteratively transferring packets. The size of the transferred packets is limited both on the lower end (by default 512 bytes) and on the higher end (by default 64KB). When requesting more data than what can fit in a packet, several packets are transferred iteratively.

4. https://github.com/XI-lab/interval_index.git

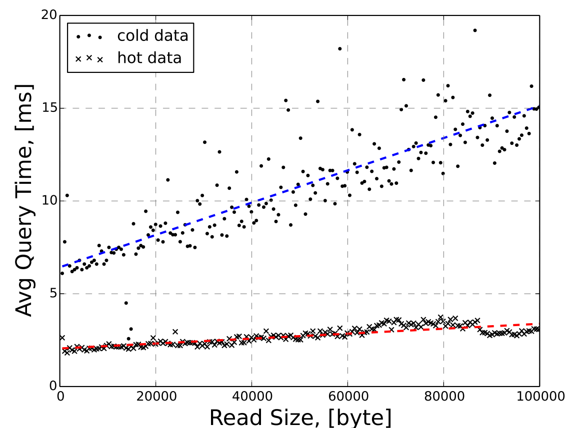

Fig. 6: HDFS read performance for hot and cold data.

The fixed costs associated to initiating a read are thus high: contacting the NameNode and initiating a TCP connection to a DataNode if the location and connections are not yet cached, then transferring the packet(s) from the DataNode. One of our main goals with CINTIA is hence to minimize the number of requests to the filesystem in order to amortize those initial costs. Figure 6 illustrates this point by plotting the average time needed to read blocks of varying size in the cluster we used for our experiments (described in more detail in Section 5).

The top curve illustrates the read performance when requesting cold data. In that case, a new connection has to be established to the datanode. The bottom curve illustrates the performance when requesting hot data after we warmed-up the cluster. In both cases, we selected large data segments randomly. As we can observe from the figure, the start-up costs of the reads are high but can be amortized when requesting larger data segments. Taking this into consideration, we implemented CINTIA as follows on top of HDFS:

- the main interval array is serialized in a file. Records in that file are of fixed length, and consist of three fields: the left and right bounds of the interval and a value field (that can also be a pointer in case of variable-length or large payloads). The checkpoints in this array do not contain any interval record. Instead, they store the position to the corresponding record in the checkpoint file (see below), as well as the value of the right border of the first interval in the checkpoint array. The last value allows to avoid reading from the checkpoint file in case there is no intervals that overlap the query;
- the checkpoint arrays are stored in a second file, which has the same structure as the interval array (sequence of fixed-size records). Each checkpoint array is stored sequentially and ends up with a null-record;
- index construction is performed through a sequence of MapReduce tasks;
- we perform reads from the index files by blocks of records. As can be seen from 6, on a warm cluster there is no big difference between reading a small and a big packet. Our experiments on HDFS show that there is no significant difference when reading shorter or larger

segments on a warm cluster and when TCP-connections are cached. Also, we consider the chunk processing time to be negligible compared to its download time. Hence, we read as much data as possible during each read from the cluster. Specifically, we chose a reading block size equal to the maximum packet size in the cluster;
- unlike the in-memory implementation, we do not perform a binary search on the main interval file during query execution, which would require $\mathcal{O}(\log N)$ calls to HDFS. Instead, we keep an in-memory skip list of the main interval file read blocks (described above). For each block, we keep its offset in the file and a left-bound value of the first record interval. One record in the skip list consumed 16B, such that the skip-list size for 1 billion records in only 7.5MB. In our implementation, clients are pre-loading the skip list upon start-up.

Distributed query processing proceeds in general as described in Algorithm 1. The first part of the query processing is resolved by looking-up the skip list, in order to get the location of the reading block storing the last interval starting inside the query. Next, and according to the algorithm, we keep uploading blocks from the main interval file until we reach a checkpoint record, at which point we switch to the checkpoints file.

When an optimal value of ChS is selected (see Section 3.6), most queries can be executed through only one or two HDFS reads (one for the main array and one for the checkpoint array). For larger queries, this method also allows us to return the first results early, and then to continue returning results as we read more intervals from the distributed file system.

While we implemented the above in HDFS only, we note that the same structures and techniques could be adapted to many recent distributed file systems such as OneFS[5], QFS[6], or to the Google File System [28].

5 EXPERIMENTAL EVALUATION

We empirically assess the performance of our index below. We proceed in two steps: First, we start by a series of micro-benchmarks testing our in-memory solution against a number of other interval structures with varying parameters. Then, we deploy CINTIA in a distributed setting and compare it to state-of-the-art solutions both on real and synthetic data.

5.1 Experimental Setup

Datasets & Queries: We use various interval datasets in order to test our approach. In addition to standard parameters like number of intervals (N), we consider two specific parameters:
- the *overlapping profile* giving, for each interval in a dataset, the number of intervals that started before the interval and that are overlapping it (see Section 3.6).
- the *average overlapping* standing for the average of the overlapping profile values. This parameter characterizes the density of a dataset, which is important for our solution, since it directly influences the optimal value of the checkpoint step (higher density leads to a bigger value of ChS).

Unless stated otherwise, all synthetic datasets in our experiment contain 1M intervals and have a uniform distribution of left borders in $[0,\ 10M)$. The lengths of the intervals follow a normal distribution with a mean value of 100 and a standard deviation 10. Those parameters yield an average overlapping of about 10.

To assess the performance of query execution, we issue 100K interval queries for each setup. The queries have their left border uniformly distributed between $[0,\ 10M)$ (similarly to the intervals) while their length is by default equal to 100. We repeat each experiment 10 times. Confidence intervals are not shown on the graphs, since for a confidence level of 95% they were negligibly small.

We also performed micro-benchmarks on a number of real datasets; due to space limitations, we refer the reader to the full version of this paper [29] for results pertaining to those experiments. For micro-benchmarks, we also use two real datasets:
- a bioinformatics dataset taken from the 1'000 Genomes Project[7]; In bioinformatics, scientists routinely query for the degree of overlap between genomic features[8]. In that context, we took the exome alignments for Chromosome 11 from sample #NA12891 as a dataset, and the exome capture targets for the same chromosome as queries. This represents a real use-case with around 11M intervals with an average overlapping of 89 and more than 10K interval queries.
- log data taken from a Hadoop cluster; We picked one of our own Hadoop clusters and logged runtime information using the Java `jstack` utility on all cluster machines every 15 seconds. We then exported all running Java functions as time intervals. The resulting dataset contains a dense set of 1.3M time intervals (Java functions) with an average overlapping of 5'610. We generated a set of 100K queries that are uniformly distributed and have a fixed length of 100 to test our algorithms on this dataset.

Compared Approaches: We compare our approach—implemented as described in Section 4—with the following popular interval data structures:

Interval Tree: We use Erik Garrison's canonical implementation[9] of the data structure.

NClist: The novel approach proposed by [16] and based on the idea of nested lists. We use the implementation provided by the authors.

R-Tree: We use Greg Douglas' C++ implementation[10]. This version has two optional parameters: minimum and maximum elements in node. We experimented with

5. http://www.emc.com/storage/isilon/isilon-onefs-operating-system.htm
6. http://quantcast.github.io/qfs/
7. http://www.1000genomes.org/
8. see for instance http://bedtools.readthedocs.org/en/latest/content/tools/intersect.html
9. https://github.com/ekg/intervaltree
10. https://github.com/nushoin/RTree

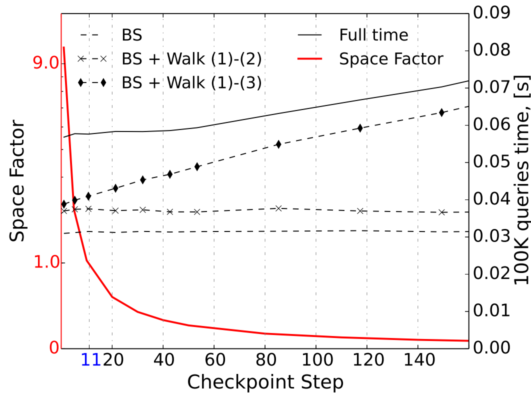

Fig. 7: Performance and memory consumption as a function of the value of the checkpoint step. BS: binary search, (1) rightmost interval starting inside the query interval, (2) rightmost interval starting before the query, (3) closest checkpoint to the query.

different values and picked the values yielding the best performance in our context (i.e., 32 and 64 for minimum and maximum number of elements respectively).

Segment Tree: We use our own implementation of the Segment Tree[11], implemented by following as strictly as possible the original description of the data structure from [30]. Due to the nature of the data structure, it may report the same results several times during query execution (the higher the interval overlapping, the bigger this overhead).

To minimize any potential bias incurred when retrieving the results, we slightly modified all implementations: Instead of collecting the results, we simply count the number of results in-place.

The machine on which all experiments were run has an Intel(R) Core(TM) i7-2600 processor, 32Gb of main memory, 2Tb of disk space, and Ubuntu 12.04.4 LTS as OS.

5.2 Micro-Benchmarks

5.2.1 Query Execution Analysis

Figure 7 gives the memory consumption of our approach (in red) as well as the query execution time (in black) for our synthetic dataset. Query execution is broken down into several phases: 1) binary search to find the last interval starting inside the query interval, 2) phase 1, plus backward swipe until the first interval starting before query is met $(1-2)$, 3) phases 1-2, plus backward swipe until a checkpoint is met, and finally 4) phases 1-3, plus walk through the checkpoint array until an interval that ends before the query is met.

The graph shows that the memory overhead grows fast when picking low checkpoint values; It is not surprising, since the number of checkpoint arrays is inversely proportional to the value of the checkpoint step.

The optimization algorithm 3 finds 11 as an optimal value of the checkpoint step for the space factor of 1.

11. https://github.com/mavlyutovrus/segment_tree

The performance trends show that query execution time is nearly constant as long as the checkpoint step is sufficiently small, and grows linearly otherwise. We now briefly analyze the impact of each part of the algorithm separately.

The binary search is a time-consuming operation but bears a nearly constant cost for a given dataset. Its cost grows logarithmically with the size of the main array.

The time delta between phase 2 and phase 1 (i.e., the time taken to search back for the first interval starting before the query) stays nearly constant, as the algorithm buffer-reads inside the queries' bounds. The time taken to search back for a checkpoint (i.e., the time between phase 3 and phase 2) grows linearly with the size of the checkpoint step. Phase 3 and 2 are both cache-efficient, since they consist of sweeping through collocated memory regions.

Finally, the time delta taken for analyzing the checkpoint array goes down with the checkpoint step initially, and then stays constant. The reason is that the amount of records we read from the checkpoint array is equal to the number of intervals that overlap the checkpoint and the query intervals. The bigger the checkpoint step, the lower the number of intervals in the checkpoint array that overlap both. As an illustration of this, the impact of the checkpoint array on the total number of intervals in a query response is less than 1% when the checkpoint step is equal to 100 (50.5%, when $ChS = 1$). In that case, almost all intervals that start before the left border of the query and overlap it are collected during the $2-3$ walk. The constant time overhead corresponds to cases when only one record in the checkpoint array is probed. This is due to high price of the first read out of the cache.

To conclude this first experiment, we give below an intuition on why query execution stays nearly constant when the checkpoint step is small. The number of intervals that start before a query and overlap with it is approximately equal to the average overlapping value if the left bounds of the intervals follow a uniform distribution. Those intervals are examined either during the walk to the checkpoint or in the checkpoint array. When the checkpoint step is small, the number of intervals that do not overlap the query but reside between the checkpoint and the first interval starting inside the query is negligible. Consequently, the time for passing through the records standing before the query interval and walking inside the checkpoint array stays almost constant.

We fix the space factor to 1 for the further experiments.

5.2.2 Influence of dataset size on query performance

Figure 8 shows the influence of dataset size on query performance. For that experiment, we increase the range of the intervals left borders with the size of the dataset to keep the average overlapping equal to 10 in all datasets. Results for the Segment tree are omitted, since they are considerably higher than for other data structures (for the 10M dataset it is approximatively 80x slower than our solution).

First, we observe that the performance of CINTIA degrades only slowly, i.e., the query execution time grows logarithmically with the size of the data. As analyzed above, it is mainly due to the time spent on binary search.

Second, we see that CINTIA significantly outperforms other data structures. This can be explained by the low level of locality and by the high number of out-of-cash seeks for the trees and Nested Containment List. We can expect even worse trends for those structures in distributed settings where locality is crucial.

CINTIA significantly outperforms other data structures. This can be explained by a lower number of out-of-cache seeks compared to trees and Nested Containment lists. We can expect even worse trends for those structures in distributed settings where locality is crucial.

5.2.3 Influence of query length on performance

Figure 9 shows the influence of query length on performance. We report the average query time per interval in the response instead of the overall query time, since for all data structures the overall trends are almost linear. We observe that for all data structures (except for the Segment tree, which reports intervals several times), the relative query time goes down linearly with the query length (X-axis is logarithmic).

CINTIA performs better than other solutions. The time spent on binary search and processing intervals that start before the query does not depend on the query length and stays nearly constant, while the number of intervals starting inside the query interval is proportional to its length. So, as the number of returned intervals increases, the relative time spent on the former decreases, while the time spent on the latter stays almost constant.

5.2.4 Influence of average overlapping on performance

Figure 10 shows the influence of the average overlapping on performance. For each average overlapping value, we created a distinct dataset with intervals of different average length. The other parameters stayed constant.

First, we observe that CINTIA again outperforms other solutions (except for the case when the average overlapping is equal to 1, for which the results are the same as for the NCList).

We also observe that the overall query execution time does not significantly increase. In fact, the reason for the slower execution time is the growing number of intervals starting before a query interval and the resulting higher number of returned intervals. Higher overlapping values have however a much stronger influence on the other data structures.

5.2.5 Influence of overlapping standard deviation on performance

Figure 11 shows the influence of overlapping variability to performance. To create this experiment, we generated datasets with uniform distribution of interval lengths. The range of the lengths is different for every dataset. The higher the range, the bigger the overlapping variability in the dataset.

Since the Interval Index performance is dependent on the average overlapping of the dataset, some might expect performance issues for highly variable interval lengths and overlapping profiles. However, our experimental results confirm that this is not the case as overlapping variations have no significant effect on the algorithm performance.

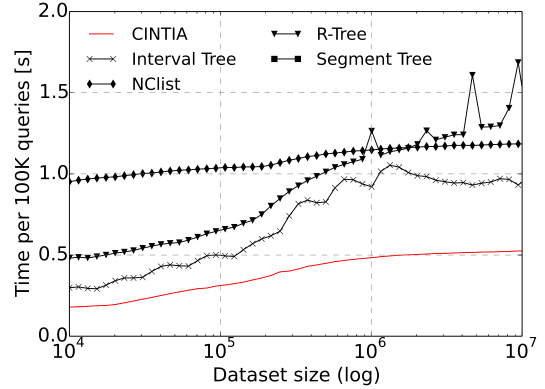

Fig. 8: Query time as a function of dataset size

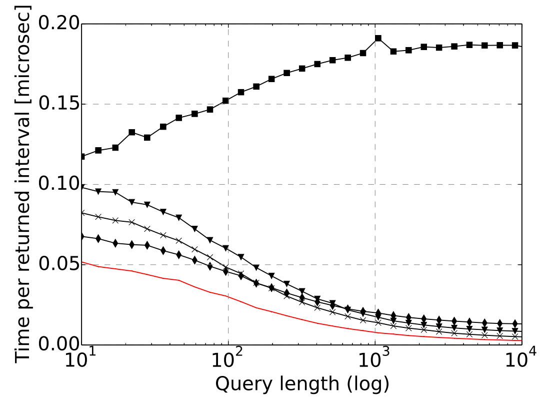

Fig. 9: Query time as a function of query length

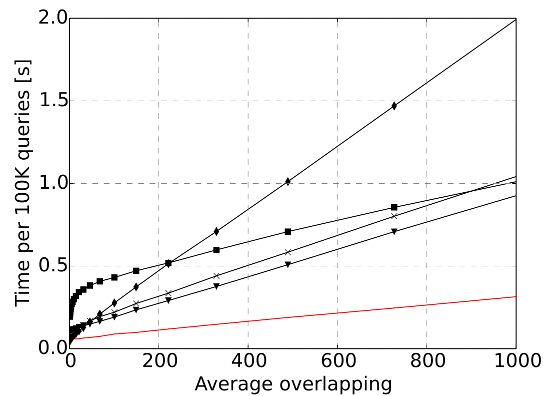

Fig. 10: Query time as a function of average overlapping

5.3 Experiments on Real Datasets

The results on real datasets, summarized in Table 1, follow similar trends. We report 95% confidence intervals in addition to the average results. For the exome dataset, the margin of error was however negligible.

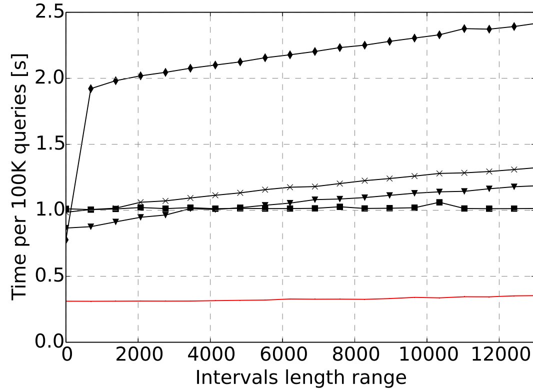

Fig. 11: Query time as a function of variability of interval overlapping

Data structure	Total query time [s]	
	Exome dataset, 11M intervals, avg. overlp=89, 10.5K queries	Jstack dataset, 1.3M intervals, avg. overlp=5611, 100K queries
CINTIA	0.02	1.68 ± 0.01
Interval Tree	0.03	3.14 ± 0.06
NClist	0.02	4.14 ± 0.05
R-Tree 8	0.06	14.15 ± 0.27
R-Tree 16	0.05	7.01 ± 0.13
R-Tree 32	0.03	4.67 ± 0.06
R-Tree 64	0.03	3.94 ± 0.04
Segment Tree	0.67	4.15 ± 0.09

TABLE 1: In-memory benchmarking on real datasets

We report different results for the R-tree to show how the value for maximum children per node dramatically alters its performance.

CINTIA outperforms other solutions in all cases.

5.4 Distributed Setting

As described above, CINTIA outperforms former interval solutions thanks to its high cache locality and the limited number of seeks (or cache line evictions) required when answering interval queries. Those qualities bode well in distributed settings, where "seeks" can be orders of magnitude more expensive than in centralized settings and might require contacting another machine. We experimentally validate this point below.

There exist unfortunately very few distributed solutions to handle interval data. To the best of our knowledge, no distributed implementation of the Interval Tree or related tree structures is available. We compare our solution to Spatial Hadoop 2.2.[12]—which is a popular extension of Hadoop designed to handle big spatial data—on one hand, and to a dedicated solution running on top of MapReduce on the other hand. Other cloud solutions described in Section 2 were either never implemented, or were not available (as we could not get access to the source code of the CNCList+ [25]).

Spatial Hadoop implements spatial indices and distributed query processing. The interval data is in that case distributed among the machines as serialized R+-trees. During query execution, the query is distributed to all machines in the cluster, which then upload and query the R+-trees in parallel. Spatial Hadoop also include a pre-selection step, which only selects those trees whose minimal bounding box overlap the query. After that all results should be uploaded to a client machine.

12. http://spatialhadoop.cs.umn.edu/

Dataset description: - size, - average overlapping	Construction time [ms]	
	Sp. Hadoop	CINTIA
10M, 10.0	285 101	90 154
10M, 100.0	292 578	87 054
10M, 10000.0	282 961	91 532
100M, 10.0	390 526	416 545
100M, 100.0	388 770	438 440
100M, 10000.0	385 126	533 213
1000M, 100.0	1 179 601	5 756 181
Exome alignement dataset	385 051	977 983

TABLE 2: Distributed benchmarks & construction time

To consider another baseline and compare Spatial Hadoop over a non-indexed solution running in the Hadoop environment, we implemented a MapReduce solution on our own. This approach distributes interval data over HDFS and then runs a dedicated MapReduce job, which scans all intervals in parallel and reports those that overlap the query.

The two proposed baselines benefit from parallel processing, but suffer from the high costs of launching a parallel job and then collecting results. Our solution works differently, minimizing the computation performed by the cluster and leveraging dedicated indices in the file system to retrieve the results directly. This allows CINTIA to execute queries on large datasets distributed over a cluster of machines in milliseconds only.

We ran our distributed experiments on a Hadoop v2.3 cluster of 10 machines. The machines have an Intel(R) Core(TM) i7-2600 processor, 32Gb of main memory, 2Tb of disk space, Ubuntu 12.04.4 LTS as OS, and are interconnected through a gigabit ethernet switch.

For the distributed experiments, we generate interval datasets of various size (from 10 millions to 1 billion), average overlapping (10, 100, 10'000) and queried them using sets of queries with different lengths (100 and 10'000).

We also built a real dataset for the distributed experiments, namely the exome alignments of 23 human chromosomes, which constitutes 222M intervals with an average overlapping of 1649. Since the baseline methods are very slow (around 20 seconds per query) we were not able to compare query execution time on the full set of the exome targets. For that reason we sampled from the exome targets a set of 30 intervals with length 120 (which is the mode of the intervals length, as more than 30% of exome targets have that length) and measured the average time to query a target.

Table 2 shows the index construction time for Spatial Hadoop and CINTIA. The results show that the dataset size has a stronger impact on CINTIA's index construction than on Spatial Hadoop. This can be explained by the fact that CINTIA first sorts all intervals. Spatial Hadoop, on

the other hand, does not sort intervals but assign them to a certain mapper according to the position of its minimal bounding box. Though computationally more efficient, this can lead to severe skews in the distribution of the intervals depending on the positions of their bounding rectangles.

Table 3 gives query performance results. We repeated each experiment 30 times for both Spatial Hadoop and MapReduce, and 100 times for CINTIA.

For CINTIA we report results both on cold and hot data (after 100K queries), since warming-up the cluster noticeably affects the query performance. It is not the case for SpatialHadoop and MapReduce, however.

An important factor that affects the performance of CINTIA is the distribution of blocks among the cluster. If there is a "collector" machine which accumulates a high number of index replicas, it will serve more read queries from the client, which leads to an inefficient use of cache on other machines. Hence for the Interval Index, the best case is when replicas are well-shuffled among cluster machines. Our current implementation creates that kind of collector on a machine where the index constructor resides. To reduce the unbalancing impact in our tests, we do not allow a client application to read from replicas on that machine.

We observe a tremendous difference in query performance: CINTIA is more than three orders of magnitude faster than our two baselines.

The baseline approaches are only marginally affected by the average overlapping and by the size of the dataset, though the biggest dataset (1B intervals) makes them both significantly slower. The query execution time of CINTIA, on the other hand, is directly proportional to the number of reads from the distributed file system (the average number of reads is given in the last column of the table), which is determined by the size of the checkpoint step and by the average overlapping as explained above. Average overlapping impacts the query performance of CINTIA on both warm and cold data. For cold data, the delays are higher as reads from HDFS are more expensive in that case.

Results also show that the dataset size has an effect on the performance of CINTIA. Executing queries on 1B intervals is almost 3 times slower than on 10M intervals on cold data. It is only around 30% more expensive on warm data. The number of reads issued to the file system stays constant in all cases. We assume that the speed difference is hence connected to the higher costs incurred when executing larger reads on HDFS.

Finally, we note that the results on the Exome data are similar to the results on our synthetic dataset with similar size and average overlapping.

6 Conclusions

Intervals have become prominent as they are the main structure to represent a number of key data types today. In this paper, we proposed a new data structure called CINTIA (Checkpoint INTerval Index Array) for storing and querying interval data. To the best of our knowledge, it is the first index designed for running low-latency queries on large-scale interval data. We introduced a series of algorithms to compactly store and efficiently query interval data using our structure and discussed their complexity. We also described two open-source implementations of CINTIA, and empirically showed their superior performance both for in-memory and distributed settings.

CINTIA's first deployment is dedicated to the management of large-scale bioinformatics data[13]. We plan to continue working on our system, in order to improve its construction time as well as the efficiency of its random inserts. In addition, we plan to extend our solution to handle multidimensional data by taking advantage of space-filling curves such a Z-curves of Hilbert curves.

7 Acknowledgments

This work was supported by the Swiss National Science Foundation under grant numbers PP00P2_128459 and 200021_143649.

References

[1] M. H. Böhlen, R. Busatto, and C. S. Jensen, "Point-versus interval-based temporal data models," in *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, 1998, pp. 192–200. [Online]. Available: http://dx.doi.org/10.1109/ICDE.1998.655777

[2] R. Snodgrass, "Temporal databases status and research directions," *ACM Sigmod Record*, vol. 19, no. 4, pp. 83–89, 1990.

[3] J. K. Lawder and P. J. H. King, "Querying multi-dimensional data indexed using the Hilbert space-filling curve," *ACM Sigmod Record*, vol. 30, no. 1, pp. 19–24, 2001.

[4] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer, "Integrating the UB-Tree into a Database System Kernel," in *Proceedings of 26th International Conference on Very Large Data Bases (VLDB)*, A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, Eds. Morgan Kaufmann, 2000, pp. 263–272.

[5] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-Store: A Column Oriented DBMS," in *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2005.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[7] Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras, *Advanced Database Indexing*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[8] T. Bozkaya, "Index Structures For Temporal And Multimedia Databases," PhD thesis, Department of Computer Engineering and Science, Case Western Reserve, University, Tech. Rep., 1998.

[9] Z. M. Kharaji and B. G. Nickerson, "Distributed Spatial Data Structuress," 2014.

[10] H. Edelsbrunner, "Dynamic Data Strcutures for Orthogonal Intersection Queries," TU Graz, Tech. Rep. Tech. Report., 1980.

[11] H.-P. Kriegel, M. Potke, and T. Seidl, "Managing Intervals Efficiently in Object-Relational Databases," in *Proceedings of 26th International Conference on Very Large Data Bases (VLDB)*, 2000, pp. 407–418.

[12] L. Arge and J. S. Vitter, "Optimal external memory interval management," *SIAM Journal on Computing*, vol. 32, no. 6, pp. 1488–1508, 2003.

[13] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. New York, NY, USA: Springer-Verlag New York, Inc., 1985.

[14] E. N. Hanson and M. Chaabouni, "The IBS-tree: A Data Structure for Finding All Intervals That Overlap a Point," Technical Report WSU-CS-90-11, Wright State University, Tech. Rep., 1990.

13. http://3dgb.cs.mcgill.ca/

Dataset description: - size, - average overlapping, - query length	Avg. intervals per query	Query execution time [ms]				CINTIA: avg. reads from HDFS per query
		MapReduce	Spatial Hadoop	CINTIA: cold cluster	CINTIA: warm cluster	
Synthetic datasets						
10M, 10.0, 100.0	10.2	21 958±184	17 589±280	2.06±0.04	1.60±0.03	1.0
10M, 10.0, 10000.0	19.7	24 260±2188	17 978±718	2.25±0.06	1.64±0.03	1.0
10M, 100.0, 100.0	97.3	20 250±257	17 125±36	5.02±0.28	1.75±0.04	1.0
10M, 100.0, 10000.0	106.8	20 248±233	17 132±35	6.11±0.28	1.73±0.05	1.0
10M, 10000.0, 100.0	9977.9	22 024±930	17 662±486	14.32±0.35	11.66±0.07	3.5
10M, 10000.0, 10000.0	9987.4	20 216±249	17 332±240	19.19±1.10	11.91±0.48	3.4
100M, 10.0, 100.0	10.7	26 547±2605	17 951±389	3.56±0.37	2.18±0.23	1.0
100M, 10.0, 10000.0	110.8	23 780±695	17 696±537	4.07±0.39	2.14±0.23	1.1
100M, 100.0, 100.0	101.0	22 755±437	17 082±134	4.21±0.33	2.00±0.10	1.0
100M, 100.0, 10000.0	201.1	23 862±547	17 849±397	4.65±0.41	2.19±0.22	1.1
100M, 10000.0, 100.0	10 051.4	24 135±1104	17 903±732	16.17±0.73	12.13±0.12	3.5
100M, 10000.0, 10000.0	10 151.5	24 984±652	17 801±489	16.96±0.79	12.49±0.39	3.5
1000M, 100.0, 100.0	114.2	77 285±1903	17 293±64	13.30±1.22	2.33±0.11	1.0
1000M, 100.0, 10000.0	1117.6	76 699±1038	18 026±484	17.29±1.25	5.67±0.46	1.5
Exome alignment dataset						
222M, 1649.9, 120.0	483.3	25 087±216	18 146±389	5.13±1.30	2.97±0.31	1.6

TABLE 3: Query execution time for distributed setting

[15] E. Hanson and T. Johnson, "Selection Predicate Indexing for Active Databases Using Interval Skip Lists," *Information Systems*, vol. 21, pp. 269–298, 1996.

[16] A. V. Alekseyenko and C. J. Lee, "Nested Containment List (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases," *Bioinformatics*, vol. 23, no. 11, pp. 1386–1393, 2007.

[17] R. Elmasri, G. T. J. Wuu, and Y.-J. Kim, "The Time Index: An Access Structure for Temporal Data," in *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB)*, ser. VLDB '90. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=645916.671968

[18] C.-H. Ang and K.-P. Tan, "The Interval B-tree," *Inf. Process. Lett.*, vol. 53, no. 2, pp. 85–89, Jan. 1995. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(94)00176-Y

[19] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84. New York, NY, USA: ACM, 1984, pp. 47–57. [Online]. Available: http://doi.acm.org/10.1145/602259.602266

[20] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '90. New York, NY, USA: ACM, 1990, pp. 322–331. [Online]. Available: http://doi.acm.org/10.1145/93597.98741

[21] C. P. Kolovson and M. Stonebraker, "Segment Indexes: Dynamic Indexing Techniques for Multi-dimensional Interval Data," in *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '91. New York, NY, USA: ACM, 1991, pp. 138–147.

[22] R. Fenk, V. Markl, and R. Bayer, "Interval Processing with the UB-Tree," in *International Symposium on Database Engineering and Applications*, 2002.

[23] P. Cudre-Mauroux, E. Wu, and S. Madden, "TrajStore: An adaptive storage system for very large trajectory data sets," *2014 IEEE 30th International Conference on Data Engineering*, vol. 0, pp. 109–120, 2010.

[24] P. Bisadi and B. G. Nickerson, "Orthogonal Range Search using a Distributed Computing Model." in *CCCG*, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/cccg/cccg2011.html#BisadiN11

[25] Z. Wang, K. Gong, S. Jin, W. Li, and Z. Liu, "An Efficient Interval Query Algorithm Based on Inverted List in Cloud Environment," in *International Conference, ICCIP*, 2012.

[26] C. Du Mouza, W. Litwin, and P. Rigaux, "Large-scale indexing of spatial data in distributed repositories: the SD-rtree," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 18, no. 4, pp. 933–958, 2009.

[27] A. Eldawy and M. F. Mokbel, "A demonstration of SpatialHadoop: An efficient MapReduce framework for spatial data," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 6, no. 12, pp. 1230–1233, 2013.

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[29] R. Mavlyutov and P. Cudre-Mauroux, "Technical report: A distributed, low-latency index to efficiently manage big interval data," http://exascale.info/papers/cintia_report.pdf, accessed: 2015-03-18.

[30] J. L. Bentley, "Solutions to Klee's rectangle problems," Technical report, Carnegie-Mellon Univ., Pittsburgh, PA, Tech. Rep., 1977.

Ruslan Mavlyutov Ruslan Mavlyutov is a Ph.D. student working with the eXascale Infolab at the University of Fribourg in Switzerland. He previously worked at Yandex and at Microsoft Research Silicon Valley. He received his Master of Science in Computer Science from Moscow Institute of Physics and Technology. He research interests are in scalable, low-latency data structures for the cloud. Webpage: https://exascale.info/members/ruslan-mavlyutov/

Philippe Cudré-Mauroux Philippe Cudre-Mauroux is a Full Professor and the Director of the eXascale Infolab at the University of Fribourg in Switzerland. He received his Ph.D. from the Swiss Federal Institute of Technology EPFL, where he won both the Doctorate Award and the EPFL Press Mention in 2007. Before joining the University of Fribourg, he worked on information management infrastructures at IBM Watson (NY), Microsoft Research Asia, and MIT. He recently won the Verisign Internet Infrastructures Award, a Swiss National Center in Research award, a Google Faculty Research Award, as well as a 2 million Euro ERC grant. His research interests are in next-generation, Big Data management infrastructures for non-relational data. Webpage: http://exascale.info/phil