Global Range Encoding for Efficient Partition Elimination*

Jeremy Chen[1], Reza Sherkat[2], Mihnea Andrei[3], Heiko Gerwens[4]
[1]University of Waterloo, [2-4]SAP SE
[1,2]Waterloo, Canada, [3]Paris, France, [4]Walldorf, Germany
[1]y522chen@edu.uwaterloo.ca, [2-4]firstname.lastname@sap.com

ABSTRACT

Skipping mechanisms have been extensively studied to improve query performance over large data volumes. A powerful skipping technique for in-memory columnar databases is partition elimination. The goal is to eliminate, as much as possible, *loading* physically partitioned data into memory and *probing* column partitions against queries. This is achieved by consulting column partition summaries. The summary is often very compact compared to the column partition itself, and is kept in memory, e.g. the MinMax zone map. These summaries have been extensively integrated into modern in-memory database systems including SAP HANA [6]. In this paper, we argue that probing by MinMax range is not efficient when there are gaps in the values that appear in a column partition. Any predicate that needs to probe values in a gap inside a MinMax range naturally ends up requiring a candidate check; this reduces the benefits of column partition pruning. To address this problem, we propose a mechanism to encode each partition (likewise, query) using global ranges, carefully designed to reduce false positive rates. Our approach not only provides a compact in-memory representation, but also supports efficient partition pruning using bitwise operations. Compared to MinMax, our experiments support that our approach significantly reduces the false positive rate. It can allocate memory budget among ranges in partition groups, based on column density, estimated false positive rates from recent workload, and gaps.

1 INTRODUCTION

Partition elimination is a powerful technique to improve query performance over large volume of data [1, 4, 6, 8, 9]. To increase parallelism and achieve operational scalability, physical partitioning is often employed to divide data into independent partitions. This is done to eliminate loading partitions, and probing them against the query, when partitions to access can be inferred explicitly from the query itself. For an in-memory database, this can prevent unnecessary loads of cold partitions (better memory utilization) and can bring significant performance improvement [6]. Small materialized aggregates for partitions, e.g. the MinMax synopsis, are small memory footprint objects that are good candidate for partition examination [5, 6]. If the predicate range of a query does not intersect with the partitions's MinMax synopsis, then it is safe to skip the column partition without incurring false negative. Pruning by partition synopsis is effective if:

- it causes no false negative (i.e. synopsis freshness [6]), and
- it minimizes the need for redundant partition examination.

This paper considers the second synopsis requirement.

*This work was done during Jeremy's internship at SAP Labs, Waterloo, Canada.

2 MOTIVATION: THE LIMITATIONS OF MINMAX SYNOPSIS FOR SPARSE DATA

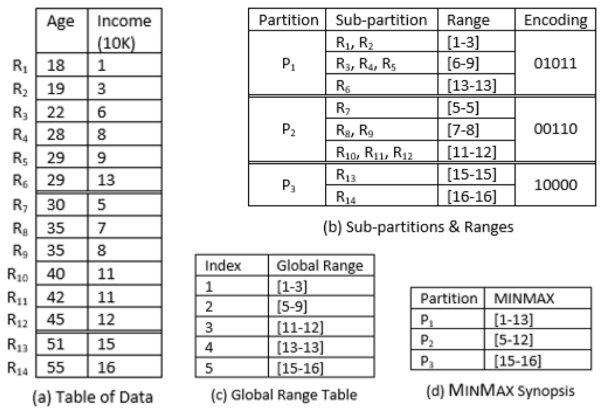
MinMax synopsis has high false positive rate for sparse partitions [7]. If the synopsis of a column partition includes data points within its min-max range that do not actually exist in the column partition, pruning by MinMax can yield false positive. This can happen when the column partition has gaps or values from a sparse distribution. For example, for a table storing the prices of an item over the years, there are usually some gaps in the price column. Although the price is expected to increase monotonically over time, it often does not increase by smallest price unit, and naturally some gaps are present. A natural improvement of the MinMax synopsis is to store several ranges for each column partition. This way, some gaps can be excluded from the column partition representation. However, this approach has two limitations. First, it requires storing a number of value pairs for each partition. This is space-consuming as we usually have thousands of partitions. Second, processing predicates against this extended synopsis becomes very expensive; for each partition it requires comparison against ranges that represent the partition.

3 OVERVIEW OF OUR CONTRIBUTIONS

To address the two limitations stated in Sec. 2, we propose a new list-based structure called the Global Range Table (GRT). This structure helps to construct compact synopses for single column partitions, which supports efficient partition pruning using bitwise operations. Fundamentally, the pruning approach is very similar to the MinMax synopsis [6] and to the Adaptive Range Filters [1] in that we use value ranges to determine whether to access each partition. The key properties of our ranges are:

(1) We construct the list of value ranges that is common across all the partitions. This facilitates probing queries against synopsis using bitwise operations, and
(2) We incorporate recent workload knowledge into our range extraction algorithm. This is motivated by the observation that frequently–accessed values can suffer more from false positives than rarely–accessed values, if the workload characteristics does not change significantly.

We use the extracted GRT to encode column partitions; each bit of the compact encoding indicates whether the corresponding column partition contains some values within the respective GRT range. The GRT ranges can be improved to have small false positive rates, based on the knowledge learned from recent workload and the importance of value ranges. We use the same encoding approach to represent each query as a bit string. This facilitates efficient query probing on partitions. Furthermore, storing only one GRT global to all partitions in memory is much cheaper, compared to storing multiple value ranges per partition. Finally, the amortized space overhead of our approach is one compressible bit string per partition. We propose algorithms to further reduce this overhead, when the memory budget to store partition synopses is limited (Sec. 4.2.2).



	Age	Income (10K)
R_1	18	1
R_2	19	3
R_3	22	6
R_4	28	8
R_5	29	9
R_6	29	13
R_7	30	5
R_8	35	7
R_9	35	8
R_{10}	40	11
R_{11}	42	11
R_{12}	45	12
R_{13}	51	15
R_{14}	55	16

(a) Table of Data

Partition	Sub-partition	Range	Encoding
P_1	R_1, R_2	[1-3]	01011
	R_3, R_4, R_5	[6-9]	
	R_6	[13-13]	
P_2	R_7	[5-5]	00110
	R_8, R_9	[7-8]	
	R_{10}, R_{11}, R_{12}	[11-12]	
P_3	R_{13}	[15-15]	10000
	R_{14}	[16-16]	

(b) Sub-partitions & Ranges

Index	Global Range
1	[1-3]
2	[5-9]
3	[11-12]
4	[13-13]
5	[15-16]

(c) Global Range Table

Partition	MINMAX
P_1	[1-13]
P_2	[5-12]
P_3	[15-16]

(d) MinMax Synopsis

Figure 1: Encoding column partitions using Global Range Table

Table 1: List of Notations

Notation	Description
V	The universe set of column values
$C[v]$	The popularity of column value v (Sec. 4.1)
N_p	The total number of partitions
N_V^p	The total number of distinct values on the column in partition p
k	The maximum number of sub-partitions for each partition
N_R	The number of ranges in rGRT (Sec. 4.2)
m	The desired number of final GRT (i.e. the length of each encoded bit string)

4 DEEP DIVE: GLOBAL RANGE ENCODING

We demonstrate pruning by Global Range Table (GRT) using an example. Table 1 summarizes the notations used in this section.

Example 4.1. *The table shown in Fig. 1a has two columns,* `Age` *and* `Income`*. This table is divided into three partitions based on the* `Age` *attribute (Fig. 1b). The list of ranges in Fig. 1c defines 5 mutually exclusive* `Income` *ranges. Using these* `Income` *ranges, each of the three* `Income` *column partitions in Fig. 1a can be encoded using a bit strings of length 5 in Fig. 1b. For instance, the* `Income` *column for partition P_3 is encoded as* 10000*, This is because this partition only contains values 15 and 16, which fall in the range* [15-16] *of the GRT table in Fig. 1c. As range* [15-16] *has index 5, only the fifth bit is set. Now consider the query* `Income` $\leq$ *12. This query intersects with three ranges from the GRT, namely* [1-3], [5-9]*, and* [11-12]*. Therefore, the query can be encoded as* 00111*. Comparing the bit string of the query with that of the encoding column in Fig. 1c, one can verify that* $00111 \wedge 10000 = 00000$*, hence partition P_3 can be pruned safely. The bit string encoding preserves more gap information, compared with MinMax, e.g. encoding P_1 as* [1-13] *implies that* [11-12] *is included. The bit string excludes this range.*

Conceptually, the global range table is a set of ranges on column partitions. The extraction of these ranges can be performed in either a single-phase process from all column partitions at once, or in a two-phase process by integrating the ranges from partitions. The two-phase approach has several advantages:

- Memory consumption: the single phase approach requires every column partition to be loaded into memory. The alternative way reduces the memory footprint by integrating ranges extracted from independent partitions.
- Performance enhancement: the two-phase approach can be implemented within the MapReduce framework, with the Map phase applied to independent partitions (Sec. 4.1) and the Reduce step to integrate ranges (Sec. 4.2).
- Capturing gaps: ranges extracted from all data might not capture gaps inside partitions. Our two-step approach avoids this by regarding gaps in partitions as constraints, penalized in the goodness measure for ranges (Eq. 3).

4.1 The Sub-Partitioning Step

This step produces a set of ranges from each partition. Once the set of all ranges have been created, they will be integrated to construct GRT (Sec. 4.2). Given a set of values V, integer k, and cost function of Eq. 2, the sub-partitioning problem is to find an optimal selection of non-empty subsets $S_1, \dots, S_k$, s.t. $\cup_{1 \leq i \leq k} S_i = V$. As the order of values is not represented in a set, we re-order the rows in each column partition when extracting ranges and consider subsets with consecutive members. That leads to a simplified problem: given an ordered list of values V, we find the optimal k mutually exclusive sub-lists $V_1 = V[1 : i_1], \dots, V_k = V[1 + i_{k-1} : i_k]$, with $i_{1 \leq j \leq k}$ being indices of sorted value list.

4.1.1 Sub-Partitioning Cost Model. A column value is popular if it can satisfy a large fraction of queries in the recent workload[1]. The more query predicates a value can satisfy, the higher its popularity ranking is. For example, suppose we have two predicates $1 \leq x \leq 5$ and $4 \leq x \leq 10$ on an integer column. Values 4 and 5 can satisfy both predicates while the other values between 1 to 10 satisfy only one predicate each. In this case, 4 and 5 are more popular than 1,2,3,6–10. Let $|Pred|$ denote the total number of predicates in the workload and let V be the set of column values. We define the popularity of a column value $v \in V$ (denoted by $C[v]$) to be the ratio of predicates from the recent workload satisfied by v. $C[v]$ is between 0 and 1, inclusive. A value is popular if it satisfies many predicates. If a popular value is in a gap within a partition, a query for this value returns nothing[2]. Thus, we observe that the probability of false positives depends directly on the popularity of the value(s) in gaps. Let G be a gap and V_G be the set of values included in G. We define the cost of gap for G to be the sum of the popularities of the values it contains:

$$Cost(G) = \sum_{v \in V_G} C[v]. \tag{1}$$

For each sub-partition s, let V_G^s be the set of values of the gaps that are included in s. If a popular value $v \in V_G^s$ (i.e. v does not exist in sub-partition s), then those predicates that v satisfies will cause false positives on the partition to which sub-partition s belongs. The reason is that the sub-partition value range will indicate the existence of v, but it is not true (i.e. false positive). Therefore, the sub-partitioning cost of a partition is the sum of gap costs included in any created sub-partition. Let Sub_p be the set of sub-partitions in the partition p. Then,

$$SubCost(P) = \sum_{s \in Sub_p} \sum_{v \in V_G^s} C[v]. \tag{2}$$

This measure gives the likelihood that the sub-partitioning scheme will create false positives for a partition, for values that do not actually exist in the partition. Thus, creating sub-partition value ranges that include popular gaps yields a high false positive rate. Intuitively, the lower the cost is, the better the sub-partitioning scheme is. When the cost is 0, that means there is no sub-partition that produces false positives on any predicate. Thus, we would like to minimize this measure. To normalize this cost, we divide it by the maximum number of possible distinct values that the partition can take. To summarize, the normalized cost penalizes gaps that cause false positives in a set of ranges extracted from each partition. The ranges are extracted via sub-partitioning.

[1]We base our approach on *predicate stability*, i.e. popular column values continue to satisfy many predicates in future queries. If predicates are not stable, our approach will remain valid but sub-optimal considering the false positive rates.
[2]However, a MinMax synopsis would incur false positive on this gap.

4.1.2 The Largest Gap Greedy Algorithm. Sub-partitions with fewer gaps reduce the cost of the ranges extracted from each partition. Therefore, if we exclude popular gaps from each sub-partition, the cost would be toward optimal. The main idea of our greedy algorithm is to find top $(k-1)$ popular gaps. We use a min-heap to keep track of the top $(k-1)$ popular gaps (i.e. with largest costs). For each node of the min-heap, we keep the gap cost (i.e. Eq. 1; the gap popularity) and the index of the gap. The index of the gap is defined as the identifier of the value immediately before the gap. The min-heap property is maintained with respect to the costs stored in nodes. We go through the sorted list of distinct values of the partition and manage the top $(k-1)$ popular gaps as well as the indices of those gaps in the heap. At the end, the indices determine sub-partition boundaries. At the beginning, there are N_V^p-1 candidate boundary points in partition p with N_V^p distinct values. Our algorithm selects $(k-1)$ positions to minimize the sum of the costs of sub-partitions (i.e. Eq. 2) in $O(N_V^p \log k)$ time. We omit algorithm detail and proofs for brevity.

4.2 Extracting GRT And Optimizations

The sub-partitioning step produces k value ranges per partition. Integrating at most kN_p ranges into a single list may have overlapping ranges, as well as duplicates. In this step, a value-range list global to all partitions is produced, with the primary goal to reduce false positive rates. To achieve this, we extend Eq. 2 to quantify the quality of a global range table GRT, as opposed to one partition:

$$Cost(GRT) = \sum_{r \in GRT} \sum_{p} \Big(C[v] : v \in V_G^p \cap r \text{ s.t. } V_G^p \cap r \neq r \Big). \quad (3)$$

We first integrate kN_p ranges (k ranges for N_p partitions) into one list, and call this rGRT (for raw GRT). We refine rGRT in two steps. First, we remove duplicate ranges and value range overlaps (Sec. 4.2.1). Then, for a given a memory budget, we show how to merge ranges effectively and derive a reduced list (Sec. 4.2.2).

4.2.1 Mutually Exclusive GRT. Starting from rGRT, the goal is to make every pair of ranges mutually exclusive. For this, we propose an approach based on the greedy algorithm proposed for the interval scheduling problem [3], where the range with smallest endpoint from the list is picked and inserted into the result list, if it does not intersect with any other ranges already in the result list. The range is discarded if it is in conflict with at least one range in the result list (i.e. cannot be scheduled together). In our approach, whenever the value range overlaps with some value range already in the result list, we split the to-be-inserted range into smaller ranges instead of discarding it. Each smaller range is either mutually exclusive or a duplicate (i.e. already completely covered) of the ranges in the result list. Then, we discard those duplicates and insert the others. For example, assume that we want to insert value range [2-18] to the list of ranges R={[1-4], [7-10], [15-16]}. In our approach, we split [2-18] into six smaller ranges: [2-4], [5-6], [7-10], [11-14], [15-16], and [17-18]. This is done based on the overlap of [2-18] with the ranges in R. We process each sub-range separately. Because [2-4], [7-10], and [15-16] are already present in the result list, we discard them and insert [5-6], [11-14], and [17-18]. We iterate this split-and-insert process for every range in rGRT and obtain a mutually exclusive global range table (xGRT) in time linear to the number of ranges in rGRT. In this process, the ranges are refined and Eq. 3 reduces, but the number of ranges grows. This increases the number of bits needed to encode each partition.

4.2.2 Greedy Merge Algorithm. A synopsis is expected to have a small memory footprint. We propose a greedy algorithm to merge ranges and reduce the size of xGRT while keeping the GRT cost low. Merging t ranges $R_1, \ldots, R_t$ has two overheads: the extra range cost[3] and the cost of the newly introduced gaps.

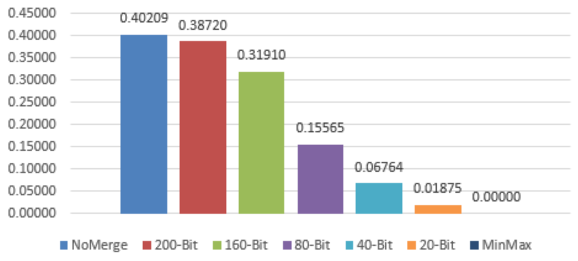
Example 4.2. *Suppose partition P={1, 3, 6, 8, 9, 13} and a given xGRT. Suppose we select two ranges* [8-9] *and* [11-12] *to merge, and let* [11-12] *be a range from* another *partition. The ranges* [8-9] *and* [11-12] *would merge into* [8-12]*. After this merge, the popularities of value 11 and 12 contribute to the GRT cost against partition P while it would not before. This is the extra cost of the range* [11-12]*. Before merging, the query* $11 \le x \le 12$ *would not have any match on this partition since the GRT would tell that this partition has no match. However, after merging, the GRT would suggest to load the partition since this partition has data within* [8-12]*. Because there is a gap between the two ranges (i.e. value 10 is missing), the cost of the gap contributes to the extra cost of merging, which comes from the popularities of 10, 11, and 12.*

The greedy merge algorithm must maintain cost matrix EC, with $EC_{i,j}$ being the extra cost of merging ranges $r_i, \ldots, r_j$ in xGRT. After merging ranges $r_u, \ldots, r_v$, all entries at row v or column v are invalidated, and every entry $EC_{i,j}$ at row u or column u must be recomputed. We continue picking the lowest extra cost and updating EC, until only m ranges remain. However, we notice that if a larger merging fully contains a smaller one, then it is guaranteed that the smaller merging has a lower (or equal) cost than the larger one since the extra-cost is non-negative. Therefore, we simplify the algorithm to use a 1-dimensional list of extra-cost, with EC having N_R-1 entries. In this case, EC_i denotes the extra-cost of merging the i-th range with the next one in xGRT. At each iteration of the algorithm, we pick the lowest extra-cost, EC_u, and merge the u^{th} range with the $(u+1)^{th}$ range and update EC: we invalidate EC_{u+1} and range $(u+1)$ in xGRT. If EC_u is the last entry, then EC_u itself is invalidated. On the other hand, we at most need to update two entries of EC. If EC_u is not the first or the last entry, we re-calculate EC_{u-1} and EC_u using the updated xGRT. Note that EC_u now stores the extra-cost of merging the u^{th} and $(u+2)^{th}$ ranges as the $(u+1)^{th}$ entry has been invalidated. If EC_u is the first (the last) entry, then we only re-calculate EC_u (EC_{u-1}). We iterate until only m ranges remain. We name the result list cGRT (for compact GRT). Our algorithm takes $O\big((kN_p - m)N_p \log(kN_p)\big)$ time to find cGRT.

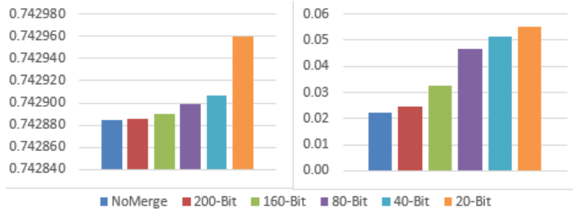
4.3 Partition Encoding and Elimination

Encoding partitions. Once the GRT is constructed with m mutually exclusive ranges, each column partition can be encoded using a bit string of length m. The bit i is 1 if and only if the partition has at least one value within the value range r_i in the GRT. This is a pre-processing step. For each partition, the encoded bit string serves as its compact synopsis, which is kept in memory.
Encoding query. At query processing time, each query is encoded as a bit string. The i-th bit is set to one if at least one value in the range r_i of the GRT satisfies the query predicate.
Partition pruning. The encoded query is compared against the encoded bit string for each partition, using bitwise AND operation. If the result bit string has at least one set bit, there might be some value(s) in the corresponding partition that satisfy the query predicate. Otherwise, no record in the partition will satisfy the query predicate, and the partition can be safely pruned.

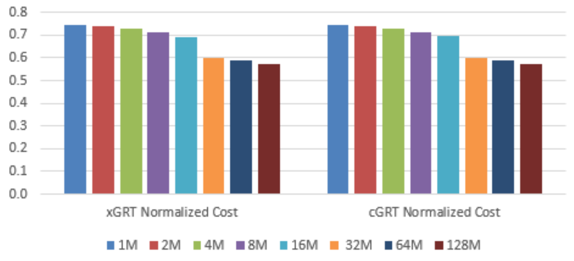
[3]I.e. the range cost of merging $R_1, \ldots, R_t$ but excluding R_i.



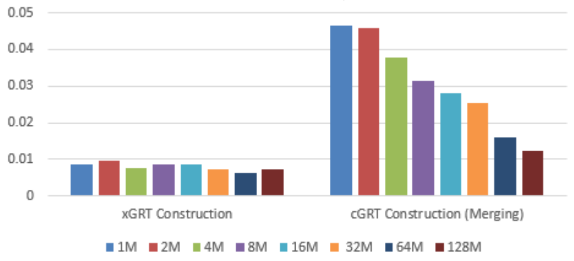
(a) GRT based encoding vs. MinMax (ratio of gaps preserved)



(b) Normalized GRT cost, varying the number of bits (left)
(c) GRT construction time (seconds), varying the number of bits (right)
Figure 2: Experiments on 1M rows



(a) Normalized GRT cost, varying the number or rows



(b) GRT construction time (seconds), varying the number or rows
Figure 3: Scalability

5 EXPERIMENTAL EVALUATIONS

We conducted experiments on a machine with Intel Core i7-4770 CPU. The dataset is from a real SAP application. We range-partitioned the table based on a date column, and extracted synopsis (MinMax or GRT based) from the document identifier (SAP UUID) of each partition. The distribution of values in this column was sparse with many gaps. Fig. 2a compares GRT encoding with MinMax based on the ratio of gaps preserved. For each column partition, the measure quantifies the ratio of values in column partition's MinMax range, that 1) do *not* appear in the column partition, and 2) can be excluded using the column synopsis. This ratio is between 0 and 1, with larger value more desirable; it indicates better pruning. This measure is zero for MinMax; none of the values appearing in a column partition gap can be excluded by the minimum and the maximum range. The memory overhead for MinMax for each partition is a pair of minimum and maximum values. For GRT based encoding, the memory overhead is one bit string per partition, plus one global range table for all partitions. Note that we need to have values (i.e. value pairs for MinMax and value ranges in GRT) in memory, instead of dictionary value identifiers. This is mainly because the pruning is performed without accessing each column and its data structures (i.e. encoded data vector and dictionary).

Using GRT based encoding (Sec. 4.2.2), one can preserve more gaps as "zero" bits of the synopses to demonstrate values in the gap that are not included in a partition. The number of gaps preserved decreases when the number of bits dedicated to each synopsis reduces; merging GRT ranges produce wider the ranges in the target GRT table. This reduces the number of zeros as well as pruning opportunity. Fig. 2b reports the normalized GRT cost for no merge (xGRT of Sec. 4.2.1) and cGRT (Sec. 4.2.2). With reduction in the number of bits, the overhead of merge is observed as increase in the normalized GRT cost and in the construction time (Fig. 2c). For both xGRT and merged GRT, the normalized GRT cost decreases when the dataset size increases (Fig. 3a,b). The reason is that partitions become larger and the number or the size of gaps reduces in each partition. Therefore, the number of split ranges in xGRT decreases. Consequently, the time spent to merge these ranges reduces.

6 CONCLUSION AND FUTURE WORK

We introduced GRT, a compact data structure to achieve partition pruning. GRT is a list of ranges which we use to encode partitions and predicates. Bit string encoding is compact (compared to original partitions) and can be used for efficient partition elimination. This encoding is superior to MinMax zone maps. In particular, when the distribution of values in column partitions is sparse and value gaps appear, pruning by MinMax becomes less effective; false positives demand for extra candidate check [7]. Encoding partitions using global range tables, as opposed to local range tables optimized per partition (e.g. [9]), has the advantage that partition elimination can be performed without re-encoding the query per partition. Our results confirm the effectiveness of our approach, in preserving gap information. We studied the application of GRT in the context of query processing on cold partitions. However, its application can be extended to other use cases, e.g. semi-join reduction and enforcing the uniqueness constraint. Our encoding is reminiscent of Bloom filters [2]; we set bits using a global range table common to all partitions, to assist point and range queries. Designing GRT to offer bounds on false positive rates is an interesting future research direction. We plan to assign different sub-partitioning quota for dense vs. sparse partitions (non-homogeneous sub-partitioning). Sparse partitions should receive more encoding space than dense and uniform partitions, to reduce false positive rates.

REFERENCES

[1] K. Alexiou abd D. Kossmann and P. Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB* 6, 14 (2013), 1714–1725.
[2] B. H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.
[3] J. Kleinberg and E. Tardos. 2005. *Algorithm Design.*
[4] N. Koudas. 2000. Space Efficient Bitmap Indexing. In *CIKM.*
[5] G. Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB.*
[6] A. Nica, R. Sherkat, M. Andrei, X. Chen, M. Heidel, C. Bensberg, and H. Gerwens. 2017. Statisticum: Data Statistics Management in SAP HANA. *PVLDB* 10, 12 (2017), 1658–1669.
[7] D. Rotem, K. Stockinger, and K. Wu. 2005. Optimizing Candidate Check Costs for Bitmap Indices. In *CIKM.*
[8] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. 2014. Fine-grained Partitioning for Aggressive Data Skipping. In *ACM SIGMOD.*
[9] L. Sun, M. J. Franklin, J. Wang, and E. Wu. 2016. Skipping-oriented Partitioning for Columnar Layouts. *PVLDB* 10, 4 (2016), 421–432.