# Kangaroo: Workload-Aware Processing of Range Data and Range Queries in Hadoop*

Ahmed M. Aly†
Google Inc.
Mountain View, CA, USA
aaly@google.com

Hazem Elmeleegy
Turn Inc.
Redwood City, CA, USA
hazem.elmeleegy@turn.com

Yan Qi
Turn Inc.
Redwood City, CA, USA
yan.qi@turn.com

Walid Aref
Purdue University
West Lafayette, IN, USA
aref@cs.purdue.edu

## ABSTRACT

Despite the importance and widespread use of range data, e.g., time intervals, spatial ranges, etc., little attention has been devoted to study the processing and querying of range data in the context of *big data*. The main challenge relies in the nature of the traditional index structures e.g., B-Tree and R-Tree, being *centralized* by nature, and hence are almost crippled when deployed in a distributed environment. To address this challenge, this paper presents *Kangaroo*, a system built on top of Hadoop to optimize the execution of range queries over range data. The main idea behind Kangaroo is to split the data into non-overlapping partitions in a way that minimizes the query execution time. *Kangaroo* is query workload-aware, i.e., results in partitioning layouts that minimize the query processing time of given query patterns. In this paper, we study the design challenges Kangaroo addresses in order to be deployed on top of a distributed file system, i.e., HDFS. We also study four different partitioning schemes that Kangaroo can support. With extensive experiments using real range data of more than one billion records and real query workload of more than 30,000 queries, we show that the partitioning schemes of Kangaroo can significantly reduce the I/O of range queries on range data.

## 1. INTRODUCTION

Hadoop has become a standard platform for big data analytics. Many businesses are increasingly becoming dependent on it, and this is especially true for the Internet companies. Moreover, the tools built on top of Hadoop such as Pig [25], Hive [28], Cheetah [6], etc make it easier for users to engage with Hadoop and run queries using friendly high-level languages.

One of the main issues with Hadoop is that executing a query usually involves scanning very large amounts of data to perform filtering, grouping, and aggregation, which can lead to high response times. Most prior work attempting to reduce the amount of data scanned during query processing has been considering either introducing traditional database indexing at the record-level or on filtering out entire partitions in case the query has some filtering criteria defined on the partition key. Moreover, except for very little work, such as [11], not enough attention has been given to addressing those concerns in the context of big *range* data, as opposed to the regular point data records.

To appreciate the significance of range data types and how they are central to many large-scale data-intensive industries, consider the following few examples:

- **Digital Advertising:** This is the main driving application for this work, since Turn is a digital advertising company. The key value for this type of companies lies in their ability to accurately and efficiently deliver the advertiser's message to their target audience in the online world, which is simply known as executing their ad campaigns. For this purpose, Turn maintains anonymized profiles (based on cookie ids) for online users., which will help in deciding whether a given user is a good match for a certain advertiser. Obviously, running such campaigns results in an enormous amount of data generation. This data is later analyzed to try to glean insights related to how different users and user segments respond to the different marketing stimuli. Hence, in this context, we can consider each user profile as a range data record, whose range is defined by its lifetime; i.e. from the time the profile was created until the time of its last activity. Similarly, queries typically have time ranges; e.g., the duration of the campaign being analyzed.

  The need for this type of analysis also extends to other online players like big publishers – e.g., popular social networks or mobile apps – where they need to better understand the way their users interact with their websites or apps either for advertising purposes or for improving the user experience. In this case, they may want to track individual user sessions, which can also be regarded as range data records.

- **Television:** The television industry has some similarities to the online world. Specifically, the set-top boxes installed in homes by the TV service provider keep track of the viewed channels (and for how long). All those range data records are sent back to the service provider to be normally used for warehousing and analytics. Some of the common queries against this type of data is to try to get a break down of the viewership for a certain TV program or TV ad. Clearly, those queries are range queries.

- **Telecom:** Calls made via cell phones can normally span multiple locations, i.e., multiple cell towers. Wireless telecom companies need to keep track of those call records, which also have associated time ranges, for billing purposes as well as

*This research was supported in part by National Science Foundation under Grant IIS 1117766.

†All of the research conducted by the first author was done while being an intern at Turn Inc., and a PhD student at Purdue University.



*WSDM'16, February 22–25, 2016, San Francisco, CA, USA.*

 ISBN 978-1-4503-3716-8/16/02...$15.00

DOI: http://dx.doi.org/10.1145/2835776.2835841

for load-monitoring and capacity planning. Examples of analysis queries on this type of data can include a query returning a breakdown of call traffic by cell tower and customer demographics within a given period of time.

- **Customer Support:** Customer support tickets normally take some time before they are closed, and they are usually the subject of a lot of analysis – for example to understand the average response types to customers in certain classes of issues or by certain call agents. Also, those analyses would typically be limited to a certain time period.

In this paper, we present *Kangaroo*, a system built on top of Hadoop and designed especially to handle range data and range queries. Its key strength is centered around its ability to effectively partition the range data in such a way that maximizes the amount of data that can be skipped due to the irrelevance to the query. In other words, queries can "jump" over many irrelevant partitions, and hence the system name. The partitioning approach used by Kangaroo also has many desirable and novel features: it (1) is workload-aware, (2) allows no data duplication across partitions, (3) allows no data splitting across partitions, (4) allows no partition overlap, (5) guarantees bounded partition sizes, and (6) guarantees bounded number of partitions.

Our extensive experimental study on real data of more than one billion records and real query workload of more than 30,000 queries clearly demonstrates the effectiveness of Kangaroo.

The rest of the paper is organized as follows. Section 2 defines the problem, and describes and motivates the key requirements in a Hadoop-based system for processing range data and range queries. Section 3 overviews Kangaroo. Sections 4 and 5 explain the different algorithms employed by Kangaroo to perform the data partitioning. The experiments and results are given in Section 6, and the related work is presented in Section 7. Section 8 concludes the paper.

## 2. PROBLEM DEFINITION

We consider a set, $R$, of $N_R$ records. Each record, $r$, is defined by a record id, $r.id$, some range data, say $r.start$ and $r.end$, and any other arbitrary attributes. A workload, $Q$, of $N_Q$ queries is imposed on $R$. Each query, $q$, is also defined by a range, $q.start$ and $q.end$, such that a record, $r$, is only relevant to $q$ if their ranges overlap. Also, every query, $q$, has a submission time, $q.submit\_time$. Our goal is to partition $R$ into a given number of partitions such that the amount of data scanned by $Q$ is minimized, and hence the cost of executing $Q$ is minimized as well.

There are six main properties we wish to have in our partitioning scheme. We list and motivate them below.

- **Workload Awareness:** Unlike traditional spatial indexing methods, which effectively partition data into multiple buckets and are mainly concerned with the data distribution irrespective of the query workload, we found that taking the workload into account can be very helpful. In particular, regions of data that are queried with high frequency need to be more aggressively partitioned compared to the other less popular regions. This fine-grained partitioning of the in-high-demand data can result in significant savings in query processing time.

- **No Data Duplication:** Some existing spatial indexing methods, particularly space-partitioning trees, can cause data records to be duplicated in multiple partitions if those records happen to cross partition boundaries. This duplication is not desirable, especially in the context of very large datasets, because of the potentially large space overhead and the additional processing needed at query time to de-duplicate the results.

- **No Data Splitting:** It is common in big data analytics that records have a nested structure, such as online user profiles for instance, where each profile contains the user's different online events that together define her interval of online activity. In this case, one possible solution to avoid data duplication across multiple partitions is to instead *split* those nested records across the partitions. So for the user profiles example, if two partitions cover two different time ranges, and the same user spans both ranges, then the profile can be split into two parts, such that each partition will only contain the part with events occurring within the partition's time range. Unfortunately, this solution is problematic – because whenever we need to analyze the complete profiles, we will have to perform a very expensive join across the different parts of each profile. In fact, this defeats the whole purpose of building user profiles, which is supposed to group the data that would normally to be analyzed together ahead of time, so that joins can be avoided.

- **No Partition Overlap:** Another alternative offered by other types of spatial indexes, such as the R-tree and its variants, is to ensure that each data record belongs to one and only one partition, which eliminates the need for data duplication or splitting. However, this comes at the expense of allowing partitions to *overlap*. The problem with this approach is that when a query's range touches a region that belongs to the overlap of multiple partitions, then all those partitions will have to be scanned, and thereby substantially increasing the overhead.

- **Bounded Partition Size:** It is important for a partitioning scheme, especially when it operates in the Hadoop environment, to provide a means of control on the amount of skew in partition sizes. The key reason is that we want to avoid big skews across the reduce tasks of the map-reduce job performing data partitioning, as each output partition will be handled by one of those reducers. For this purpose, we define two configuration parameters, $min\_partition\_size$ and $max\_partition\_size$, that the user can set to control the lower and upper bounds of the partition sizes respectively.

- **Bounded Number of Partitions:** Another important feature of a partitioning scheme running on Hadoop is the ability to control the number of output partitions. Allowing too many small partitions can be very harmful to the overall health of the Hadoop cluster (See [22, 18, 29]). In particular, the name node in charge of the Hadoop File System (HDFS) keeps track of the individual *blocks* comprising the files stored on HDFS. This is a central shared resource in the cluster, so when it gets overloaded, it will slow down the whole cluster. Therefore, it is a common practice in Hadoop to use large block sizes – in the order of hundreds of MBs – and also to avoid files that are much smaller than a single block size.

  To this end, our partitioning scheme has the parameter, $K_t$, representing the *target* number of partitions. While $K_t$ is used to guide the partitioning algorithm, the *actual* number of partitions, $K_a$, may end up being different from $K_t$. The reason for this discrepancy is that the setting of $min\_partition\_size$ and $max\_partition\_size$ will normally impact the final number of partitions. However, we still require $K_a$ to be bounded w.r.t. $K_t$, precisely: $K_t \leq K_a \leq 2 \times K_t$.

## 3. THE KANGAROO SYSTEM

The Kangaroo system is built to address the above challenges. It can efficiently process Web-scale range data given a workload of range queries in a Hadoop environment. As described in Section 2, it takes the historical query workload into account, ensures that data is neither duplicated nor split, that partitions do not overlap, and that the number and size of partitions are both bounded.

Note that point data and point queries are special cases of range data and range queries respectively, where each point can be thought of as a range whose start and end are identical. So they can be easily handled by Kangaroo.

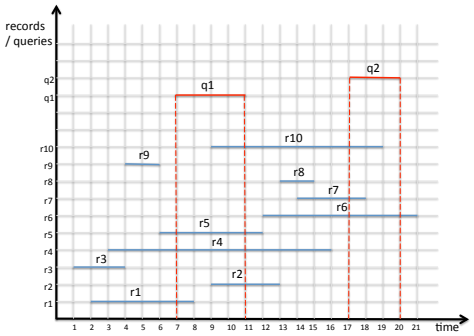



**Figure 1: Sample range data and range queries in their raw format (1D)**

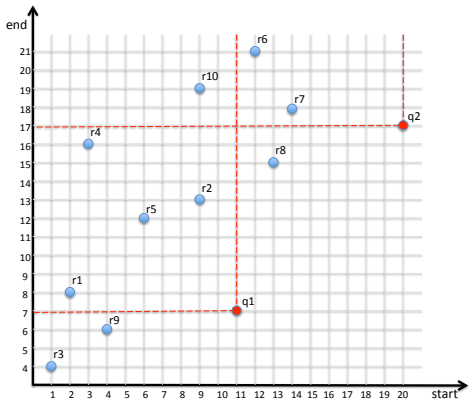



**Figure 2: Sample range data and range queries in their transformed format (2D)**

## 3.1 Main Approach

**Space Transformation:** The main idea behind Kangaroo is that it begins by transforming the problem into a different space where the requirements of no duplication, no splitting, and no overlap can be easily met. Consider Figures 1 and 2. They both represent a scenario where we have 10 different range data records, $r_1 \dots r_{10}$, and two range queries, $q_1$ and $q_2$. The ranges for both data records and queries are considered to be time intervals in this scenario. In Figure 1, they are plotted in the regular 1D space. It can be observed from the figure that it is impossible in this 1D space to partition this data into two or more non-overlapping partitions, with no data duplication or splitting. There does not exist any single partitioning point that would cleanly separate the data records into two different groups – let alone a bigger number of groups. The only solution is to drop one of our constraints by allowing partitions to overlap or data records to duplicate or split.

However, by transforming the problem into the 2D space, as shown in Figure 2, we can find valid partitioning schemes that satisfy all of our constraints. The transformation is performed as follows. For data records, their start and end values will correspond to the first and second dimensions in the new space respectively. This way, every range data record in the 1D space will now become a data point in the 2D space. On the other hand, queries will be transformed to rectangular ranges. The idea is that any overlapping record in the 1D space should now be a data point falling inside the query's rectangular range. Note that a record's range will only overlap with the query's range when both its start value is less than or equal to the query's end value, and its end value is greater than or equal to the query's start value. Thus, in the 2D space, the width of the query rectangle should span all possible start values from the minimum such value to the query's end value. Similarly, the height of the query rectangle should span all the possible end values from query start value to the maximum possible end value.

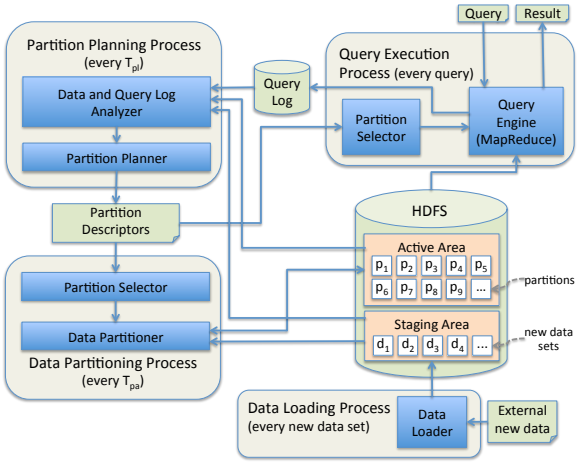



**Figure 3: Architecture of Kangaroo**

For example, in Figure 2, $q_1$'s range is [7,11] in the 1D space. This is transformed into the rectangle whose top left corner is (1, 21) and whose bottom right corner is (11,7). Similarly, $q_2$'s range in the 1D space is [17,20], while its rectangle in the 2D space has (1,21) as its top left corner and (20,17) as its bottom right corner. We can see that in Figure 1, $q_1$ overlaps with $r_1$, $r_2$, $r_5$, and $r_{10}$, whose corresponding data points in Figure 2 are enclosed inside $q_1$'s rectangle. The same relationship applies between $q_2$ and records $r_6$, $r_7$, and $r_{10}$.

It is now clear that all queries will always share the same top left corner, which is the top left corner of the entire 2D space. Therefore, each query can be uniquely identified by its bottom right corner, which is the data point corresponding to its range's end and start values (the boundaries in reverse order).

**Cost Model:** In Kangaroo, we estimate the cost of executing a query by the number of records it has to read. Thus, the main optimization goal of Kangaroo is to partition the data in a way that minimizes the number of retrieved records for a given query workload. We estimate the cost, i.e., quality, of a partitioning layout by the amount of records the queries of the workload will have to retrieve. More formally, given a partitioning layout, say $L$, the cost will be:

$$Cost(L) = \sum_{\forall p \in L} O_q(p) \times C_r(p), \quad (1)$$

where $O_q(p)$ is the number of queries that overlap with Partition $p$, and $C_r$ is the count of records in $p$. In Section 5, we show how the above cost function can be efficiently evaluated for any given partitioning layout. Furthermore, we show how Equation 1 can be adjusted to have more precise estimate of the cost of query execution.

## 3.2 System Architecture

The overall architecture of Kangaroo is given in Figure 3. At the heart of the system is the data store maintained within HDFS. The data store has two main areas: a *staging* area, which only serves as a transient location for incoming new datasets, and an *active* area, into which the new datasets are ultimately loaded and made available to answer user queries.

As shown in Figure 3, Kangaroo has four main processes, each running on its own schedule independently from the other processes. However, they do interact through their inputs and outputs.

**Partition Planning Process:** This is the key process distinguishing Kangaroo from other Hadoop-based data processing systems. The *partition planning* process is responsible for identifying the (near) optimal partitioning scheme (according to the cost model in Equation 1) to be later used for partitioning the data and serving the query load. It

has a *data and query analyzer* component, which aggregates and maps the active and new data records residing in HDFS, along with the historical queries from the query log – onto the 2D space. The data and queries in their new format are then passed to the second component, the *partition planner* to run the partitioning algorithm that would find the best partitioning scheme to minimize the useless data scanning while satisfying all the required constraints. The output partitioning scheme is then written to disk (typically on HDFS) to be available for other Kangaroo processes. This process runs every given time, $T_{pl}$. Depending on the anticipated rate of change in the workload patterns, $T_{pl}$ can be configured to have different values; e.g, weekly, monthly, etc. It can also be dynamically triggered by a continuous monitoring system for the workload pattern changes. In Section 4, we will describe in detail the different algorithms considered for the partition planner component.

**Data Partitioning Process:** This process runs at a higher frequency (every $T_{pa}$) than the partition planning process; e.g., daily. It is responsible for using the partitioning scheme derived by the planning process to actually construct the new partitions out of both the active (previously-partitioned) and the newly arriving (never-partitioned) data. There are two components used during the *data partitioning* process. The first component is the *partition selector*, which first loads the most recent partition descriptors from disk to memory, and then given an input data record, it can find the matching partition descriptor. This component is used by the second component, the *data partitioner*, to properly assign each scanned data record to its new partition. The data partitioner is implemented as a map-reduce job, where the mappers read different chunks of records and then send each record to the appropriate reducer, which will ultimately write the corresponding partition file to HDFS. Data partitioning is treated as an atomic transaction in the sense that all output partition files are first written to a temporary directory, and then only when partitioning is complete, the old active directory is deleted and the temporary directory is renamed to replace it. If any failures occur during partitioning, the temporary directory is deleted and the process can be repeated.

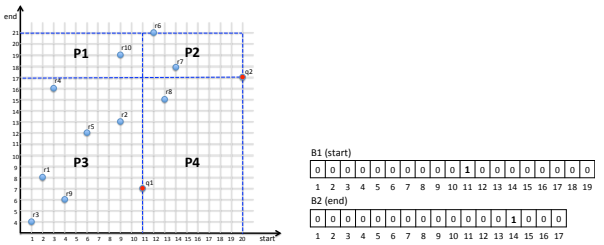
**Query Execution Process:** This process is continuously running as long as queries are being served by the system. It also has two components, where the first component is also an instance of the partition selector. It has a slightly different role in this context, where it is given a query as input and in return it provides all the partitions overlapping with the query range. The second component, which is the *query engine*, uses the partition selector to decide which partitions need to be scanned to answer a given query. The query engine is based on the Cheetah system [6], the data warehousing component used in Turn Data Management Platform (DMP) [12]. In addition to answering incoming queries, the query engine also logs all those queries, so that they can be later used by the partition planning process.

**Data Loading Process:** The *data loading* process is triggered by the arrival of a new dataset. The *data loader* would ingest the new data, perform any necessary sanity checking and transformations, and finally load the data into the staging area in HDFS. The new dataset waits in the staging area until the next run of the data partitioning process. Afterwards, the data partitions are moved to the active area to be ready for query answering. The data loading process is described in more detail in the context of Turn DMP [12].

# 4. PARTITIONING ALGORITHMS

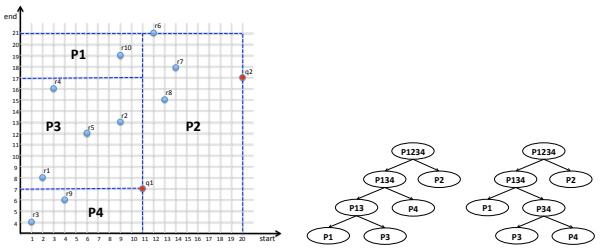
## 4.1 Types of Partitioning Schemes

We consider two types of partitioning schemes that the partitioning algorithms can return: *grid-based* and *tree-based*. An example of grid-based partitioning schemes is given in Figure 4. In this type, the partitions form a coarse-grained grid structure that is overlaid on top of the original fine-grained grid of the 2D space. Hence the partitioning scheme can be represented using a pair of bit strings, $B_1$ and $B_2$, with lengths $(N_1 - 1)$ and $(N_2 - 1)$ for the first and second dimensions respectively. Each bit will denote either a row or a column




(a) Partitons layout (b) Bit strings for the partitions in (a)

**Figure 4: Grid-based partitioning**




(a) Partitons layout (b) Possible tree structures for the partitions in (a)

**Figure 5: Tree-based partitioning**

in the original grid. A set bit indicates that the corresponding row or column acts as a borderline between partitions. For example, in Figure 4, $B_1[11] = 1$ and $B_2[14] = 1$, indicating the two borderlines in the 4-partitions scheme depicted in the figure. (Note that the indexes below the bits in Figure 4(b) denote the positions and not the actual values of the first and second dimensions.)

Figure 5 shows an example of the more general tree-based partitioning scheme. The example also has four partitions, but they do not conform to a grid structure anymore. Instead, they can be represented by any of the two binary trees shown in Figure 5(b). In these trees, the root node represents a large partition covering the entire 2D space. Then, it is split into two smaller partitions captured by its two child nodes. Similarly, the children of each internal node are the outcome of splitting the partition it represents into two partitions, and so on.

Note that with the grid-based partitioning scheme in Figure 4(a), $q_1$ will have to scan 7 records (partitions $P_1$ and $P_3$) even though only 5 are relevant, while $q_2$ will only scan its 3 relevant records (partitions $P_1$ and $P_2$). This gives a total of 10 scanned records for both $q_1$ and $q_2$. On the other hand, when the tree-based partitioning scheme in Figure 5(a) is used, $q_1$ will scan the 5 records it needs, while $q_2$ will scan an extra record beyond it 3 relevant records – resulting in a total of 9 scanned records for both queries, which is a lower cost compared to that of the grid-based partitioning scheme in Figure 4(a).

Because of the large search space involved in both types of partitioning schemes, we considered multiple approximate algorithms to find the (near) optimal solution that minimizes the total cost. These schemes are described in the following subsections.

## 4.2 Grid-Based Algorithms

### *4.2.1 Genetic*

The grid-based genetic algorithm has the same general outline common to all genetic algorithms. It starts by generating an *initial random population* of partitioning schemes. Then, it iterates a given number of iterations. In each iteration, random pairs in the population go through *crossovers* to generate new offspring. A percentage of this offspring will further undergo *mutations*. The new generation consisting of all the offspring following the crossovers and mutations is

then merged with the current population, and only the *most fit* partitioning schemes (the ones with the least costs) will survive until the next iteration, and so on. After all the iterations are over, the minimum cost partitioning scheme in the final population is selected as the answer. This general outline is given in Algorithm 1.

To complete the description of this algorithm, some important details need to be covered, e.g., how the partitioning schemes are represented? how are the random partitioning schemes generated in the initial population? how are the crossovers and mutations performed? and how are the constraints on the partition size and number of partitions enforced?

**Representation:** As described in Section 4.1, grid-based partitioning schemes can be represented using a pair of bit strings, $B_1$ and $B_2$. This is the same representation used by the genetic algorithm.

**Algorithm 1** GridGenetic($R$, $Q$, $N_1$, $N_2$, $K_t$, $min\_partition\_size$, $max\_partition\_size$)

```
1: //generate initial population of partitioning schemes
2: population = { }
3: for i = 1 to population_size do
4:    repeat
5:       ps = GridRandomPS(N1, N2, Kt)
6:    until IsFeasible(ps, min_partition_size)
7:    ps.cost = PSCost(ps, R, Q)
8:    insert ps into population, such that population remains sorted by
      cost.
9: end for
10: for i = 1 to n_iterations do
11:    //generate offspring
12:    offspring = { }
13:    for i = 1 to offspring_size do
14:       repeat
15:          pick two random parent partitioning schemes, ps1 and ps2 from
             population.
16:          ps_child = GridCrossover(ps1, ps2)
17:          rand = a random number between 0 and 1
18:          if rand < mutation_prob then
19:             ps_child = GridMutation(ps_child)
20:          end if
21:       until IsFeasible(ps_child, min_partition_size)
22:       add ps_child to offspring.
23:    end for
24:    //merge offspring with population
25:    for each ps in offspring do
26:       ps.cost = PSCost(ps, R, Q)
27:       insert ps into population, such that population remains sorted by
          cost.
28:       if population.length > population_size then
29:          remove population[population_size + 1].
30:       end if
31:    end for
32: end for
33: ps* = population[1]
34: SplitLargePartititons(ps*, max_partition_size)
35: return ps*.
```

**Random partitioning schemes:** Algorithm 2 shows how we create the two bit strings of a new random partitioning scheme, given the target number of partitions along with the cardinalities of the two dimensions in the 2D space. First, we pick a random pair of values, $K_1$ and $K_2$, such that $K_1 \times K_2 = K_t$. Then $B_1$ and $B_2$ are randomly generated with a total number of bits equal to $N_1$ and $N_2$, of which ($K_1$-1) and ($K_2$-1) are set respectively.

**Crossovers:** The crossover algorithm is given in Algorithm 3. Given a pair of partitioning schemes, we first decide on $K_1$ and $K_2$ for their child. It is chosen out of all valid ($K_1$, $K_2$) pairs, such that it is the closest to the midpoint of the two pairs corresponding to the two parents. We experimentally verified that this strategy is more effective than just randomly picking any valid pair for the child. In the next step, we construct $B_1$ and $B_2$ for the child by first *OR*-ing the $B_1$'s of the parents and the $B_2$'s of the parents respectively. Finally,

**Algorithm 2** GridRandomPS($N_1$, $N_2$, $K_t$)

1: pick a random pair ($K_1$, $K_2$), such that $K_1 \times K_2 = K_t$.
2: construct a random bit string, $B_1$ with ($N_1$-1) bits, such that exactly ($K_1$-1) bits are set.
3: construct a random bit string, $B_2$ with ($N_2$-1) bits, such that exactly ($K_2$-1) bits are set.
4: return the partitioning scheme, $ps$, corresponding to the pair of bit strings, ($B_1$, $B_2$).

a sufficient number of random bits is either set of reset to ensure that $B_1$ and $B_2$ for the child have exactly ($K_1$-1) and ($K_2$-1) set bits.

**Algorithm 3** GridCrossover($ps_1$, $ps_2$)

1: $K_t = ps_1.K_1 \times ps_1.K_2$
2: find a new pair ($K_1$, $K_2$), such that $K_1 \times K_2 = K_t$ and ($K_1$, $K_2$) is the nearest such point to the midpoint of ($ps_1.K_1, ps_1.K_2$) and ($ps_2.K_1, ps_2.K_2$).
3: $B_1 = ps_1.B_1 \vee ps_2.B_1$.
4: randomly set or reset enough bits in $B_1$ until exactly ($K_1$-1) bits are set.
5: $B_2 = ps_1.B_2 \vee ps_2.B_2$.
6: randomly set or reset enough bits in $B_2$ until exactly ($K_2$-1) bits are set.
7: return the partitioning scheme, $ps$, corresponding to the pair of bit strings, ($B_1$, $B_2$).

**Mutations:** As shown in Algorithm 4, the mutation operation is performed simply by swapping a random 0-bit and a random 1-bit in each of $B_1$ and $B_2$ of the input partitioning scheme.

**Algorithm 4** GridMutation($ps$)

1: swap a random 0-bit and a random 1-bit in $ps.B_1$.
2: swap a random 0-bit and a random 1-bit in $ps.B_2$.
3: return $ps$.

**Partition size and count constraints:** Recall that all our algorithms have to conform with the constraint that the size of each partition in the output partitioning scheme should fall between $min\_partition\_size$ and $max\_partition\_size$. Additionally, the actual number of partitions should not exceed double the target number of partitions. The minimum size constraint is enforced using the **IsFeasible()** function (Lines 6, 21 in Algorithm 1), which will only allow a partitioning scheme to be considered if all of its partitions satisfy this constraint.

The maximum size constrain is satisfied through a post-processing step using the **SplitLargePartitions()** function. If the size of a large partition in the final partitioning scheme exceeds $max\_partition\_size$, it gets further partitioned, uniformly, into a number of sub-partitions given by the ratio of its size to $max\_partition\_size$. The partitioning key in this case is the record id, rather than the record range, to ensure that uniform partitioning can always be achieved.

Note that the worst case in terms of partitions imbalance in the output partitioning scheme (prior to splitting large partitions) will happen when all partitions are virtually empty except for one very large partition that almost has all of the data records. In this case, that large partition will be further split a maximum of $K_t$ times, leading to an actual number of partitions, $K_a = 2 \times K_t$. We study the tradeoff in setting $K_t$, $min\_partition\_size$, and $max\_partition\_size$, and their impact on $K_a$ in Section 6.

### 4.2.2 Genetic with Dynamic Programming

We also considered a variant of the grid-based genetic algorithm that uses dynamic programming along one of the dimensions. In particular, it uses the exact same framework captured by Algorithms 1, 2, 3, and 4. However, whenever any of these algorithms attempt to compute the second bit string, $B_2$, for a given partitioning scheme, it is not randomly generated. Instead, it is computed using a dynamic

programming algorithm. The intuition here is that along one dimension, it is possible to compute the optimal 1D partitioning scheme somewhat efficiently and without using any approximate methods.

Algorithm 5 shows how the optimal $B_2$ is computed. Let us first explain the need for dynamic programming. If $B_2$ can only have $K_2$ set bits, then the last set bit can have several valid positions. Going in the backward direction, those valid positions start from the last bit in $B_2$ back to position $K_2$ (where all the previous bits will have to be set too). Out of those alternative positions, to find the one leading to the optimal $B_2$, we first need to solve a smaller subproblem for each one of them. In the subproblem corresponding to position $pos$, we want to find the optimal positioning of ($K_2$-1) 1-bits in the substring of $B_2$ preceding $pos$. Once all the subproblems are solved, the optimal position, $pos^*$, for the last set bit in $B_2$ will be the one with the minimum sum of costs for (1) its corresponding subproblem occurring *before* $pos^*$, and (2) the $K_1$ individual partitions occurring *after* $pos^*$.

Note that those individual partitions are completely defined by $B_1$ along the first dimension and $pos^*$ along the second dimension. For example, in Figure 4, if $B_1$ is as shown and $pos^*$=14, then the individual partitions will be $P_1$ and $P_2$.

The above discussion shows how the problem has an optimal substructure, and hence can be solved using dynamic programming. At a high level, Algorithm 5 builds the dynamic programming matrix, $M$, where one axis represents the cardinality of the second dimension in our 2D space ($i$ varied from 1 to $N_2$), and the second axis represents the target number of partitions we wish to have for that dimension ($j$ varied from 1 to $K_2$). The matrix is built by incrementally computing the costs for all valid combinations across the two axes. During initialization, no partitioning is considered ($j$=1). Then, computing the cost for each subsequent cell in the matrix will be by finding the minimum across multiple sums of costs, as described above. Finally, when the entire matrix is computed, the minimum cost for the whole problem can be found in $M[N_2, K_2]$. The optimal $B_2$ can then be derived by retracing the computation from $M[N_2, K_2]$ back to $M[1, 1]$.

To compute the time complexity for this algorithm, note that there are O($N_2 \times K_2$) valid cels in the matrix. Moreover, the number of smaller subproblems required to compute each cell is O($N_2$). Thus, the total time complexity is O($N_2^2 \times K_2$). Considering that this algorithm can be called a fairly large number of times by the genetic algorithm, it can potentially slow it down compared to the pure genetic approach in Section 4.2.1.

**Algorithm 5** PartitionDP($R$, $Q$, $B_1$, $N_2$, $K_2$)

```
1: //initialization
2: for i = 1 to N_2 do
3:    M[i, 1] = 0
4:    p_list = list of partitions of the form p(x_1, 1, x_2, i), such that (a) x_1=1
      or B_1[x_1 − 1]=1, (b) x_2=N_1 or B_1[x_2]=1, and (c) B_1[x]=0 for x_1 ≤
      x < x_2
5:    for each p in p_list do
6:       M[i, 1] += PCost(p, R, Q)
7:    end for
8: end for
9: //main loop
10: for i = 2 to N_2 do
11:    for j = 2 to min(i, K_2) do
12:       M[i, j] = ∞
13:       for h = j − 1 to i − 1 do
14:          temp_cost = M[h, j − 1]
15:          p_list = list of partitions of the form p(x_1, h + 1, x_2, i), such
             that (a) x_1=1 or B_1[x_1 − 1]=1, (b) x_2=N_1 or B_1[x_2]=1, and (c)
             B_1[x]=0 for x_1 ≤ x < x_2
16:          for each p in p_list do
17:             temp_cost += PCost(p, R, Q)
18:          end for
19:          if temp_cost < M[i, j] then
20:             M[i, j] = temp_cost
21:          end if
22:       end for
23:    end for
24: end for
25: return B_2 resulting in partitioning scheme with minimum cost,
    M[N_2, K_2], by retracing the computation of M[N_2, K_2] back to
    M[1, 1].
```

**Algorithm 6** TreeRandomPS($N_1$, $N_2$, $K_t$)

```
1: construct a partition of the form p(1, 1, N_1, N_2), and make it the root of
   a new partition tree, pt.
2: k = 1 //number of leaf nodes in pt
3: while k < K_t do
4:    SplitRandomLeaf(pt)
5:    k += 1
6: end while
7: return the partitioning scheme, ps, corresponding to pt.
```

## 4.3 Tree-Based Algorithms

### *4.3.1 Genetic*

The tree-based genetic algorithm uses exactly the same outline of Algorithm 1. The main difference, however, is in the way the partitioning schemes are represented, and consequently the related operations of generating new random partitioning schemes, crossovers, and mutations. The mechanisms used to enforce the partition size and count constraints are the same as the ones described for the grid-based algorithms in Section 4.2.1.

**Representation:** As explained in Section 4.1, tree-based partitioning schemes are represented as binary trees, where the leaf nodes capture the final partitions, while the internal nodes capture the parent enclosing partitions.

**Random partitioning schemes:** Algorithm 6 shows how the random partitioning schemes are generated for the tree-based genetic algorithm. It starts by building a single-node tree whose root is the partition covering the entire space. Then, it picks a random leaf node, and randomly splits it into two partitions to form its two children. This process repeats until exactly $K$ leaf partitions are created.

**Crossovers:** The crossover operation is described in Algorithm 7. It is performed by splitting the two input parents using the same random split line. If, for example, the selected split line was vertical, then the right part of the first parent will be trimmed, keeping only the left part to be covered by its tree. Conversely, the second parent's tree will only keep the right part, while the left part will be trimmed. Then, a new root node is created for the child's tree that covers the entire 2D space. Subsequently, the roots of the two trimmed parent trees are added to the child's root as its two children. At this point, the child's tree may contain more or less leaf nodes than $K_t$. To correct this situation, existing leaf nodes are either randomly split or randomly merged with their siblings – until the total number of leaf nodes equals $K_t$.

The functions of trimming a tree given a split line and of merging a leaf node with its sibling are both more complex than the simple function of splitting a leaf node. The **TrimTree()** function operates recursively starting from the root. The root node is first trimmed using the split line. Then, if the split line goes through one of the child nodes, that child is properly trimmed too. If the child node was completely on the *to-keep* side of the split line, it remains unchanged. Otherwise, if it is on the *to-trim* side of the split line, then it is completely removed from the tree along with all of its descendants. For nodes that get partially trimmed, all their children will be examined and processed in the same way.

The **MergeRandomLeaf()** function is also recursive in the sense that it first finds a random leaf node to be merged, and then identifies its sibling. The sibling is precessed recursively as follows. Initially, the sibling node is expanded towards the neighboring to-be-merged node up until the border opposite to their shared border. Then, for each child of the sibling node, if the child shares a border with the to-be-merged node, then it is also expanded in the same direction until the opposite border of the to-be-merged node is reached. Otherwise, if the child does not share a border with the to-be-merged node, then it

remains unchanged. By the end of processing the sibling node and its descendants, the randomly selected leaf node would have been completely merged with its neighbors and the total number of leaf nodes is decreased by one.

**Algorithm 7** TreeCrossover($ps_1$, $ps_2$)

```
1: K_t = number of leaf nodes in ps_1
2: randomly set orientation to either "horizontal" or "vertical".
3: based on orientation, randomly pick a horizontal line, y = s, or a
   vertical line, x = s, where 1 ≤ s ≤ N_2 or 1 ≤ s ≤ N_1 resp.
4: TrimTree(ps_1.pt, s, orientation, "after_trim_line")
5: TrimTree(ps_2.pt, s, orientation, "before_trim_line")
6: construct a partitioning scheme, ps, such that ps.pt.left = ps_1.pt and
   ps.pt.right = ps_2.pt.
7: k = number of leaf nodes in ps.pt
8: while k < K_t do
9:    SplitRandomLeaf(ps.pt)
10:   k += 1
11: end while
12: while k > K_t do
13:   MergeRandomLeaf(ps.pt)
14:   k -= 1
15: end while
16: return ps.
```

**Mutations:** Algorithm 8 illustrates the mutation operation for an input tree-based partitioning scheme. It is as simple as randomly splitting a leaf node in the tree to increase the leaf nodes by one, and then randomly merging a leaf node to bring the total number back to $K_t$.

**Algorithm 8** TreeMutation($ps$)

```
1: SplitRandomLeaf(ps.pt)
2: MergeRandomLeaf(ps.pt)
3: return ps.
```

### 4.3.2 Greedy

The tree-based greedy algorithm, Algorithm 9, is comparable to Algorithm 6, which was used to generate a random tree-based partitioning scheme. The greedy algorithm also starts by creating a root node covering the entire 2D space, and it keeps on splitting leaf nodes until the total number of leaf nodes is $K_t$. However, the main difference is in the way it selects the next leaf node split. It applies a greedy strategy rather than a random strategy. In particular, for each leaf node and for each feasible split for that node, that would not violate the minimum partition size requirement, the leaf node split resulting in the maximum gain in terms of cost reduction is chosen.

We denote the tree generated by the greedy algorithm as the *KNGR-tree* (pronounced "Kangaroo-tree"). This is because, as will be discussed in Section 6, it was shown to have superior properties compared to the other competing algorithms, and as a result, it is the default partitioning algorithm used by Kangaroo.

It is also worth noting that the greedy choice in the KNGR-tree is comparable to the one used for the KD-tree. Some of the key differences however, are that the KD-tree is not aware of the query load, and it normally does not provide any guarantees on the size or count of partitions (data records in leaf nodes).

# 5. EFFICIENT EVALUATION OF THE COST FUNCTION

According to Equation 1 discussed in Section 3.1, the cost corresponding to a partition can be modeled as the number of queries that overlap with the partition multiplied by the number of records inside the partition. This cost model would perform well, i.e., result into partitions of good quality, if all the records have the same size (number of bytes). However, a more precise estimate of the cost corresponding to a partition would consider the size of each record in the cost formula. More formally, the cost of a certain partition, say $p$ is:

**Algorithm 9** TreeGreedy($R$, $Q$, $N_1$, $N_2$, $K_t$, $min\_partition\_size$, $max\_partition\_size$)

```
1: construct a partition of the form p(1, 1, N_1, N_2), and make it the root of
   a new partition tree, pt.
2: k = 1 //number of leaf nodes in pt
3: while k < K_t do
4:    max_gain = 0, max_n = null, max_n_1=null, max_n_2 = null
5:    for each leaf node, n, in pt do
6:       for each possible split of n resulting in two new leaf nodes, n_1 and
         n_2, do
7:          if IsFeasible(n_1, min_partition_size) and IsFeasible(n_2,
            min_partition_size) then
8:             gain = PCost(n, R, Q) - PCost(n_1, R, Q) - PCost(n_2, R, Q)
9:             if gain > max_gain then
10:               max_gain = gain, max_n = n, max_n_1=n_1, max_n_2
                  = n_2
11:            end if
12:         end if
13:      end for
14:   end for
15:   split max_n into max_n_1 and max_n_2.
16:   k += 1
17: end while
18: construct partitioning scheme, ps*, corresponding to pt.
19: SplitLargePartititons(ps*, max_partition_size)
20: return ps*.
```

$$Cost(p) = O_q(p) \times \sum_{\forall r \in p} Size(r), \tag{2}$$

where $O_q(p)$ is the number of queries that overlap with $p$ and $Size(r)$ is the size of record $r$.

A basic operation in each of our proposed algorithms is to compute the cost corresponding to a given partition according to the above equation. For instance, in the tree-based greedy algorithm, we determine for each leaf node the best split that would achieve the maximum gain in terms of cost reduction. In particular, we try all the possible horizontal and vertical splits and for each pair of emerging partitions, we compute the cost. Thus, the computation of the cost of a certain partition has to be extremely efficient because it will have a direct impact on the overall running time of all the partitioning algorithms.

Given a partition, say $p$, the cost corresponding to $p$ contains two components: 1) the size total data size in $p$ and 2) the number of queries overlapping with $p$.

Because the raw data is not partitioned, one can have a complete scan of the data in order to determine the number of points in a given partition (i.e., the first component of the cost of a partition). Obviously, this is not practical to perform. Because we are interested in aggregates, i.e., count of points in a rectangle, scanning the individual points involves redundancy. [17] presents the idea of maintaining *prefix sums* in a grid in order to answer range-sum queries of the number of points in a window, in constant time, irrespective of the size of the window of the query or the size of the data. Hence, we preprocess the raw data in a way that enables quick lookup (in $O(1)$ time) of the count corresponding to a given rectangle. We maintain a two-dimensional grid that has a very fine granularity. The grid does not contain the data points, but rather maintains aggregate information. In particular, we divide the space into a grid, say $G$, of $n$ rows and $m$ columns. Each grid cell, say $G[i,\ j]$, initially contains the total number of points that are inside the boundaries of $G[i,\ j]$. This is achieved through a single MapReduce job that reads the entire data and determines the count for each grid cell. Afterwards, we aggregate the data corresponding to each cell in $G$ using prefix-sums as in [17]. As we explain in [1] in more detail, to compute the number of points corresponding to any given partition, only four values need to be added/subtracted. Thus, in Kangaroo, the process of finding the number of points for any given partition is performed in $O(1)$.

Extending the idea of prefix sums in a grid for counting the number of rectangles (i.e., queries) that overlap a partition is a bit challeng-

ing due to the problem of duplicate counting of rectangles. One way to address this challenge is to insert all the queries into an R-tree, and then given a partition, find the number of overlapping queries from the R-tree. However, this is inefficient because redundancy is involved as we are interested in aggregates. Euler histograms [4] were proposed to efficiently find the number of rectangles that intersect a given region without duplicates. Kangaroo employs a variant of the Euler histogram in [4] to efficiently compute the number of queries that overlap any given partition (without duplicates). Grid $G$ is piggybacked with additional counters that enable Kangaroo to find the number of overlapping queries in $O(1)$. For more details, the reader is referred to [1].

Note that the above techniques of grid-based pre-aggregation are key for the efficiency of our partitioning algorithms. Without pre-aggregation, e.g., using a straightforward approach of successive data scans or R-Tree-based computation, our proposed partitioning algorithms would be impractical (would take hours or even days for high values of $K_t$). With the $O(1)$ counting mechanisms, most of our partitioning algorithms are able to complete execution within one or two seconds as we demonstrate in Section 6.

# 6. EXPERIMENTS AND RESULTS

The Kangaroo system is implemented at Turn Inc to serve as part of its Data Management Platform (DMP) [12]. Turn DMP is a cloud-based service, particularly built for digital marketers. It already has a query execution process similar to the one shown in Figure 3. It translates a SQL-like language (Cheetah Query Language, or CQL) into map-reduce jobs. In this section, we focus on measuring the effectiveness of the partition planning process (in Figure 3), which plays the central role in the effectiveness of the entire system. In other words, we evaluate the partitioning algorithms proposed in Section 4. Note that this process runs periodically on a single node (i.e., it is not distributed).

We have also independently verified the positive impact of highly effective partitioning algorithms on the query speedup, which was expected because of the reduction in the amount of data scanned by each query. It is worth noting, however, that the actual gains in query processing are dependent on the cpu cost for each query, which can widely vary from one workload to another – for example, queries evaluating UDFs can be much more cpu-intensive than simple filtering-and-aggregation queries. For this reason we decided to focus our reported experiments on evaluating the partitioning algorithms.

Our performance measures are: 1) the running time of the algorithms, and 2) the I/O reduction percentage achieved by the partitioning algorithms for a given query-load. To measure the latter, we determine the number of unnecessary I/Os that a no-partitioning scheme would make for a given query-load and compare it to the number of unnecessary I/Os that our proposed algorithms would make for the same query-load. The less the number of unnecessary I/Os our proposed partitioning algorithms would make, the higher the I/O reduction percentage is.

As a baseline, we compare our proposed algorithms to what we refer to as the Random partitioning scheme that for a given $K_t$, uniformly splits the data by equally distant horizontal lines and equally distant vertical lines, i.e., performs uniform grid partitioning, such that the total number of partitions (i.e., grid cells) is $K_t$. We also tried to compare Kangaroo to an existing system, particularly Spatial-Hadoop [11] , which focuses on processing spatial data in Hadoop. Unfortunately, it expected a more recent Hadoop version than the one running on the test cluster we have access to. And since it was a shared cluster, we were not able to upgrade the Hadoop version and compare against SpatialHadoop. As an alternative, we ran an experiment (described in Section 6.3) that shows the effect of workload awareness, which is one of the key differentiators between Kangaroo and SpatialHadoop.

It is important to note that for the genetic algorithms we propose, we examined different values for the parameters: population size, number of offspring, and mutation probability. We preset these parameters to values that achieve the best. Precisely, our best configuration of these parameters is population size = 100, number of offsprings per iteration = 10, and mutation probability = 0.05. It is also worth noting that for all the experiments, to realize exactly $K_t$ by the partitioning algorithms, we set the maximum partition size to $\infty$ and the minimum partition size to 0. These are the default values of these parameters, unless stated otherwise.

In the rest of this section, we study: 1) the effect of the number of partitions ($K_t$) on our performance metrics, 2) the effect of query load awareness on the I/O reduction percentage, 3) the effect of the query selectivity on the I/O reduction percentage, 4) the effect of the scale of the search space on the running time of the partitioning algorithms, and 5) the effect of the max-partition-size parameter on the number of actual partitions, i.e., $K_a$ computed by the algorithms.

In the figures, we refer to the tree-based algorithms as **TB** and the grid-based algorithms as **GB**. For instance, 'Greedy TB' refers to the the tree-based greedy algorithm, while 'Genetic GB' refers to the grid-based genetic algorithm.

## 6.1 Experimental Setup

To evaluate Kangaroo, we use real datasets and real query workload, we have available at Turn. Our data is a sample of over one billion user profiles (each defined by a cookie id). Each user profile contains the user's history of impressions, clicks, and actions. It also has start and end timestamps defining the user's activity lifespan. Our query workload is a sample of 30,000 queries collected from the logs of Turn DMP over a period of six months. Each query specifies a time range of interest, where only users who were active during that time range are to be processed by the query. Since each query has a submission time, we shift the interval of each query in the workload by the number of days since the submission time. The rationale behind this interval adjustment step is that we expect future queries to have a similar behavior to that of historical queries, except that the time ranges they specify will likely be relative to their (future) submission times. Therefore, a good approximation for those future queries that we wish to optimize for is to consider them to be identical to the historical queries after having their time ranges properly shifted.

## 6.2 Effect of the Number of Partitions

In this experiment, we study the effect of the number of partitions ($K_t$) on our performance metrics. At different values of $K_t$ (in log-scale), Figure 6(a) shows the percentage of I/O reduction each of our proposed partitioning algorithms can achieve in addition to that of the random partitioning scheme. As the figure demonstrates, the performance of all the proposed algorithms is better than that of the random partitioning scheme. Furthermore, our proposed algorithms tend to achieve maximal (100%) reduction percentages especially for high values of $K_t$. Notice that the Greedy TB algorithm achieves the best performance, reaching almost 100% reduction for only 512 partitions. Observe that the amount of reduction of the Genetic GB algorithm seems consistently proportional to the value of $K_t$. However, the three algorithms, the curves are not showing consistent proportions of reduction for the different values of $K_t$ because of the fuzziness of the genetic algorithms.

Figure 6(b) shows the running time of our proposed algorithms at different values of $K_t$ (in log-scale). As the figure demonstrates, the performance of the Genetic with Dynamic algorithm is the worst. The reason is due to the high cost of the Dynamic programming evaluation step. For high values of $K_t$, e.g., 2048, the algorithm can take more than a day to execute. However, the running time of the three other algorithms is relatively low (below one second). Interestingly, the Greedy TB algorithm has the lowest running time while achieving the highest I/O reduction percentage.

## 6.3 Effect of Workload Awareness

In this experiment, we study the effect of the workload awareness on the I/O reduction percentage that the partitioning algorithms can

achieve. For simplicity, we choose the Genetic TB algorithm as it achieves the best performance as demonstrated in Section 6.2. In the experiment, we examine two costing models, one that considers both the number of points and the workload in computing the partitions, and the other considers only the number of points. One might view the latter model as a way to generate a two-dimensional k-d tree to partition the data according to the number of points.

Figure 6(c) gives the performance of the two models. It is clear from the figure that the workload-awareness enables the partitioning algorithm to achieve better I/O reduction. Observe that the improvement is bigger with small $K_t$ values because when a small number of partitions is specified, the algorithm has less flexibility and this is where the usefulness of workload-awareness is pronounced the most.

## 6.4 Effect of Query Selectivity

In this experiment, we study the effect of the query selectivity on the I/O reduction percentage that the partitioning algorithms can achieve. We fix the value of $K_t$ to 256. We categorize the queries of the workload according to the query selectivity. Intuitively, the higher the selectivity of the query is, the higher the chance of the partitioning algorithm to achieve high reduction percentages, and vice versa. For instance, when the query selectivity is low, e.g., 0%, i.e., the query has to scan all the data anyway, then, there is no room for reducing the I/O cost, i.e., the expected I/O reduction percentage is 0. In other words, an optimal curve for this experiment when the values of the x-axis are the query selectivities from 0% to 100% should be a straight line with slope = 1.

Figure 6(d) demonstrates the performance of the different partitioning algorithms for different query-selectivity values. It is clear from the figure that the Greedy TB algorithm tends to achieve the best performance, with consistent reduction percentages according to the query-selectivity. Furthermore, it's curve is close to a being a line with slope = 1, i.e., optimal performance as explained above.

## 6.5 Effect of the Scale of Search Space

In this experiment, we study the effect of the scale of the search space on the running time of the partitioning algorithms. Our original search space is defined by a grid of $1200 \times 500$ lines. We fix the value of $K_t$ to 100 and vary the scale of the grid by a scale factor that represents the portion of the grid the should be considered. For instance, if the scale factor is 0.5, it means that we consider half the size of the search space in the partitioning process.

Figure 6(e) demonstrates the performance of the different partitioning algorithms for different scale factors. It is clear from the figure that the running time of the algorithms is directly proportional to the search space size. Observe that the running time of all the algorithms except the Genetic with Dynamic is below one second, which means that they are practical for even higher scale factors.

## 6.6 Effect of Skew in Partition Size

In this experiment, we study the effect of the parameter maximum partition-size on the number of actual partitioning $K_1$. We examine such effect in the Greedy TB algorithm as similar results can be obtained for the other three partitioning strategies. We fix the value of $K_t$ to 100 and vary the value of the maximum partition size as follows. We define the ideal partition size as the total size of the whole space divided by $K_t$. We set the maximum partition-size as multiple of the ideal size.

Figure 6(f) demonstrates the ratio between the actual number of partitions to the target number of partitions, i.e., $\frac{K_a}{K_t}$, for different values of the maximum partition-size. It is clear from the figure that the ratio $\frac{K_a}{K_t}$ always varies between 1 and 2. The worst case happens when the maximum partition size equals the ideal size. In this case, it is possible that the partitioning algorithm keeps fine-tuning a small region of space with $K_t$ splits while leaving the rest of the space unpartitioned. In this case, the unpartitioned region requires at most another $K_t$ additional splits, and hence the overall number of partitions $K_a$ is $2K_t$. Observe that when the maximum partition-size is sufficiently large, the ratio $\frac{K_a}{K_t}$ tends to 1 because in this case, the size of each of the $K_t$ initial partitions is most probably less than the maximum partition-size, and hence no further splitting is needed.

# 7. RELATED WORK

Work related to Kangaroo can be categorized into three main categories: 1) centralized data indexing, 2) data indexing in distributed platforms, and 3) workload-aware query processing.

In centralized indexing, e.g., B-Tree [7], R-Tree [3, 16, 20], Quad-Tree [27], Interval Tree [9], k-d Tree [5], the goal is to split the data in a centralized index that should reside on one machine. Most of the indexes in this category aim at distributing the size of the data among a set of files, where in most cases, there is no restriction on the number of files that can be used. Consequently, in these indexes, the structure of the index can have unbounded decomposed until the finest granularity of data is reached in each split (e.g., file). For instance, the R-Tree recursively keeps splitting the space into rectangles until the number of points in each rectangle is less/greater than a certain threshold. This model of unbounded decomposition plays well for any query workload distribution; the very fine granularity of the splits (e.g., files) enables any query to retrieve its required data by scanning the minimal (or close to minimal) amount of data with very little redundancy.

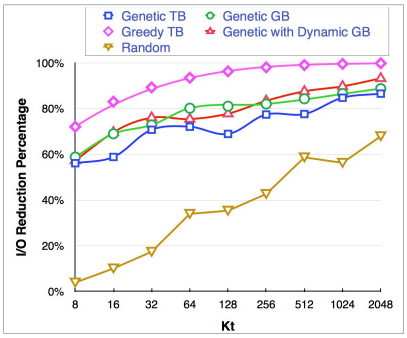
In distributed indexing, e.g., (see [10, 19, 2, 11, 13, 14, 15, 21, 23, 24]), the goal is to split the data in a distributed file system in a way that optimizes the distributed query processing by minimizing the I/O overhead. Unlike the centralized indexes, indexes in this category are usually geared towards fulfilling the requirements of the distributed file system, e.g., keeping the number of input splits (in Hadoop) within a certain bound. For instance the Eagle-Eyed Elephant (E3) framework [13] was proposed to avoid accesses of data splits that are irrelevant to the query at hand. However, E3 considers only one-dimensional point data, and hence is not suited for interval-based data/queries. [11] presents SpatialHadoop; a system that can index spatial two-dimensional data using two-dimensional Grids or R-Trees. A similar effort in [21] addresses how to build R-Tree-like indexes in Hadoop for spatial data. However, none of these efforts is workload-aware.

Several research efforts tried to leverage the skeweness of the query-workload in order to optimize the overall query execution. [1, 8, 26] present query-workload-aware systems and data partitioning mechanisms in distributed platforms. However, these systems cannot support range data and range queries. For instance, in [1], we consider only two-dimensional point (i.e., non-interval) data.

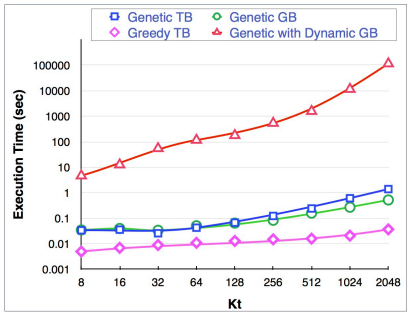
To the best of our knowledge, Kangaroo is the first effort to consider the problem of multi-dimensional range data/query indexing in a distributed system while being query-load aware.

# 8. CONCLUSIONS

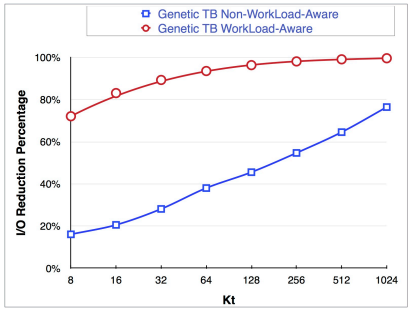
In this paper, we have introduced *Kangaroo*, a system built on top of Hadoop and designed especially to handle range data and range queries. Its key strength is centered around its ability to effectively partition the range data in such a way that maximizes the number of partitions that queries can skip for not having any relevant data. In other words, queries can "jump" over many irrelevant partitions, and hence the system name. The partitioning approach used by Kangaroo also has many desirable and novel features: it (1) is workload-aware, (2) allows no data duplication across partitions, (3) allows no data splitting across partitions, (4) allows no partition overlap, (5) guarantees bounded partition sizes, and (6) guarantees bounded number of partitions. We also described four different partitioning algorithms to be used by Kangaroo and showed experimentally how one of them, tree-based greedy, outperforms all the others. Our experimental study was based on a real datasets of over one billion data records and 30,000 queries. The study showed clearly the effectiveness of Kangaroo in processing range data and queries in big distributed systems.



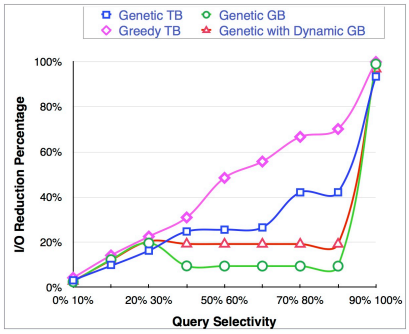

(a) I/O reduction percentage achieved at a given $K_t$.



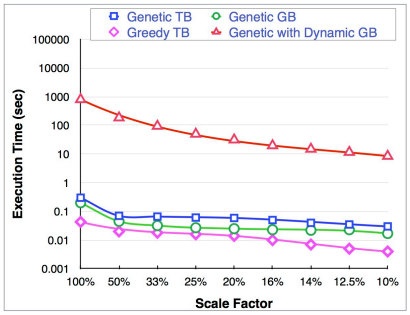

(b) Execution time of the partitioning algorithms at a given $K_t$.



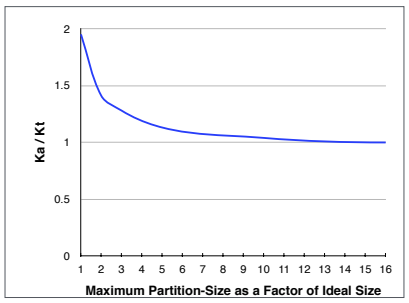

(c) Effect of query-load awareness.




(d) I/O reduction percentage achieved given the query selectivity. $K_t = 256$.




(e) Effect of the scale of the search space on the running time of the partitioning algorithms.




(f) Effect of the maximum partition size parameter on $K_a$. The ratio $\frac{K_a}{K_t}$ is always between 1 and 2.

**Figure 6: Experimental Results**

# 9. REFERENCES


[1] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah. AQWA: adaptive query-workload-aware partitioning of big spatial data. *PVLDB*, 8(13):2062–2073, 2015.

[2] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[4] R. Beigel and E. Tanin. The geometry of browsing. In *LATIN*, pages 331–340, 1998.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[6] S. Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. *PVLDB*, 3(2):1459–1468, 2010.

[7] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[8] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[9] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 978-3-540-77973-5.

[10] J. Dittrich, J. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only aggressive elephants are fast elephants. *PVLDB*, 5(11):1591–1602, 2012.

[11] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*, pages 1352–1363, 2015.

[12] H. Elmeleegy, Y. Li, Y. Qi, P. Wilmot, M. Wu, S. Kolay, A. Dasdan, and S. Chen. Overview of turn data management platform for digital advertising. *PVLDB*, 6(11):1138–1149, 2013.

[13] M. Y. Eltabakh, F. Özcan, Y. Sismanis, P. J. Haas, H. Pirahesh, and J. Vondrák. Eagle-eyed elephant: split-oriented indexing in hadoop. In *EDBT*, pages 89–100, 2013.

[14] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011.

[15] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *PVLDB*, 4(7):419–429, 2011.

[16] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[17] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *SIGMOD*, pages 73–88, 1997.

[18] L. Jiang, B. Li, and M. Song. The optimization of HDFS based on small files. In *IC-BNMT*, pages 912–915, 2010.

[19] A. Jindal, J. Quiané-Ruiz, and S. Madden. CARTILAGE: Adding flexibility to the hadoop skeleton. In *SIGMOD*, pages 1057–1060, 2013.

[20] H.-P. Kriegel, P. Kunath, and M. Renz. R*-tree. In *Encyclopedia of GIS*, pages 987–992. 2008.

[21] H. Liao, J. Han, and J. Fang. Multi-dimensional index on hadoop distributed file system. In *NAS*, pages 240–249, 2010.

[22] G. Mackey, S. Sehrish, and J. Wang. Improving metadata management for small files in HDFS. In *CLUSTER*, pages 1–4, 2009.

[23] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.

[24] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB*, pages 476–487, 1998.

[25] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[26] A. Pavlo, C. Curino, and S. B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

[27] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2006.

[28] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[29] S. Zhang, L. Miao, D. Zhang, and Y. Wang. A strategy to deal with mass small files in HDFS. In *IHMSC*, volume 1, pages 331–334, 2014.