2-28-2022

Faster Multidimensional Data Queries on Infrastructure Monitoring Systems

Yinghua Qin
San Jose State University

Gheorghi Guzun
San Jose State University, gheorghi.guzun@sjsu.edu

Recommended Citation
Yinghua Qin and Gheorghi Guzun. "Faster Multidimensional Data Queries on Infrastructure Monitoring Systems" *Big Data Research* (2022). https://doi.org/10.1016/j.bdr.2021.100288

Big Data Research 27 (2022) 100288

Contents lists available at ScienceDirect

Big Data Research

www.elsevier.com/locate/bdr

Faster Multidimensional Data Queries on Infrastructure Monitoring Systems

Yinghua Qin *, Gheorghi Guzun *

Department of Computer Engineering, San Jose State University, San Jose, CA, USA

ARTICLE INFO

Article history:
Received 15 March 2020
Received in revised form 20 June 2021
Accepted 28 October 2021
Available online 17 November 2021

Keywords:
Bit-sliced index
Bitmap index
Multidimensional data
Preference top-k queries
Performance

ABSTRACT

The analytics in online performance monitoring systems have often been limited due to the query performance of large scale multidimensional data. In this paper, we introduce a faster query approach using the bit-sliced index (BSI). Our study covers multidimensional grouping and preference top-k queries with the BSI, algorithms design, time complexity evaluation, and the query time comparison on a real-time production performance monitoring system. Our research work extended the BSI algorithms to cover attributes filtering and multidimensional grouping. We evaluated the query time with the single attribute, multiple attributes, feature filtering, and multidimensional grouping. To compare with the existing prior arts, we made a benchmarking comparison with the bitmap indexing, sequential scan, and collection streaming grouping. In the result of our experiments with large scale production data, the proposed BSI approach outperforms the existing prior arts: 3 times faster than the bitmap indexing approach on single attribute top-k queries, 10 times faster than the collection stream approach on the multidimensional grouping. While comparing with the baseline sequential scan approach, our proposed algorithm BSI approach outperforms the sequential scan approach with a factor of 10 on multiple attributes queries and a factor of 100 on single attribute queries. In the previous research, we had evaluated the BSI time complexity and space complexity on simulation data with various distributions, this research work further studied, evaluated, and concluded the BSI approach query performance with real production data.

1. Introduction

A preference (top-k) query is a critical function in performance monitoring systems. It enables users to discover top performance issues and take action promptly.

Based on the monitoring targets, the performance monitoring systems could be classified into infrastructure performance monitoring systems, database performance monitoring systems, and application performance monitoring systems. A monitoring system collects data, analyzes the data, and provides insightful information to the system administrators. Streaming in real time with small time-scale granularity, it can be thousands or millions of transactions per second. Performance monitoring data usually contains hundreds of features covering multiple dimensions of the monitoring targets. According to the studies published in [32] and [2], the ad hoc top-k query on high-dimensional data is one of the top challenges in data analytics.

In a performance monitoring system, a top-k query can be a single attribute top-k query or a multiple attributes top-k query. A single attribute top-k query is based on one single attribute for ranking. For instance, a query about "top 10 CPU utilization VMs" is a single attribute top-k query based on the "CPU Utilization" attribute. A multiple attributes top-k query is based on multiple attributes for ranking and typically has preference weights for each attribute. For example, a query about "lowest 10 performance score VMs" is a multiple attributes top-k query given that a performance score formula contains multiple attributes (e.g. Performance Score = 0.4 * CPU Utilization + 0.6 * Memory Utilization).

In most cases, when the datasets are small, the preference queries can be answered without pre-processing or indexing within the response time constraints [6]. Pre-processing the data and keeping the top-k query result on the storage can improve the query response time. However, pre-processing can only handle a limited number of attribute combinations based on a common time-scale resolution. The computing cost of the pre-processing is high when the number of combinations is large for features and temporal resolutions. This issue makes it unfeasible to cover all of the temporal resolutions and feature combinations for modeling during pre-processing for the top-k query.

* Corresponding authors.
E-mail addresses: yinghuasjsu@gmail.com (Y. Qin), gheorghi.guzun@sjsu.edu (G. Guzun).

https://doi.org/10.1016/j.bdr.2021.100288

In this work, we introduce an optimized multidimensional grouping top-k query approach using the bit-sliced [23,26] and bitmap [29] indexing for performance monitoring systems. We evaluate different types of queries with 80+ properties and attributes of the infrastructure performance monitoring system. The data are collected in an hourly granularity for one month from a hybrid cloud data center environment with 9,000 virtual machines.

The project objective is to enable users to perform fast and interactive preference queries based on different time-scale windows with various criteria in real time. In the preference queries, our experiments cover the performance comparison between a baseline approach using top-k array sequential scan sorting and two approaches with bit-sliced and bitmap indexing. In the multidimensional grouping, our experiences cover the performance comparison between collection streaming grouping approach and bit-sliced grouping approach. The purpose of the experiments is to evaluate the performance gain of using bit-sliced and bitmap indexing for preference queries. We used a set of synthetic preference queries for performance evaluation. The speed of response time for indexing is the key aspect that we evaluated and recorded during the experiments.

The project generates a performance comparison matrix for top-k query use cases and indexing slicing technologies. The data samples are based on the general data type for hybrid cloud data centers. The evaluation outcome could be used as a guidance and reference for the real-time performance monitoring systems performing the top-k query. The bit-sliced and bitmap indexing approaches proposed in this project enable the users to perform ad hoc preference queries with the fast response time. The technology can be applied to various monitoring systems and time-series based datasets for data exploration and analytics.

The significant contributions of this project include the following items: First, we propose a crossing attribute filtering approach using a bit-wise operation on top of a bit-sliced indexing algorithm; meanwhile, the project evaluates three types of top-k algorithms: array sort algorithm, binning with bitmaps algorithm, and bit-sliced algorithm. At the same time, the project evaluates various types of top-k queries: single attribute top-k queries, multiple attributes top-k queries, and crossing attribute filtering top-k queries. Furthermore, the project experiments are based on hybrid cloud data center with various numbers of records up to 700K.

To further illustrate the technical aspects and experiment results, the rest of this paper is organized into several sections. Section 2 presents the background and related work. Sections 3 and 4 present the problem formation of the preference query and the multidimensional grouping approaches. Section 5 and section 6 show the cost analysis comparison among the algorithms. Sections 7 and 8 present the experiments and results. Finally, the conclusion is presented in section 10.

2. Background and related work

In this section, we elaborate on the background of the performance monitoring system, bitmap indexing, bit-sliced indexing, and top-k query.

2.1. Performance monitoring of data center

In a performance monitoring system [25] for modern data centers, the monitoring domains are usually broad and deep. The monitoring domains cover all the components in the physical infrastructures, virtual infrastructures, container platforms, and run-time application environments. This integrated model enables users to have an overview of the availability of the whole data center on one screen. Meanwhile, it generally contains deep-dive and drill-down capabilities for each of the components. The model also

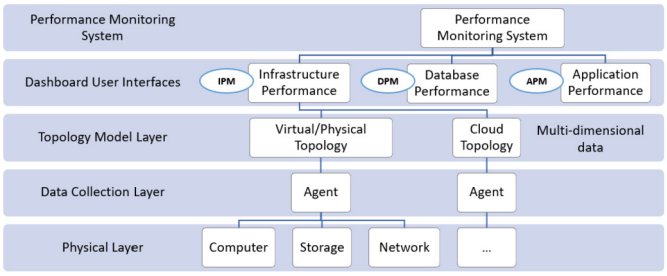


Fig. 1. 5-layer performance monitoring system. Each layer illustrates the components it contains. An integrated model is at the topology model layer.

enables the users to perform root-cause diagnostics on individual components. In previous work [25], a layering definition as shown in Fig. 1 is described.

The physical layer contains the infrastructure components that a performance monitoring system monitors. The data collection layer defines various data collection technologies and connection protocols. The topology model layer enables object-oriented model integration for the collected data. The dashboard user interface layer defines three performance monitoring domains, which include infrastructure performance monitoring, database performance monitoring, and application performance monitoring.

In this project, we run the top-k query experiments on infrastructure performance monitoring data. Through the infrastructure component API, we collect the production environment data in a real-time manner.

Meanwhile, the data queries generally need to be capable of performing across multiple domains to fulfill the analytic requirements. Previous work [24] proposed an integrated model-based approach for data center monitoring using a virtualization infrastructure, as shown in Fig. 2. In that example, a virtual center contains multiple servers. A server contains multiple virtual machines. A virtual machine contains CPU, memory, disks, etc. Each type of object has a set of properties and attributes. The CPU object has CPU utilization, CPU swap, CPU wait, and CPU co-stop, etc. performance metrics.

An integrated model brings multidimensional analytic opportunities and challenges. While each object instance has m attributes, an integrated model with n number of object types would increase the number of dimensions to n times m.

Although different types of queries are used for the business analysis on different stages, from descriptive to predictive to prescriptive (Table 1), top-k query is the first step in identifying the object's performance issues. An overview dashboard visualizes the top-k query results and works as the entrance for further analysis and investigations. In this project, we use the virtual machine operating system performance counters as the metrics for top-k query experiments.

2.2. Top-k queries

Top-k query technologies have been broadly used in various fields to get the most important answers from a large answer space. In the marketing science space, paper [11] presented a way of selling products and services through top-k probabilistic goods. [33] defined the probabilistic skyline operator and set a foundation for the top-k query for the probabilistic skyline. [34] presented a favourite probabilistic products query with a paralleling algorithm based on the model of uncertain dynamic skyline query over probabilistic product set. [31] proposed a partitioning solution for the monochromatic cases, and a pruning heuristics approach for the bi-chromatic cases on probabilistic reverse top-k queries over uncertain data. [30] used an effective pruning heuristics approach to

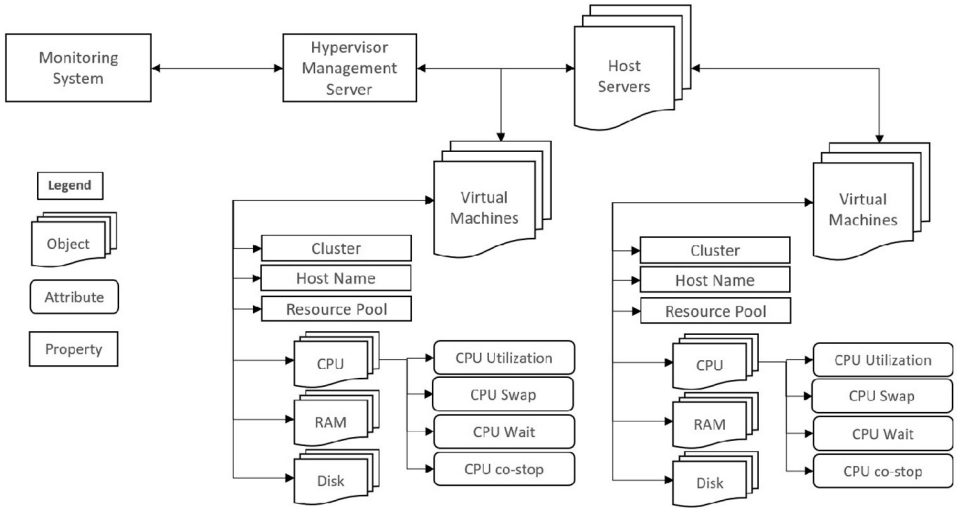


Fig. 2. A topology model in an infrastructure performance monitoring system contains objects, attributes, and properties.

Table 1
Progression of Data Analytic in Performance Monitoring Systems.

Descriptive	Diagnostic	Predictive	Prescriptive
Inventory	Dependency	Forecasting	Optimization
Events	Root Cause	Trending	Planning
Metrics	Correlation	Time-to-Full	Modeling

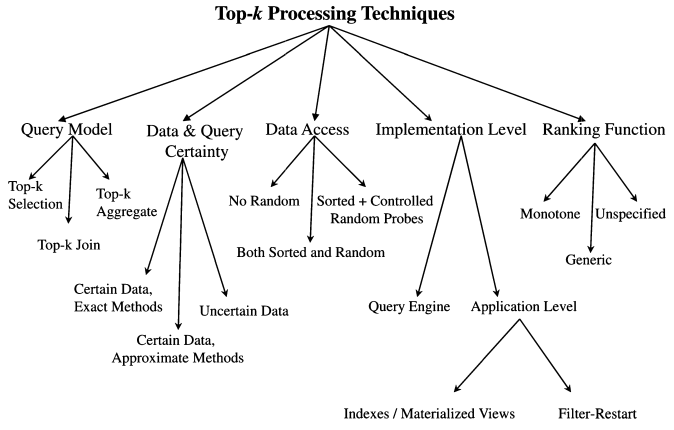


Fig. 3. Top-k query processing techniques classification [9].

discover the influential factors on the probabilistic reverse top-k queries over uncertain data.

Based on a study on the top-k queries technologies [19], top-k processing involves query optimization and indexing technology. Fig. 3 shows the top-k processing techniques discussed in paper [19].

Various top-k query techniques have been proposed: TA (Threshold Algorithm) [10], BPA (Best Position Algorithm) [1], IO-top-k algorithm [4], several database-based top-k query techniques [5,3], and other various optimizations [7,12]. However, the general top-k query techniques are not able to address the high dimensional data query challenge [9].

2.3. Bitmap indexing

Bitmap indexing employs the bit(s) (0 or 1) to encode one distinct attribute value of a column. The most popular encoding approach is k of N encoding, which uses k bit(s) to encode one distinct value.

ID	Date	VM ID	CPU Swap Time
1	2019.1.10	1001	12
2	2019.1.10	1002	11
3	2019.1.10	1003	5
4	2019.1.10	1005	7
5	2019.1.12	1006	6
...	...		...

BitMap Encoding

CPU Swap Time	
0	0 0 0 0 0
1	0 0 0 0 0
2	0 0 0 0 0
3	0 0 0 0 0
4	0 0 0 0 0
5	0 0 1 0 0
6	0 0 0 0 1
7	0 0 0 1 0
8	0 0 0 0 0
9	0 0 0 0 0
10	0 0 0 0 0
11	0 1 0 0 0
12	1 0 0 0 0

Fig. 4. Bitmap 1-of-N coding example using CPU swap time attribute. Given the CPU swap time has values of 0, 1, 2,..., or 12, 1-of-N bitmap encoding generates 13 bitmaps. Each bitmap contains N bits. For example, the bitmap-12 has 5 bits 1,0,0,0,0 with the first bit as a set-bit. This represents the first record has CPU swap time as 12 while other records have different values.

Take CPU swap time (ns) as an example: the following Fig. 4 contains a CPU swap time attribute value ranging from 0 to 12. 1-of-N encoding generates 12 bitmaps while each bitmap contains N bits. N is the number of records in the dataset.

Bitmap indexing forms a sizeable sparse matrix for a dataset that has many attributes to index. For example, a table of 100 records with 10 columns that have 10 distinct values for each column will form a 100x100 matrix for 1-of-10 encoding. A sparse matrix contains many zeros, which requires compression to optimize the storage space. Researchers have developed and implemented various algorithms for storage compression with excellent query performance at the same time. The commonly used bitmap compression algorithms include WAH, PLWAH, CONCISE, EWAH, and Roaring bitmaps.

EWAH [21] utilizes 32/64 bits to compress bitmaps based on the CPU architecture. It continues constructing 32/64 bits of data to represent the running data stream. Take 32 bits for example: the first bit of the 32 bits describes what the current compression value (0, or 1) is, the next 16 bits reflect how many 32 bits of current values are compressed, and the last 15 bits represent how many 32 bits of dirty (raw) data are stored.

The CONCISE [8] algorithm claims a 50 percent reduction in size compared with WAH. The algorithm uses the mixed fill word to reduce the worst-case memory footprint. Each 32-bit word is represented as either a 31-bit literal word or a filled word. The leftmost bit of a word indicates the original word or filled word via bit 1 and bit 0. For a filled word, the second bit indicates the filled/compressed with bit 0 or 1. The next 5 bits represent which

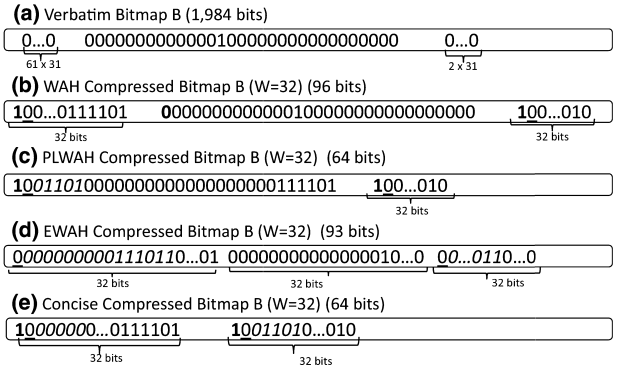


Fig. 5. Bitmap Verbatim, WAH, PLWAH, EWAH and CONCISE compression examples.

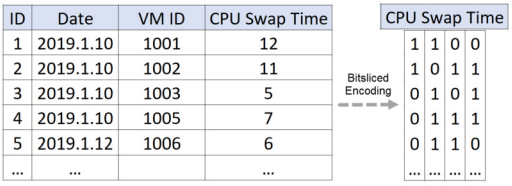


Fig. 6. Bit-sliced Encoding. In this example, the value of CPU swap time value is encoded as a binary format and thus generates a four-sliced BSI. Each of the slices contains n bits while n is the number of records.

bit will be flipped. The last 25 bits count how many 32-bit values are compressed.

Roaring [6] bitmap has a hybrid data structure which incorporates compressed and uncompressed bitmaps using sorted arrays. It separates data into dense and sparse chunks. The dense chunk contains more than 4,096 integers with a total size of 216 bits per chunk. Roaring bitmap is a more complex compression approach compared with the WAH and the CONCISE.

Fig. 5 shows an example of bitmap verbatim and its compression forms in the compression algorithms: WAH, PLWAH, EWAH, and Concise.

In this project, we use the Roaring bitmap as the binning bitmap in the single attribute top-K algorithm.

2.4. Bit-Sliced Index (BSI)

BSI [23,26] stands for Bit-Sliced Index. It uses bitmap [29] as a data structure. The k-of-N encoding refers to traditional bitmap encoding, which uses k bit(s) to encode one distinct value of an attribute. Bit-sliced encoding refers to the "slicing" of the binary representation of an attribute value set, which uses one bitmap for one slice. Fig. 6 shows an example of bit-sliced encoding. In a previous study [18] compared the BSI index approach and Sequential Top-k Algorithm (STA), as well as other state-of-the-art indexing techniques for answering top-k queries. It did not evaluate an approach with the bitmap index. The BSI uses bit-wise operations, similar to the bitmap index. However, they represent the attribute data differently. Although there are limitations on multiplication and sum for bitmaps, the performance of a bitmap for a single attribute is faster than the array sort approach. In this paper, we evaluate bitmap for single attribute top-k.

Preference (top-k) query is one of the top challenges in data analytics for large volume and high dimensional datasets [28,27,16,15]. Data scientists and researchers have been studying and proposing using various indexes for large datasets. O'Neil et al. [23] proposed the bit-sliced indexing for the multidimensional grouping query on the data warehouse. Rinfret et al. [26] introduced the arithmetic of bit-sliced addition, subtraction, top-k, and generalized range restrictions of non-Boolean form for the use case of information retrieval. Additionally the bit-vector data structures have the potential to use compression [13,17,14].

To the best of our knowledge, although the BSI has been researched extensively, it has not been used on real-time performance monitoring systems of hybrid cloud data centers. Thus, it is interesting to explore the feasibility and performance assessments of this method on data center infrastructure monitoring and optimization.

Table 2
Notation Description of Top-K Query Algorithms.

Notation	Description
n	Number of rows/records in the data
m	Number of attributes in the data
s, p	Number of slices used to represent an attribute
w	Computer architecture word size
Q	Query vector
b	Number of bins use in binning bitmap top-k

3. Top-k query proposed approach

In this section, we first define the top-k queries and then describe the three query algorithms using bit-sliced indexing (BSI), binning bitmap indexing, and array-sort.

The first query algorithm described in this section is the bit-sliced top-k query algorithm (BSItop-k). A previous study [18] introduced the bitmap operation-based slicing approach. In this project, in addition to the previous study, we propose a new cross attribute filtering based on BSI. It enables the top-k query to perform on various attribute filtering schemes.

The second query algorithm in this paper is the bitmap indexing query algorithm (Bitmaptop-k). In the previous study [18], bitmap indexing was not used for the top-k query comparison. In this project, we evaluate a bitmap approach using Roaring compression [22] as an example. Bitmaps are known to become sparse when encoding a higher range of values. Compression is necessary in this case, and we chose Roaring compression as it was shown to perform better than the word-aligned compression variants in bitmap queries [22].

The third query algorithm in this paper is the array-bubble-sort top-k algorithm (Arraytop-k). We introduce a simplified top-k array sort as a baseline for performance comparison.

3.1. Problem formulation

For clarity, we define the notations used further in this paper in Table 2. The notations are used in the problem formulation description and cost analysis formulas.

Consider R as a dataset with m attributes. Each data item t in R has numeric value $\{r_1(t), r_2(t), ..., r_m(t)\}$. Q is a query weight vector with $Q = \{q_1, q_2, ..., q_m\}$. S is a scoring function where $S(t) = Sum(q_1.r_1(t) + q_2.r_2(t) + ... + q_m.r_m(t))$. K is the number of top items, and P is the indicator of max (highest) or min (lowest) for the top items.

A single attribute query ($SAQuery$) has input $Rr(t)$, Qq, K, P. The output is K items whose $S(t) = q * r(t)$ is the K highest or lowest among all data items.

A multiple attribute query ($MAQuery$) has input of the following form: $R\{r_1(t), r_2(t), ..., r_m(t)\}$, $Q\{q_1, q_2, ..., q_m\}$, K, P. The output is K items whose $S(t) = Sum(q_1.r_1(t) + q_2.r_2(t) + ... + q_m.r_m(t))$ is the K highest or lowest among all data items.

A top-K query usually performs on a subgroup of data based on various filtering dimensions. For example, a query of "the lowest 10 performance score VMs in past 1 week" is a top-k query with date range as filtering. We denote the filtering range as F = {lowerBound, upper-bound}.

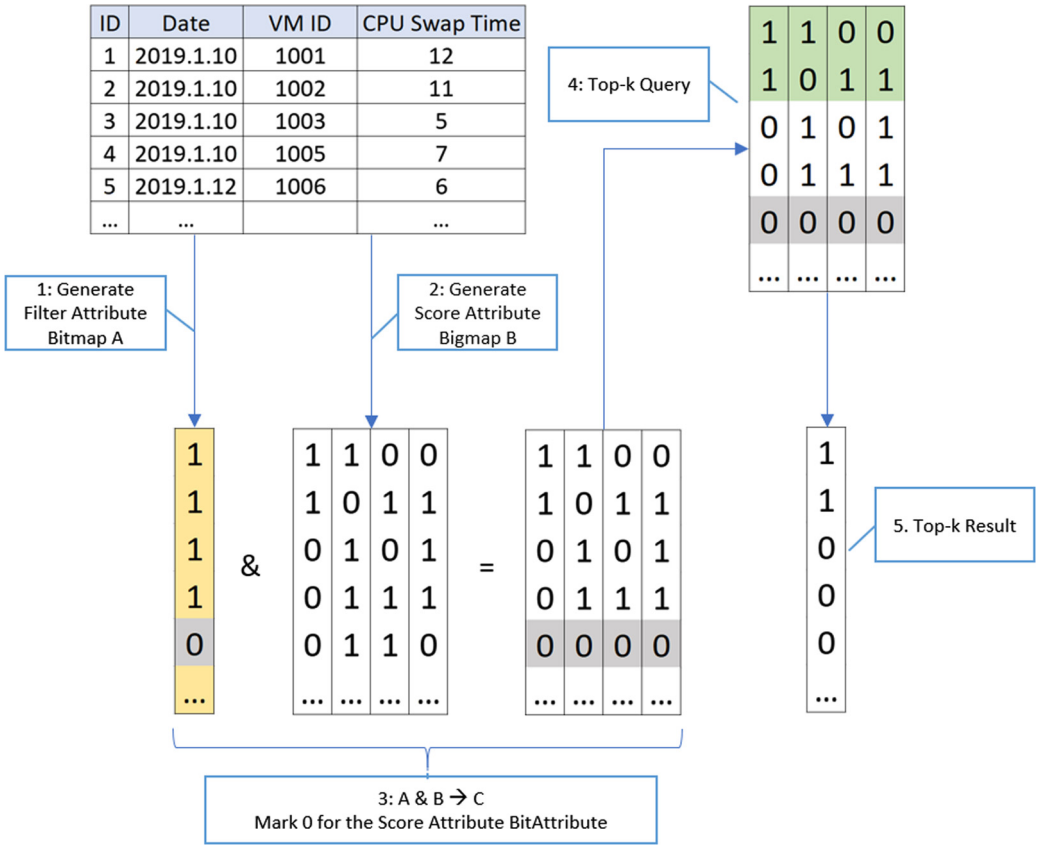


Fig. 7. Example of BSI arithmetic applied for finding top-2 tuples given a date range filtering query.

SAQueryFiltering denotes a single attribute query with filtering, while *SAQuery* denotes a single attribute query without filtering.

MAQueryFiltering denotes a multiple attributes query with filtering, while *MAQuery* denotes a multiple attributes query without filtering.

In the remaining part of this section, we describe the three approaches, BSItop-k, Bitmaptop-k, and Arraytop-k to answer the four general top-k preference queries: SAQuery, MAQuery, SAQueryFiltering, MAQueryFiltering.

3.2. BSI hybrid EWAH top-k query execution

In this project, we implement and evaluate the BSI approach by using the Hybrid EWAH bitmap as a bitmap library [13]. This section introduces BSI algorithms for the single attribute top-k query with cross attribute filtering algorithm (SAQueryFiltering) and multiple attributes weighted top-k query algorithm (MAQuery).

BSI is an implementation built on top of the bitmap compression algorithms. In the BSI algorithm, a BSI attribute represents a metric in the performance monitoring system.

For a crossing attribute filtering process, there are one filtering BSI attribute and one score BSI attribute. Without losing generality, we take the date metric as the filtering attribute, and the virtual machine CPU swap wait (ns) metric as the score attribute. Fig. 7 shows an example of performing a SAQueryFiltering with the BSI approach.

We split the SAQueryFiltering processing into the following steps.

Step 1. Generate filtering attribute bitmap A. In this example, we use the “Date” attribute as the filtering attribute. “Date” attribute represents the data collection time. As discussed in [18], BSI has a fast “range between” function which uses the bit-wise operations for the bitmap of each slice. With this, bitmap A is generated with the set-bits representing included records.

Step 2. Generate score attribute bitmap array B. In this example, “CPU swap time” is used as the score attribute. Bitmap array B contains the binary encoding of the attribute values.

Step 3. Mark 0 for the score attribute. To apply the filtering on the score attribute, we use the bit-wise operation “bitmap A AND bitmap array B” to mark the value in bitmap array B as zero if its corresponding bit in bitmap A is not a set-bit.

Step 4. Use the BSI attribute for top-k query. As discussed in [18], BSI has a fast “TopKMax” function which uses the bit-wise operations for the bitmap of each slice. From the highest bit to the lowest bit, the function iterates each bit-slice, enters the top-k items into bitmap G, and keeps the uncertain items in bitmap E. When there is enough of top-k in bitmap G, the function stops looping. If the looping reaches and checks the lowest bit but still not enough of top-k, the function picks the remaining top-k from bitmap E which contains a list of tie items. Finally, the function returns the top-k. The return list from BSI top-k function is unsorted. We add sorting for the return list.

In the BSI top-k query algorithm, we use a function to mark out-of-range records to be score = 0 for max top-k filtering. Algorithm 1 shows the pseudocode of the SAQueryFiltering steps.

For multiple attributes weighted top-k query algorithm (MAQuery), the BSI algorithm contains the following steps:

Step 1. Bit-sliced multiply attribute 1 and weight 1.

Step 2. Bit-sliced multiply attribute 2 and weight 2.

Step 3. Sum the two weighted attributes.

Step 4. Using BSI attribute for top-k Query.

The BSI top-k query algorithm for multiple attributes summation top-k pseudocode 2 is listed below with two attributes as examples.

Algorithm 1: BSI Single Attribute Top-k with Filtering.

```
Input: R - dataset;
Q - query weight of the single attribute;
k - number of top items;
P - max or min for the top;
F - filtering range;
Output: T - top k items
1  Function SAQueryFiltering(R,Q,K,P,F)
2      filterAttribute = R.rangeBetween (F)
3      scoreAttribute = R[Q]
4      scoreAttribute.setMark(notIn (filterAttribute))
5      T = top-k (scoreAttribute, k, P)
6      return T
Input: FA - the bitmap of filtering attribute
Output: mark zero for records out of range
7  Function setMark(FA)
8      numberOfSlices = this.bsi.length
9      for i ← 0 to numberOfSlices by 1 do
10         this.bsi[i]=this.bsi[i].and(FA)
11     end
```

Algorithm 2: BSI Multiple Attributes Sum Top-k.

```
Input: R - dataset;
Q - query weight of the single attribute;
k - number of top items;
P - max or min for the top;
F - filtering range;
Output: T - top k items
1  Function MAQuery(R,Q,K,P,F)
2      scoreAttribute1 = R[1].Multiply(Q[1])
3      scoreAttribute2 = R[2].Multiply(Q[2])
4      scoreAttribute = sum (scoreAttribute1, scoreAttribute2)
5      T = top-k (scoreAttribute, k, P)
6      return T
```

3.3. Roaring bitmap top-k query execution

The second approach evaluated in this project is the bitmap approach by using Roaring bitmap [22] as an implementation example. Bitmap approach works only for single attribute top-k. The binning bitmap approach cannot be used with multiple attributes weighted top-k. Since binning bitmap contains only the position of the record instead value, it requires the value addition operation to sum two attributes. It cannot achieve this via a bit-wise operation. Due to this limitation, the binning bitmap is not applicable for a multiple attributes top-k query and will be used in our experiment only for a single attribute query.

We identify four main parts of the main algorithm of the Roaring bitmap top-k query. An example of performing the top-k query on the roaring bitmap approach is shown in Fig. 8.

Step 1. Generate filter attribute bitmap A.

Step 2. Generate binned score attribute bitmap array B.

Step 3. Iterate the bitmap array B to compute a top-k bitmap.

Step 4. Select top k from the top-k bitmap.

The first step is to generate a filtering bitmap A based on the filtering criteria. In the Fig. 8 example, we use "date" attribute as the filtering attribute and "date = 2019.1.10" as the filtering criteria. In the example figure first step, it generated a filtering bitmap A as [1,1,1,1,0]. The size of bitmap A is n which is the number of records in the dataset. A set-bit in bitmap A represents a record that satisfies the filtering criteria.

The second step is to generate a bitmap array based on the top-k ranking score attribute. The number of bitmaps generated is determined by the bin value range definition. To make it a fair comparison with the bit-sliced approach, in the experiments, we set 20 as the fixed number of bins to reflect the typical maximum number of slices in the BSI. The more bins, the more memory required and the less query time. In Fig. 8 example, without losing generality, we take VM CPU swap wait (ns) as the top-k ranking score attribute. In the example figure second step, it generated a bitmap array B. The size of bitmap B[i] is n which is the number of records in the dataset. A set-bit in bitmap B[i] represents a record that has a score value within the bitmap binning value range.

The third step is to compute top-k bitmap based on the filtering bitmap A from step 1 and the score bitmap array B from step 2. Initially, the program sets the bitmap *sumB* as bitmap B[b-1] and then perform bit-wise operation "*sumB* AND A" to get an interim bitmap *c*. *b* denotes the number of bins in the score attribute bitmap array. B[b-1] is the last item in the array. If the cardinality of bitmap *c* is equal or greater than *k*, then it returns bitmap *c* as the top-k bitmap for the next step processing. If the cardinality of bitmap *c* is less than *k*, then the program iterates over the bitmap array B in a reverse order to get another bitmap array B[b-2]. For each of the B[i] where *i* is iterated from $b-1$ to 0, the system performs a bit-wise operation "sumB = sumB OR B[i]" to get an updated *sumB*. After that, the system performs a bit-wise operation "c = sumB AND A" to filter the result on bitmap *sumB* and returns the result as *c*. The iteration stops until the cardinality of *c* is equal or greater than *k* or the whole array is iterated.

The last step is to perform a sort on the returned bitmap *c* from step 3. The size of the result extracted from the above step is either equal or larger than the top-k number. In either case, the result is not sorted. Therefore, to make the top-k list result to be identical across three algorithms, further sorting is performed to get the final top-k items. The sorting used in this project is the array top-k selection. Algorithm 3 shows the pseudocode of those steps.

Algorithm 3: Bitmap Single Attribute Top-k.

```
Input: R - dataset;
Q - query weight of the single attribute;
k - number of top items;
P - max or min for the top;
F - filtering range;
Output: T - top k items
1  Function SAQueryFilteringBitmap(R,Q,K,P,F)
2      numberOfBins = b
3      while binNumber ≤ numberOfBins do
4          B[binNumber] = RoaringBitmap.bitmapOf(R[ Q ]
5          .matchIds(binNumber)
6          binNumber ++
7      end
8      A = RoaringBitmap.bitmapOf (F)
9      sumB = B[B.length-1]
10     for i ← B.length to 0 by −1 do
11         sumB = B[i] OR sumB
12         c = sumB AND A
13         if sumA.cardinality ≥ K then break
14     end
15     T = top-k (c, k, P)
16     return T
```

3.4. Array sequential sort top-k query execution

To compare the performance, we use a sequential sort algorithm as the baseline approach. The complexity of the sequential sort is $O(k \times n)$ where k is the number of top-k ranking and n is the total number of records.

Given a dataset with the number of records *n*, for the top-k query with $k < \log_2 n$, a sequential sort is faster than the other sorting algorithms with the time complexity as $O(n \log n)$. In Fig. 9, the threshold line $k < \log_2 n$ shows the upper-bound of *k* is up to 20+ for a several millions dataset. Given that our typical dataset has less than 1 million data d$n = 700K >> 2^{15}$, a sequential sort algorithm has optimal performance as a baseline.

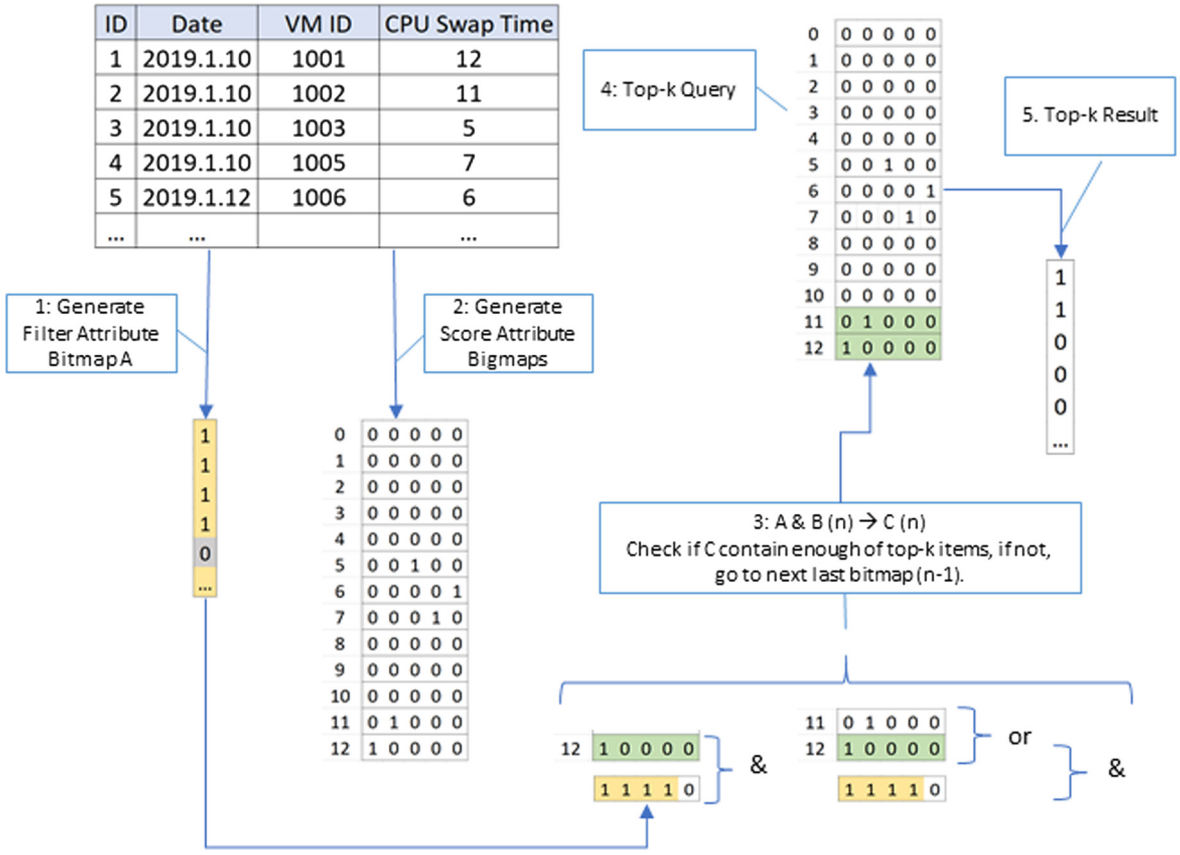


Fig. 8. Example of Roaring bitmap arithmetic applied for finding top-2 tuples given a date range filtering query.

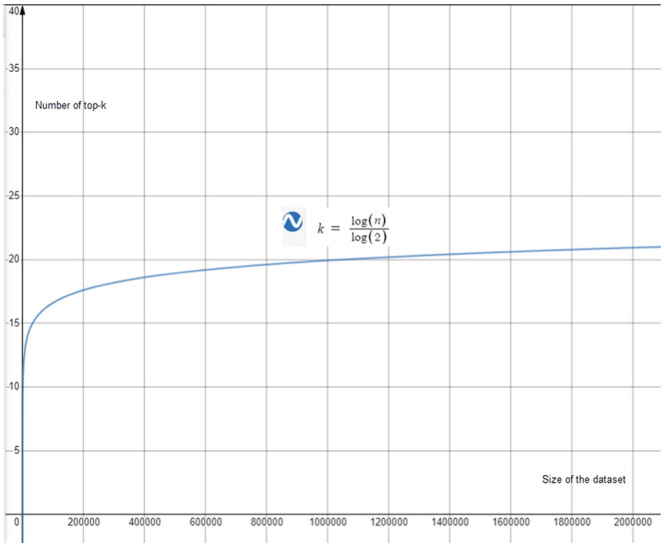


Fig. 9. Dataset size and top-k threshold line for sequential sort algorithm.

The algorithm of the top-k sequential sort is similar to the regular bubble sort. The key difference is that it bubbles up only k items. Algorithm 4 illustrates the single attribute sequential sort top-k query pseudocode. Algorithm 5 demonstrates the multiple attributes sum top-k pseudocode with two attributes as an example.

4. Multidimensional grouping proposed approach

In this section, we describe the two grouping algorithms using bit-sliced indexing (BSI) and Java Stream API.

Algorithm 4: Array Single Attribute Top-k.

Input: R - dataset;
Q - query weight of the single attribute;
k - number of top items;
P - max or min for the top;
F - filtering range;
Output: T - top k items

```
1 Function SAQueryArray(R,Q,K,P,F)
2     for i ← 0 to k by 1 do
3         for j ← 0 to n − k − 1 by 1 do
4             swap the positions if R[Q] (i) >R[Q] (j)
5         end
6     end
7     T = R.range[0,k]
8     return T
```

Algorithm 5: Array Multiple Attributes Sum Top-k.

Input: R - dataset;
Q - query weight of the single attribute;
k - number of top items;
P - max or min for the top;
F - filtering range;
Output: T - top k items

```
1  Function MAQueryArray(R,Q,K,P,F)
2      for i ← 0 to k by 1 do
3          s(i) = R[1](i)*Q[1](i) + R[2](i)*Q[2](i)
4          for j ← 0 to n − k − 1 by 1 do
5              s(j) = R[1](j)*Q[1](j) + R[2](j)*Q[2](j)
6              swap the positions if s(j) >s(j)
7          end
8      end
9      T = R.range[0,k]
10     return T
```

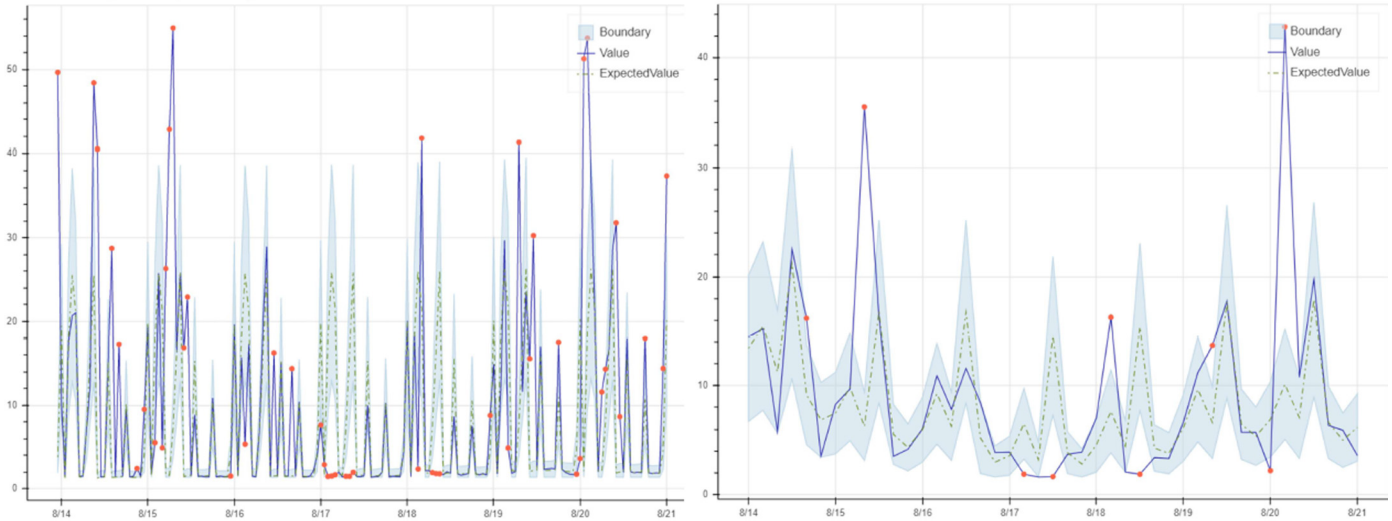


Fig. 10. Grouping results of two different types of data granularity: per hour and per four-hours.

The first grouping algorithm described in this section is the bit-sliced grouping algorithm. We introduce a bit-sliced algorithm for multidimensional grouping. It enables users to aggregate performance metrics on various dimensions for exploratory analysis.

The second grouping algorithm is a baseline approach for performance comparison. We use the Java Stream API for grouping and summation. The Java Stream API uses functional programming with simple expression to achieve grouping and summing in one line. As a built-in library in Java 8, stream API is a suitable baseline approach for performance comparison.

4.1. Problem formulation

In the performance monitoring system, the majority of the grouping functions are multidimensional. A typical use case is to query performance metrics grouping by both monitored object ID and temporal dimensions. For example, to find out which virtual machines are the top CPU consumers last week, the performance monitoring system could query the virtual machine performance metrics by rolling up to weekly granularity. In anomaly detection, the system aggregates the metrics on various granularity to find out the time pattern related anomalies. The temporal granularity could be 15 minutes, 1 hour, 4 hours, 24 hours, day-of-week, hour-of-day, and day-of-month, etc. On each of the grouping datasets, we calculate the aggregate metrics including sum, max, min, mean, and standard deviation, etc.

Fig. 10 shows the grouping results of two different data granularity values. One is per hour. Another is per four-hours. During the exploratory data analytic, users usually evaluate various results of different types of data granularity. In this example, it is to evaluate what type of granularity would be suitable for anomaly detection. On the visual level, the four-hours aggregated data has high accuracy for anomaly detection and fewer false-positive alarms than the per hour data has.

With the annotations described in Table 3, we formulate the multidimensional grouping problem in this section.

Consider R as a dataset with m attributes. Each data item t in R has numeric value $\{r_1(t), r_2(t), ..., r_m(t)\}$. G is a grouping input vector with $G = \{g_1, g_2, ..., g_x\}$. Q is a query weight vector with $Q = \{q_1, q_2, ..., q_m\}$. O is a grouping output vector with $O = \{O_1, O_2, ..., O_m)\}$.

For a multiple attributes grouping ($MAGrouping$) with single score attribute, it has the input of the following form: R, $Q\{q\}$, $G\{g_1, g_2, ..., g_x\}$. The output is $O\{O_1, O_2, ..., O_m\}$.

Table 3
Notation Description of Grouping Algorithms.

Notation	Description
n	Number of rows/records in the data
x	Number of attributes grouping-by
m	Number of attributes rolling-up values after grouping
s	Number of slices used to represent an attribute
w	Computer architecture word size
c	Cardinality of one attribute
d	Number of days in a month for data collection
h	Number of hours in a day for data collection
t	Data collection interval in the unit of minutes
v	Number of objects monitored
G	Grouping-by attributes
O	Grouping query output
Q	Query vector

In the remaining part of this section, we describe the two approaches, BSIGrouping and StreamGrouping to perform the multidimensional grouping: MAGrouping.

4.2. BSI hybrid EWAH grouping

In general, a grouping process contains filtering and summation operations. The BSI grouping algorithm leverages the bit-sliced index for both filtering and summation. In the previous project [18], we have the summation algorithm for BSI but that algorithm is limited on "bit-sliced bitmap ADD a constant value" operation. In this project, we introduce an algorithm for the summation inside the bit-sliced attribute itself. An example of performing grouping on BSI data is shown in Fig. 11.

We split the grouping processing with BSI into four steps:

Step 1 - Generate grouping attribute bitmap arrays for each grouping attribute in $G\{g_1, g_2, ..., g_x)\}$.

Step 2 - Generate BSI attribute for the score column $R\{r_1(t), r_2(t), ..., r_m(t)\}$.

Step 3 - Generate filtering bitmap array through permutation of grouping attributes bitmaps.

Step 4 - Filter BSI attribute value by performing a bit-wise operation between bit-slice and the filtering bitmap.

Step 5 - Calculate the sum of the BSI attribute. The position of the bit-slice is i. i has the range from 0 to s where s denotes the number of slices of the score attribute BSI. p_i is the i power of 2. The cardinality of the bit-slice is c_i. In this step, the program calculates the sum of the product of the cardinality c_i and p_i.

The summation result is: $\sum_{i=1}^{s} c_i.p_i$.

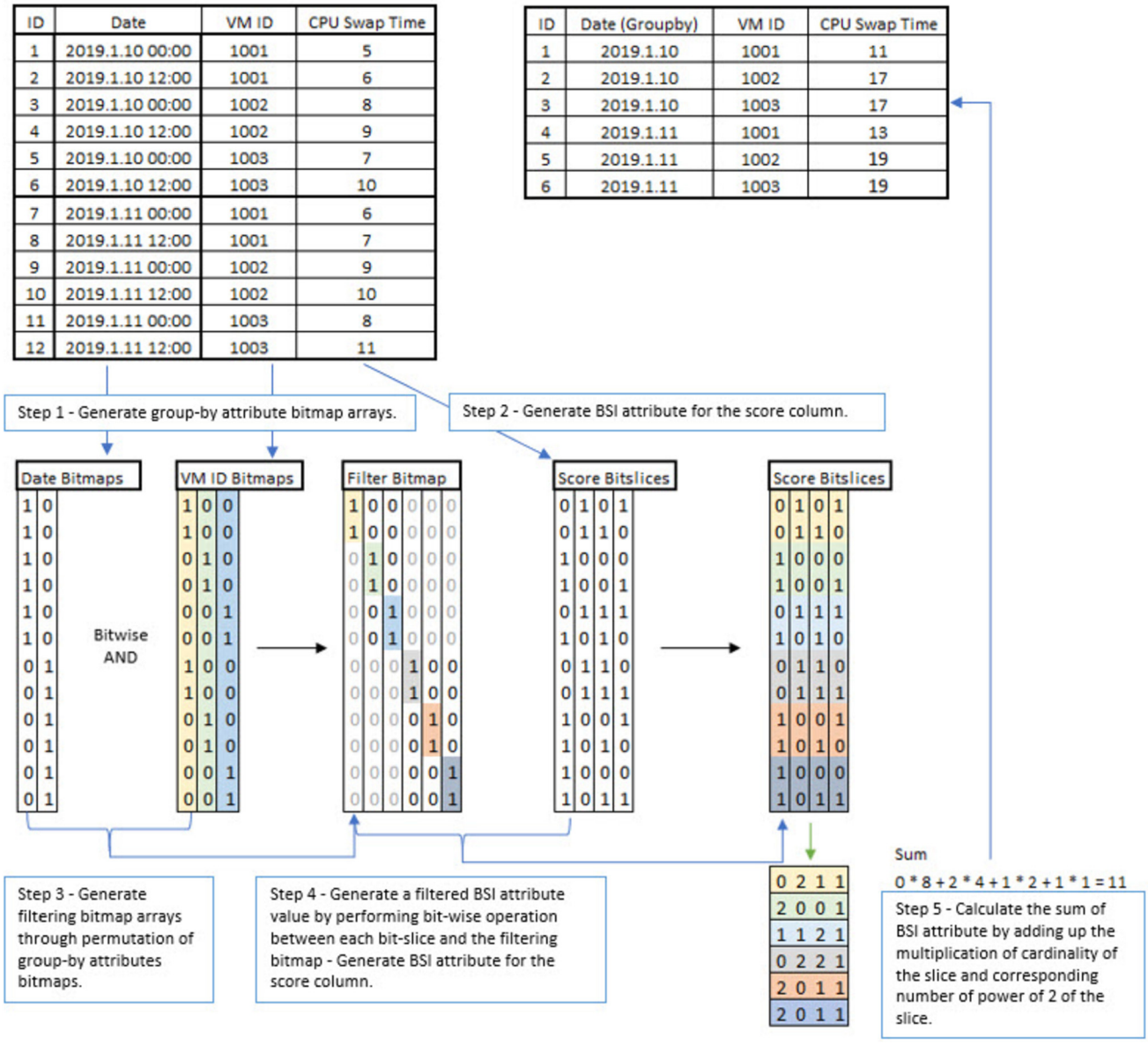


Fig. 11. Example of BSI arithmetic applied for calculating the sum of CPU swap time metric given a grouping on attributes: date and virtual machine ID.

Algorithm 6: BSI Multidimensional Grouping.

Input: R - dataset;
Q - query weight of the single attribute;
GBBitMaps - group-by attributes bitmaps;
Output: O - grouping output

Function `MAGroupingBSI(`*R,Q,GBBitMaps*`)`
  filterBitMaps = generatePermutation (GBBitMaps)
  scoreAttribute = R[q]
  for *each of the filterBitMap i in filterBitMaps* **do**
    for *each of the slice j in scoreAttribute* **do**
      slice[j] = slice [j] AND filterBitMaps[i]
    end
  end
  O = BSISum (scoreAttribute)
  return O

Input: scoreAttribute - the BSI of filtered score attribute
Output: O - the summation value of socre attribute

Function `BSISum(`*scoreAttribute*`)`
  numberOfSlices = this.bsi.length
  for $i \leftarrow 0$ **to** *numberOfSlices* **by** 1 **do**
    O = O + this.bsi[i] * Math.pow(2, i)
  end

4.3. Java collection stream grouping

Our project experiments are built on Java 8. On Java 8, there is a stream API that provides database SQL like query type operations for Java collection objects. For example, operations include sum, grouping, and filter, etc. We use this as the baseline approach for performance comparison with the BSI approach.

In Java Stream, it uses the source data structure collection as input, and produces pipeline data. On the pipeline data in Java Stream, we could perform specific operations. The Java Stream operations use functions as parameters, from simple lambda expression to complex functionality.

In the Java 8 stream definition, a stream is a pipeline of functions that can be evaluated. Streams can transform data through either intermediate operations or terminal operations. For intermediate operations, the common examples include map, filter, distinct, sorted, and peek. For terminal operations, it includes collectors operations such as collect, findAny, etc. The return data of terminal operations is a map.

Algorithm 7 illustrates using Java Stream to compute the sum of memory usage of virtual machines having memory allocation larger than 32GB.

Algorithm 7: Java Stream Example - Filter and Sum.

```
Input: Data - dataset;
Output: The sum of memory usage of virtual machines having memory
        allocation larger than 32GB
1 Data.stream()
2 .map(Memory::getMemoryMemory) .filter(memoryAllocation − >
  a.getAllocation() > 32GB) .map(Memory::getMemoryUsed) .reduce(0,
  Integer::sum)
```

In the content of our grouping operation, Java Stream API includes grouping and sum at the terminal operations. Algorithm 8 illustrates using Java Stream to compute the sum of memory usage of virtual machines grouping by virtual machine ID.

Algorithm 8: Java Stream Example - Grouping and sum.

```
Input: R - dataset;
Q - query weight of the single attribute;
G - group-by attributes;
Output: O - grouping output
1 Map<List<String>, Integer> O = Stream.of(data.elements).collect(
  Collectors.groupingBy(x -> { ArrayList<String> G=new
  ArrayList<String>(); for(int c:G) G.add(x.attributes[c]); return G;
  },Collectors.summingInt(R::getScore(R))));
2 return O
```

Java Stream is a pipeline process and could be expressed in one line. For clarity, we split the grouping processing with Java Stream into two steps:

Step 1. Get grouping attributes and score attribute using lambda functions.

Step 2. Aggregate results using the collector grouping function. Java *groupingBy* is a collector which works with the stream API terminal operation of collect. It supports returning sum, min, max, and average as the grouping aggregation values.

Fig. 12 shows an example of performing grouping on Java Stream data.

5. Top-k query cost analysis

In previous paper [18], we analyzed the space complexity of BSI approach comparing with sorted list and raw data. The BSI space complexity is 10 times smaller than sorted list and 5 times smaller than raw data. In this section, we analyze the cost of computing top-k preference queries in terms of time complexity for three approaches described in Section 3.

For clarity, we define the notations used in this paper in Table 2. The notations are used in the problem formulation description and cost analysis formulas.

Table 4 shows a summary of the complexity of the three approaches. The rest of this section describes the cost analysis details for each algorithm.

5.1. BSI hybrid EWAH top-k query

In the BSI top-k approach, the top-k preference queries are processed using bit-wise operations over the bit slices. Therefore, the number of slices (s), the word size of the computer CPU architecture (w), the number of attributes (m), and the number of non-zero weights in the query are the key factors of the time complexity. As analyzed in the article of BSI implementation [18], for single attribute top-k query with weight 1, the BSI complexity could be expressed as:

Table 4
Time Complexity of Top-K Query Algorithms.

Approach	Single Attribute	Multiple Attributes Sum
BSI	$O\left(s\frac{n}{w}\right)$	$O\left(sm\frac{n}{w}\right)$
Bitmap	$O\left(\frac{n}{b}+kk\right)$	NA
Array	$O\,(nk)$	$O\,(nk+n)$

$$O\left(\frac{n}{w}\right) \tag{1}$$

For multiple attributes top-k query, with the number of attributes m and weight 1, the BSI complexity can be defined as:

$$O\left(sm\frac{n}{w}\right) \tag{2}$$

5.2. Roaring bitmap top-k query

As shared in the article of Roaring bitmap [22], the two elements (e.g. A and B) "AND" and "OR" bit-wise operation in the Roaring implementation have the following complexity:

- Time complexity = $O\,(n)$ when both two elements are bitmaps;
- Time complexity = $O\,(n)$ when two elements has one bitmap and one array;
- Time complexity = $O\,(nlogn)$ when both two elements are arrays.

Therefore, the best case time complexity is $O\,(n)$ while the worst case time complexity is $O\,(nlog(n))$.

In this project, we use a binning approach that contains b number of bins. Each bin is represented as a bitmap. A bin has maximum $\frac{n}{b}$ number of set bits. In step 3 which computes top-k bitmap in the bitmap single attribute top-k algorithm, the maximum number of bit-wise operations is b when the data has skewed distribution and most of the data fall into lowest value bin. For normally distributed data, the first highest value bin contains all the top max items. The number of bit-wise operations is one "and." Therefore, step 3 best case time complexity is $O\left(\frac{n}{b}\right)$ while the worst-case time complexity is $O\left(\frac{n}{b}\frac{n}{b}\right)$.

In step 4, after retrieving the top-k bitmap from step 3, the number of results extracted is either equal to or larger than the top-k number. To get a sorted top-k, a further sorting on the extracted list is performed to get the final top-k items. The sorting used in this project is the array top-k selection. The complexity of the sorting depends on the cardinality of the returned top-k bitmap. For the best case with the number of results extracted equals to k, its the complexity is $O\,(kk)$. For the worst case in which the last bin bitmap is full with $\frac{n}{b}$ items, where n is number of records of the dataset and b is number of bins, the complexity is $O\left(k\frac{n}{b}\right)$.

The combined complexity of step 3 and step 4 has the best case complexity as:

$$O\left(\frac{n}{b}+kk\right) \tag{3}$$

The worst case complexity is:

$$O\left(\frac{n}{b}\frac{n}{b}+k\frac{n}{b}\right) \tag{4}$$

5.3. Array sequential sort top-k query

Quicksort has the time complexity of $O\,(nlog(n))$.

Array sequential sort for top-k has the time complexity of $O\,(nk)$.

As the line $k = log(n)$ illustrates in Fig. 9, when the $k <= log(n)$, the top-k array-bubble-sort has better performance than quicksort.

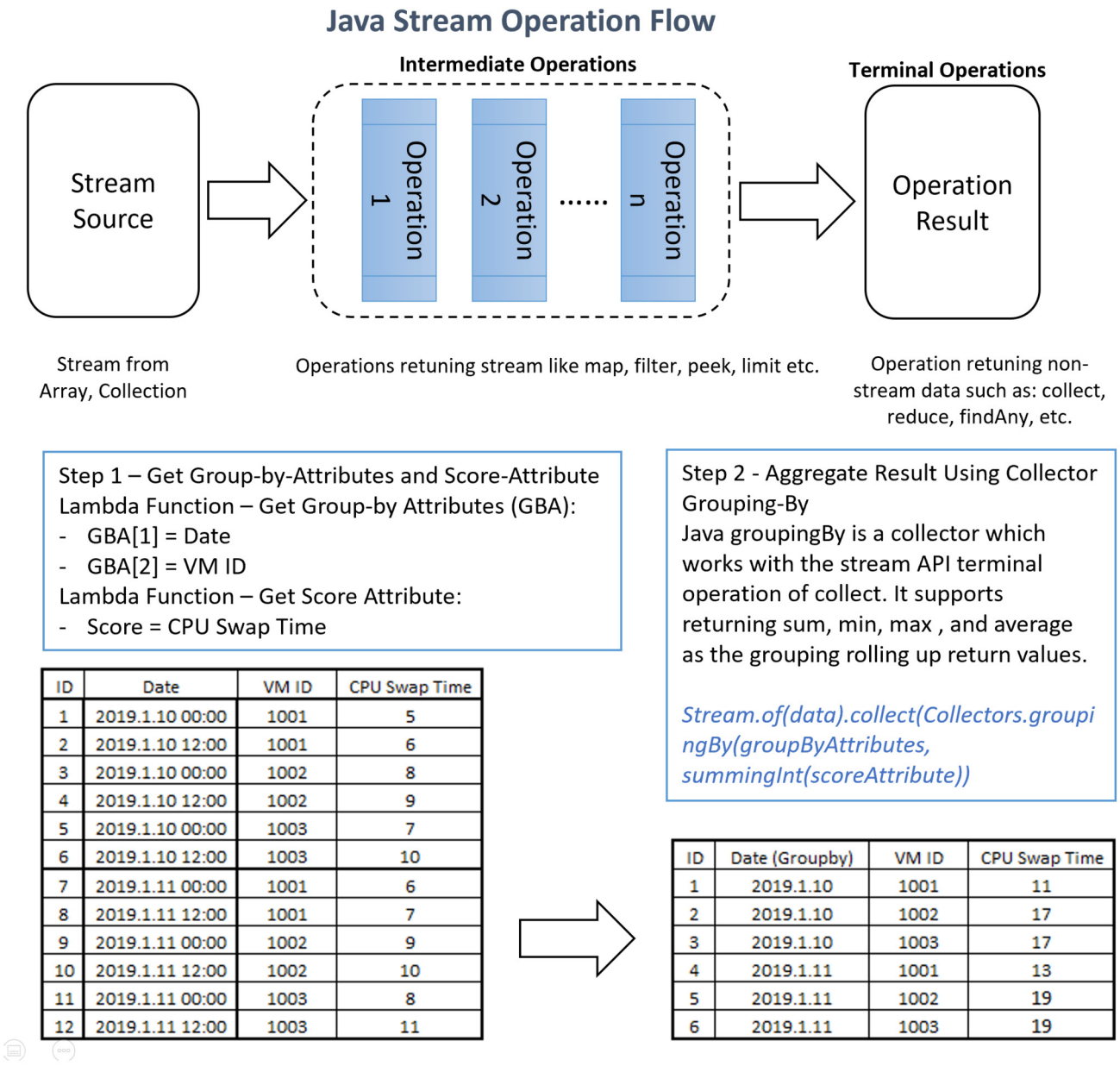


Fig. 12. Java Stream Grouping Data Flow.

In this project, we use datasets [20] with sizes of 100K, 300K, and 700K records. The $log(n)$ figures of those datasets are much higher than the k = 15. Therefore, the array-bubble-sort has lower complexity than quicksort for any given $k < 20$.

In the multiple attributes top-k, other than the sort, there is a step to multiply the attributes with weights and calculate the sum. The complexity of the multiplication and sum is $O(n)$.

Therefore, the combined complexity of the array approach is:

$$O(nk + n) \tag{5}$$

5.4. Comparison of top-k approaches

To compare the time complexity of those three approaches, we plot the big O on the Fig. 13. When the number of records exceeds about 100K, the bitmap worst-case and array-bubble-sort time complexities are 10 to 100 times bigger than the BSI approach. Bitmap best case can have either smaller or bigger time complexity than BSI depending on these moving parameters k, b, m, s, and w. Fig. 13 is based on a typical parameter set of performance monitoring system: $10 < k < 20$, $2 < b < 32$, $1 < m < 5$, $2 < s < 32$, and $w = 32$ or 64. By varying parameter k, the time complexity expressions in the bitmap best case and BSI approaches have same term $k * k$. Their difference in performance is not affected by different k parameters. Given that $n >> k$, as k increases, time complexity of the array-bubble-sort increases in a factor of n while the time complexities of BSI and bitmap increase in a factor of k which is much smaller than n. Therefore, without losing generality, our project top-k query experimental evaluation uses $k = 15$.

6. Multidimensional grouping cost analysis

In this section, we analyze the cost of computing top-k preference queries in terms of time complexity for two grouping approaches described in Section 4.

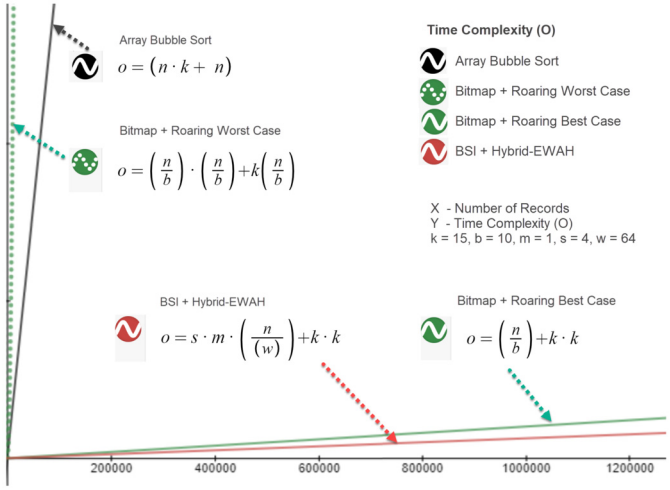


Fig. 13. Big O comparison of array-bubble-sort, bitmap worst case, bitmap best case, and BSI. The X-axis is for the number of records. The Y-axis is for time complexity (O). k, b, m, s, and w are the parameters which impact the big O. BSI big O shows as a solid red line. Array big O shows as a black dot-line. Bitmap big O has two lines: worst case as green dotted line and best case as a green solid-line. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Table 5
Time Complexity of Grouping Algorithms.

Approach	Single Attribute	Multiple Attributes
BSI	$O\,(sc)$	$O\left(s\prod_{i=1}^{g} c_i\right)$
Bitmap	NA	NA
Array	$O\,(n)$	$O\,(n)$

For clarity, we define the notations used further in this paper in Table 3. The notations are used in the problem formulation description and cost analysis formulas.

Table 5 shows a summary of the complexity of the three approaches. The bitmap approach time complexity is listed as NA due to the high time complexity of using bitmap for summation in the grouping. The rest of this section describes the details of the cost analysis for each algorithm.

6.1. BSI hybrid EWAH grouping

In the BSI grouping approach, the grouping rollup is processed using bit-wise operations over the bit slices. Therefore, the number of slices (s), the word size of the computer CPU architecture (w), the number of grouping-by attributes (g), and the cardinality of the grouping-by attributes are the key factors of the time complexity. For single attribute grouping, the BSI complexity could be expressed as:

$$O\,(sc) \tag{6}$$

In multiple attributes grouping, with the number of attributes g, the BSI complexity can be defined as:

$$O\left(s\prod_{i=1}^{g} c_i\right) \tag{7}$$

6.2. Java stream grouping

Based on the Java documentation, the time complexity of Java Stream API for grouping is $O\,(n)$. n is the number of records in the dataset. The time complexity of Java Stream API is:

$$O\,(n) \tag{8}$$

6.3. Comparison of grouping approaches

Java Stream API approach time complexity is determined by the number of records in the dataset. BSI grouping approach time complexity depends on two factors. One factor is the number of slices in the score BSI attribute. Another factor is the sum of the cardinality of the grouping-by attributes. The number of slices in a BSI attribute ranges from 0 to 32. The cardinality of grouping-by attributes varies on different systems.

Taking a temporal dimension grouping as the example, there are d = 30 days and h = 24 hours per day in a monthly dataset. v is the number of monitored objects.

For a daily grouping, the time complexity is:

$$O\,(dvs) \tag{9}$$

For an hourly grouping, the time complexity is:

$$O\,(dhvs) \tag{10}$$

The number of records in an infrastructure monitoring system is determined by the number of objects being monitored and the data collection interval. For a monitoring system with v number of virtual machines with t minutes data collection interval, the number of monthly records is:

$$n = dh\frac{60}{t}v \tag{11}$$

For a typical infrastructure monitoring system of virtual machines, the time complexity of Java Stream API is:

$$O\left(dh\frac{60}{t}v\right) \tag{12}$$

For a generic temporal grouping for the infrastructure monitoring system, the BSI grouping approach has better time complexity than the Java Stream API approach.

To compare the time complexity of those two approaches, we plot the big O on Fig. 14.

7. Top-K query experimental evaluation

In this section, we present the results of evaluating the three approaches: the bit-sliced indexing (BSI), bitmap, and sequential top-k sort.

We first describe the experimental setup and the datasets used. Then, we compare the single attribute query performance results of 80+ individual attributes. Next, we compare the multiple attributes query performance results of 80+ different combinations of two attributes sum.

In the performance measurement, we don't measure bin creation time for the bitmap index. When there are new data streaming in, additional time would be required to fit the new data into the bins. For BSI, new data are appended to the BSI attributes, thus there is no need for re-indexing.

7.1. Experimental setup

All the experiments in the project were run on the following environment configuration.

- OS: Windows 10 Enterprise
- Processor: Intel Core i7-7820 HQ @2.90GHz
- RAM: 32GB
- System Type: 64-bit Operating System, x64-based CPU
- Implementation: Java JDK 1.8 version.

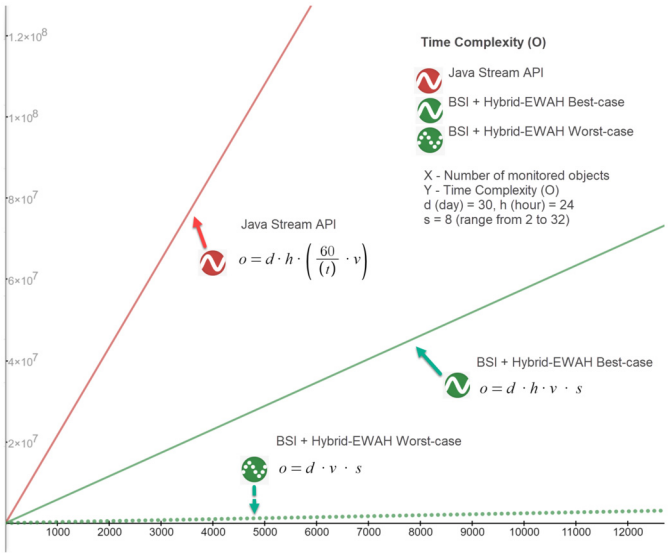


Fig. 14. Big O comparison of BSI and Java Stream API for grouping. The X-axis is for the number of monitored objects (VM as an example). The Y-axis is for time complexity (O). BSI big O grouping by hours and virtual machines shows as a solid green line. BSI big O grouping by days and virtual machines shows as a dotted green line. Java Stream big O shows as a red solid line.

Table 6
Infrastructure Performance Monitoring Data Attributes - Property, Temporal and CPU Metrics.

Attribute	Cardinality
Start Time	2070
End Time	2070
ESX Server	13
Virtual Machine	333
VM UUID	333
datastore name	12
VM CPU used(MHz)	30903
VM CPU totalHz(MHz)	262
VM CPU cores	14
VM CPU percent ready time(%)	82
VM CPU summed percent ready(%)	260
VM CPU latency(%)	505
VM CPU swap wait(ms)	1140
VM CPU coStop(ms)	13564
VM CPU swap wait percent(%)	34
VM CPU coStop percent(%)	189

7.2. Dataset

In this project, we use the hybrid cloud data center virtualization infrastructure topology model and data [24]. Fig. 2 demonstrates the integrated topology model of a typical virtualization infrastructure of a data center. The integrated model-based approach facilitates diagnostic analysis. It has been broadly used in the dependency drill down, root cause diagnostic, and event correlation analytic.

A topology model contains objects, properties, and attributes. In this project, the top-k queries are based on the object and metric which can be both used in the filtering criteria. The metric is used as the score measurement to get the top-k ranking.

The virtual machine performance metrics are collected from the virtualization hypervisor web service API with hourly granularity for one month. The dataset [20] contains 9,000 virtual machines, 700K+ rows of records, and 80 different attributes. The size of the data is around 700K * 80 * 8 bytes = 448 MB. It is pre-processed to align the units scale. Its attributes are illustrated in Table 6, 7, and 8.

Table 7
Infrastructure Performance Monitoring Data Attributes - Memory Metrics.

Attribute	Cardinality
Start Time	2070
End Time	2070
ESX Server	13
Virtual Machine	333
VM UUID	333
datastore name	12
VM memory allocated(GB)	33
VM memory used(GB)	316
VM memory capacity(GB)	37
VM memory active(GB)	135
VM memory balloon(GB)	69
VM memory balloon target(GB)	69
VM memory granted(GB)	190
VM memory overhead(GB)	3
VM memory shared(GB)	294
VM memory swapped(GB)	32
VM memory zero(GB)	268455
VM memory shares	20
VM memory reservation(GB)	13

Table 8
Infrastructure Performance Monitoring Data Attributes - Storage, Availability Metrics and Performance Score.

Attribute	Cardinality
Start Time	2070
End Time	2070
ESX Server	13
Virtual Machine	333
VM UUID	333
datastore name	12
VM ds allocated(GB)	209
VM space used(GB)	8545
VM transfer rate(kb/s)	114013
VM uptime percent(%)	38
VM OS uptime seconds(s)	283603
VM Disk Latency score	913
VM CPU contention score	946
VM CPU utilization score	501
VM Memory utilization score	302
VM Memory Balloon score	228
VM Memory Swapped score	96

7.3. Performance measurement: single attribute (CPU swap time)

In the first step of our experiments, we take one performance metric CPU swap time(ns) as a single attribute for performance comparisons of the three algorithms. The performance counter CPU swap time is the time that a virtual machine CPU waits for swap page-ins. The query response time is captured with and without crossing attributes filtering.

For the 300K data experiment, the BSI approach is 12 times faster than the baseline approach top-k sequential sort, but it is 3 times slower than the binning bitmap approach for the use case of the top-k query without crossing attributes filtering.

The BSI approach is 40 times faster than the baseline approach top-k sequential sort, and 3 times faster than the binning bitmap approach for the use case of the top-k query with crossing attributes filtering.

By using the same dataset with VM ID grouping, the total number of records goes down to 10K after grouping. For this 10K data experiment, the BSI approach is 4 times faster than the baseline approach top-k sequential sort, but it is 2 times slower than the binning bitmap approach for the use case of a top-k query without crossing attributes filtering.

Meanwhile, the BSI approach is 2 times faster than the baseline approach top-k sequential sort, and 3 times faster than the binning bitmap approach for the use case of the top-k query with crossing

Top 15 from 300K Data

Algorithm	Query Time (ms)	1 (ns)	2 (ns)	3 (ns)	4 (ns)	5 (ns)	6 (ns)
Array Bubble Sort	37.69	36,565,700	37,343,000	36,257,701	36,528,199	42,507,599	36,949,500
BSI + EWAH	3.47	4,250,600	3,995,301	2,910,100	2,509,300	3,566,400	3,568,401
Roaring	0.77	755,600	662,201	983,400	677,600	809,600	733,000

Top 15 from 300K Data with Crossing Attributes Filtering

Algorithm	Query Time (ms)	1 (ns)	2 (ns)	3 (ns)	4 (ns)	5 (ns)	6 (ns)
Array Bubble Sort	36.38	34,611,900	42,481,000	34,128,601	32,280,400	36,088,400	38,711,900
BSI + EWAH	0.85	731,700	1,035,300	726,699	632,300	1,007,000	987,701
Roaring	2.84	2,690,200	2,764,799	2,833,100	2,647,200	2,804,000	3,286,900

Top 15 from 10K Data with Grouping

Algorithm	Query Time (ms)	1 (ns)	2 (ns)	3 (ns)	4 (ns)	5 (ns)	6 (ns)
Array Bubble Sort	12.15	12,092,600	12,116,400	12,055,501	11,897,500	11,842,500	12,904,800
BSI + EWAH	3.25	3,403,800	4,074,000	2,865,300	2,827,000	3,682,701	2,646,500
Roaring	1.48	1,397,500	1,323,400	1,389,400	1,674,100	1,554,901	1,527,900

Top 15 from 10K Data with Grouping and Crossing Attributes Filtering

Algorithm	Query Time (ms)	1 (ns)	2 (ns)	3 (ns)	4 (ns)	5 (ns)	6 (ns)
Array Bubble Sort	11.78	13,044,099	11,394,800	11,148,800	12,684,600	10,749,000	11,629,100
BSI + EWAH	5.86	9,023,600	5,720,999	4,504,999	5,387,401	4,348,800	6,165,500
Roaring	17.89	16,235,301	13,887,200	14,055,299	18,035,300	20,604,999	24,530,400

Fig. 15. Performance comparison of the top-k query from 300k and 10k data with/without crossing attributes filtering for one single attribute CPU swap time. Query time is measured as nanoseconds. Parameters: n=300K; n= 10K; k =15.

attributes filtering. Fig. 15 demonstrates the query performance result in nanoseconds unit.

7.4. Performance measurement: single attribute (70+ metrics)

After measuring the top-k query performance of a single attribute CPU swap time, we further analyze 70 more metrics and their performance distribution. We use the 75, 50, and 25 percentiles to present a statistic summary of our measurements.

Based on our measurement results, the BSI and binning bitmap approaches are both 10 times faster than the array approach. Comparing BSI and bitmap approaches, while there are a few attributes with better performance in the binning bitmap approach, BSI is 2 to 3 times faster on average than the binning bitmap approach on both the 700K and 200K dataset. Fig. 16 demonstrates the performance summary of the 700K dataset single attribute top-k query performance results in the units of a millisecond and a nanosecond log 10 scale. Charts A, B, C, D show performance measurements in the unit of a millisecond, while the charts E and F have their analyses in a log 10 scale of nanoseconds. Charts A and B are the measurements for the array, bitmap, and BSI. Charts C and D provide a zoom-in view with a perceivable comparison on bitmap and BSI measurements. Charts A, C, E are the results of a 700K dataset, while charts B, D, F represent the results of a 200K dataset.

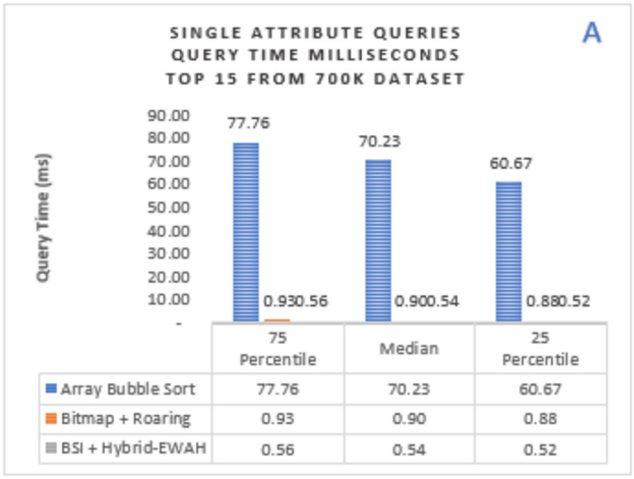
7.5. Performance measurement: multiple attributes

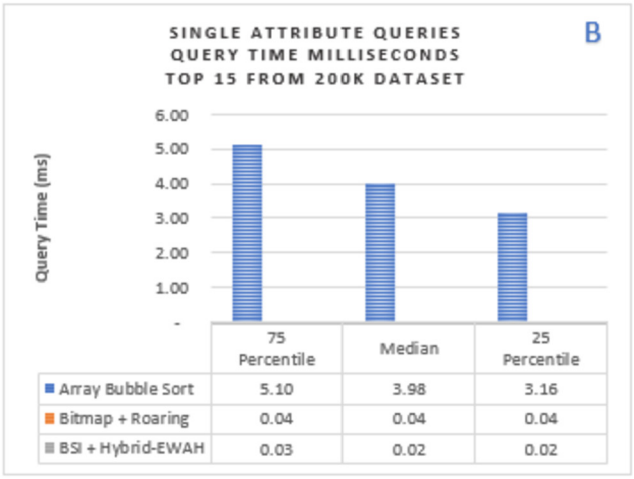
After the 70+ single attribute experiments, to further evaluate the multidimensional top-k query performance, we measured the execution time of multiple attributes sum of both BSI and array. To analyze the performance norm of the BSI, binning bitmap, and sequential sort approaches among all those various attributes combination, we use the 75, 50, 25 percentiles graph for comparison once again. Based on our measurement results, BSI is about 4 times faster than the sequential sort approach for large dataset 700K data.

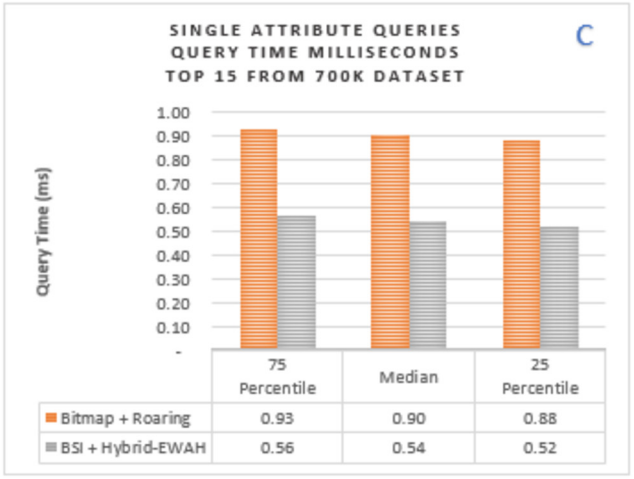
Fig. 17 demonstrates the 75, 50, and 25 percentile measurements of 70+ multiple attributes query performance comparison of BSI and array. In Fig. 17, charts A and B show performance result in the unit of a millisecond, while charts C and D have their analyses in the log 10 scale of nanoseconds. Chart A and chart C are the results of the 700K dataset, while chart B and chart D hold the results of the 200K dataset. Fig. 18 illustrates the detail view of each multiple-attributes query performance.

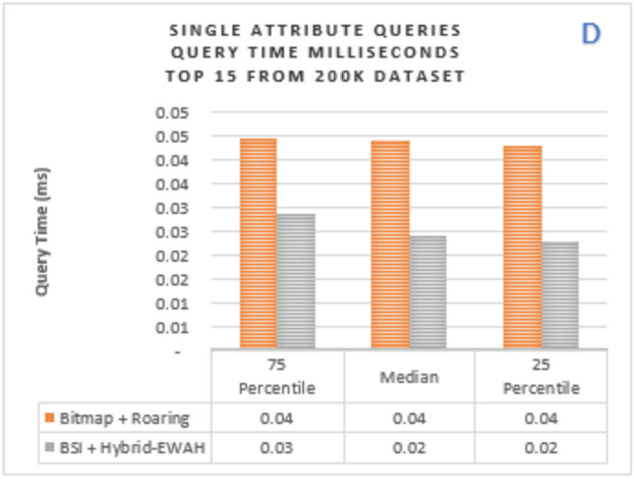
8. Multidimensional grouping experimental evaluation

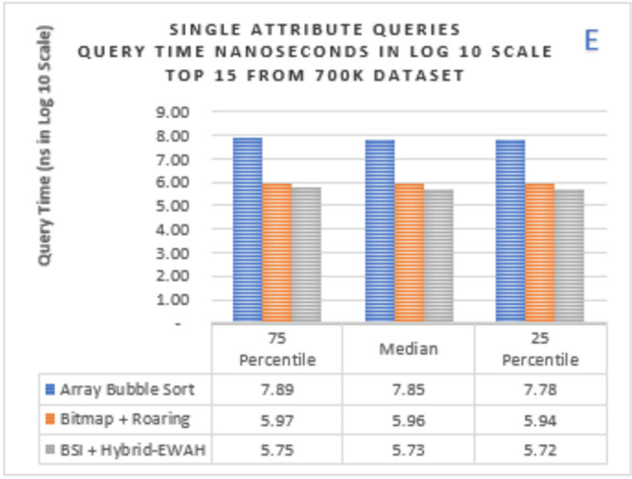
In this section, we present the results of the evaluation of two grouping approaches: BSI and Java Stream API.











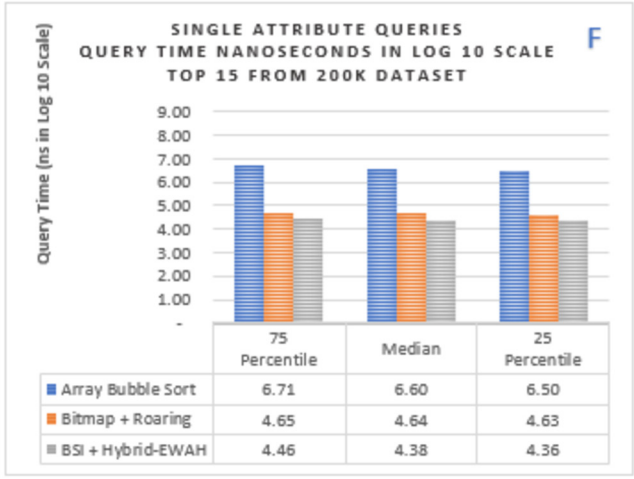


Fig. 16. 700K single attribute top-k query. The figure shows 25, 50, 75 percentiles of top-k query performance. Parameters: n=700K; k =15; Number of single-attribute queries measured: 75.

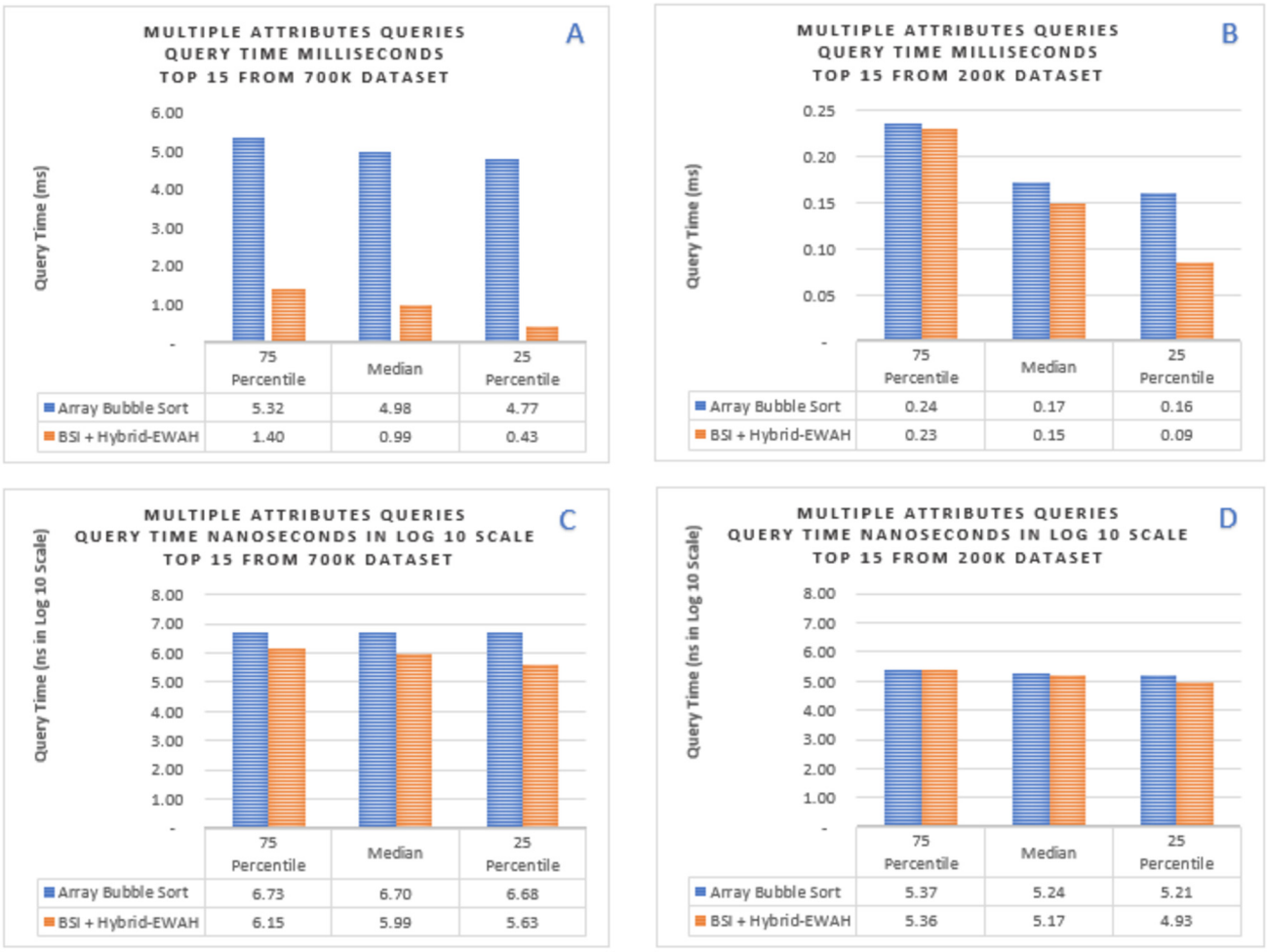


Fig. 17. The figure shows the top-k query performance measurements of 75 two-attributes combinations on a 700K dataset. Measurement unit: nanoseconds. Parameters: n=700K; k =15.

8.1. Experimental setup

We run the experiments on the environment with following configuration.

- OS: Linux CentOS
- Number of Processors: 56
- Processor: Intel(R) Xeon(R) CPU E5-2660 v4 @ 2GHz
- RAM: 256 GB
- System Type: 64-bit Operating System, x64-based CPU
- Implementation: Java JDK 1.8 version.

Table 9 shows the detail information of the process in the experiment environment.

Table 9
Processor Detail of Grouping Experiment Environment.

Feature:	Specification
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	56
On-line CPU(s) list:	0-55
Thread(s) per core:	2
Core(s) per socket:	14
Socket(s):	2
NUMA node(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	79
Model name:	Intel(R) CPU 2.00GHz
Stepping:	1
CPU MHz:	1200.073
CPU max MHz:	3200.0000
CPU min MHz:	1200.0000
BogoMIPS:	3999.84
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	35840K

8.2. Dataset

We use the dataset [20] in the grouping experiments. We perform the experiments from 1 to 9 grouping dimensions. Each experiment starts with the temporal dimension and the unique object id, and then adds object properties related dimensions. In each of the measures, we aggregate every 75 attributes with grouping dimensions. Table 10 shows the grouping attributes.

8.3. Performance measurement: multidimensional grouping

In the experiments, we run several dimensions ranging from 1 to 9 for all 75 metrics in a 700K dataset. Table 11 lists the result in detail. It reveals that the BSI grouping approach is about 10 to 40 times faster than the Java Stream approach.

In Table 11, the number of dimensions is the number of grouping attributes. The measurement unit is millisecond. The "BSI

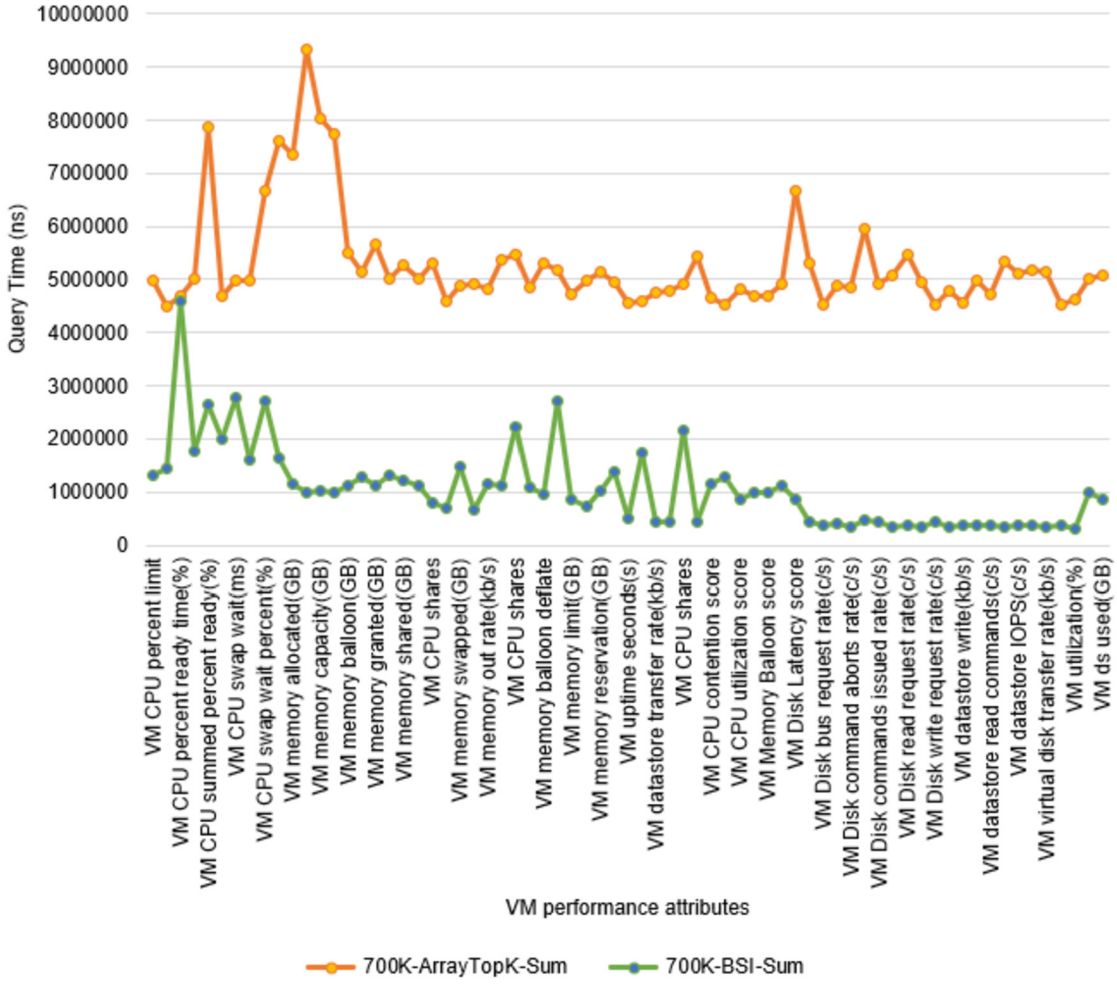


Fig. 18. The figure shows a detail view of the top-k query performance measurements of 75 two-attributes combinations on a 700K dataset. Measurement unit: nanoseconds. Parameters: n=700K; k =15.

Table 10
Grouping-by Attributes.

Group-by Attributes	Cardinality
Day of the month	30
VM Id	33
VM CPU cores	14
VM memory allocated(GB)	33
VM memory capacity(GB)	37
VM memory reservation(GB)	13
Datastore name	12
VM ds allocated(GB)	209
VM CPU totalHz(MHz)	262

Table 11
Grouping-by Performance Measurement.

Dimensions	Stream API (ms)	BSI (ms)	Times Faster
1	341.98	7.09	47.21
2	482.48	48.47	8.95
3	628.08	49.43	11.71
4	625.03	52.61	10.88
5	650.86	51.35	11.68
6	726.70	50.48	13.39
7	759.08	55.34	12.72
8	797.39	56.39	13.14
9	846.08	59.91	13.12

Times Faster" column shows how many times faster comparing the BSI approach to the stream API approach.

Fig. 19 shows the average performance comparison between BSI and Java Stream API for 1 to 9 grouping dimensions. The orange dotted line shows the linear regression forecast for Java Stream API performance. The linear forecast formula is shown as Equation (13).

$$y = 55.412x + 373.79 \tag{13}$$

The R squared is 0.9121. The dark gray dotted line shows the linear forecast for BSI grouping performance. The linear forecast formula of BSI grouping performance is shown as Equation (14).

$$y = 4.0784x + 27.506 \tag{14}$$

The R squared is 0.504. Based on the linear forecasting Equation (13) and Equation (14), when the number of dimensions increases, both the BSI approach and the Java Stream API approach consume more time. The BSI approach time growth rate is two magnitudes smaller than the Java Stream API approach time growth rate.

8.4. Performance measurement: one dimension

The first dimension used in the experiment is "date." On the single dimension experiment, the BSI approach on average is 47 times faster than the Java Stream API approach. Fig. 20 shows the detailed view of each individual attribute group-by performance for BSI and Java 8 default stream function.

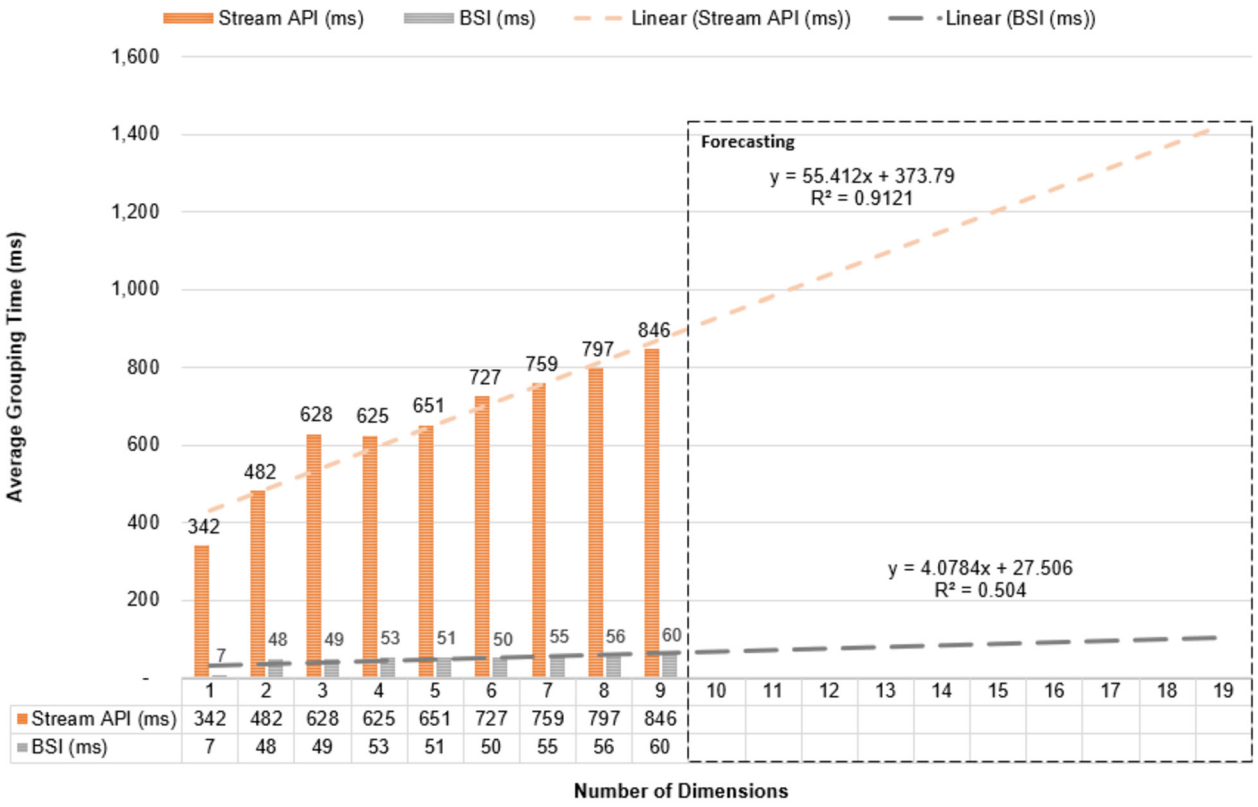


Fig. 19. Multidimensional grouping performance comparison. 700K dataset 1 to 9 dimensions grouping BSI vs. Java 8 stream API. Query time is measured as milliseconds. X axis is the number of dimensions. Y axis is the performance measurement in millisecond unit.

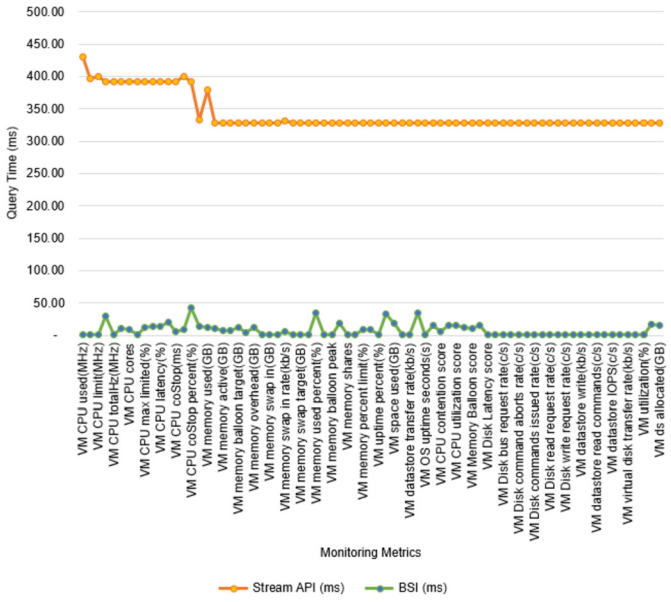


Fig. 20. The figure shows the performance on a 700K dataset for single dimension grouping top-k query of the BSI approach vs. the Java Stream API approach. The query time measurement unit is millisecond.

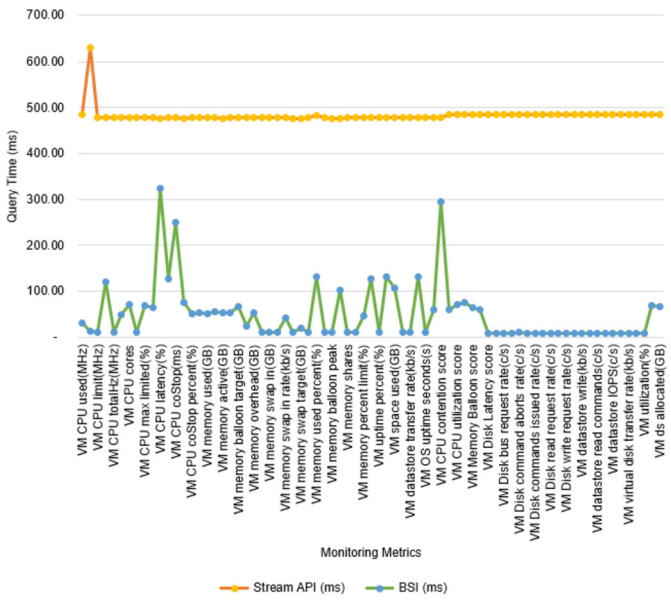


Fig. 21. The figure shows the performance on a 700K dataset for two dimensions grouping top-k query of the BSI approach vs. the Java Stream API approach. The query time measurement unit is millisecond.

8.5. Performance measurement: two dimensions

The first two dimensions used in the experiment are “date” and “virtual machine ID.” On the two dimensions experiment, BSI is about 8 times faster than the Java Stream API approach. Fig. 21 shows the detailed view of each individual attribute group-by performance for BSI and Java 8 stream API.

8.6. Performance measurement: three to nine dimensions

When the number of dimensions goes up to 3, 4, 5,....., and 9, the BSI performance becomes consistently 10 12 times faster than the Java Stream API approach. Fig. 22 presents the detail view of three dimensions grouping top-k query performance for BSI and Java Stream API.

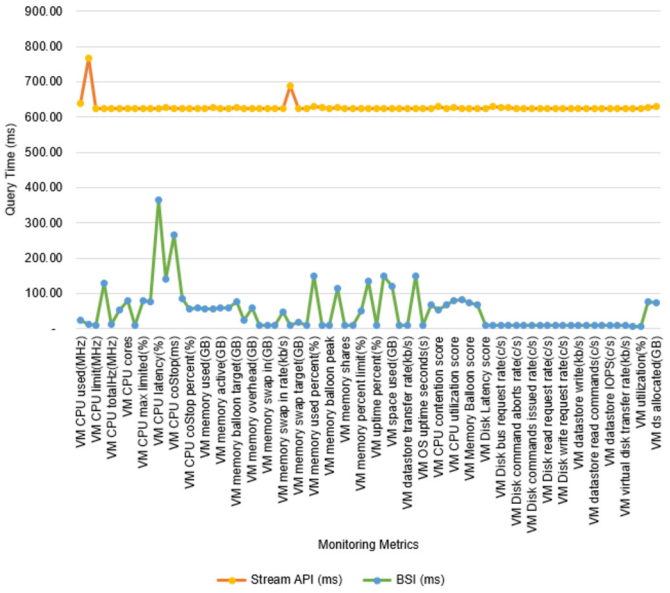


Fig. 22. The figure shows the performance on a 700K dataset for three dimensions grouping top-k query of the BSI approach vs. the Java Stream API approach. The query time measurement unit is millisecond.

9. Future work

As future work, we plan on evaluating anomaly detection algorithms using the BSI data structures. In the infrastructure performance monitoring systems, anomaly detection (also outlier detection) is the identification of rare items, events, or observations, which raises suspicions by differing significantly from the majority of the infrastructure data.

Typically the abnormal items will translate to some problems such as system attack, performance problems, or errors in the text. Anomalies are also referred to as outliers novelties, noise, deviations, and exceptions. To find out the outliers, we usually generate a baseline range for comparison. A manually defined baseline range is a static threshold. A machine learning baseline creating from historical data is a dynamic threshold. Generating dynamic thresholds based on historical data is critical for artificial intelligent anomaly detection. However, for large scale data in a real-time monitoring system, dynamic thresholds will need to be generated on every incoming data sample with multiple scenarios consideration. Event, metrics, property, and time are various dimensions categories that need to be considered.

In a real-time monitoring system, anomaly detection has the following scenarios:

1. Detecting the incoming data bucket, which has abnormal data sample event-count comparing previous data buckets. It could detect the number of failed backup jobs in a time window, which is different from last time windows.

2. Detecting the incoming data bucket, which contains anomalous geographic property information. For example, the transaction location is different from where it happened before. It could detect a manual vMotion which moves a VM from one cluster to another cluster at different geographic locations. It could be used for any transactions having geographic information.

3. Detecting the incoming data bucket, which has an anomalous metric. A metric contains the value of min, max, mean, median, and standard deviation. We use single or multiple values comparison of the incoming data and previous data buckets for anomaly detection.

4. Detecting the incoming data bucket, which has an anomalous metric sum. The sum will need to be handled separately. We need to allow the user to choose to consider the null value as average or as zero during a sum.

5. Detecting the incoming data bucket, which has an anomalous metric time pattern. A metric time pattern contains the value of min, max, mean, median, and standard deviation for time-of-day and time-of-week. The abnormally could be based on single or multiple values comparison of time-of-day and time-of-week buckets.

6. Detecting the incoming event has an abnormal event time pattern. For example, a type of event usually only happens in a certain time pattern, but it abnormally happens outside of that pattern. While a new job creation usually happens during office hours, but it is created outside of the office hours now.

For each of those scenarios, there would be a set of baseline need to be generated for anomaly detection.

The majority of those detection functionalities require dynamic grouping and summation operations. Given the results obtained in this work with the bit-sliced data structures and associated algorithms on those grouping and summation operations, we will explore further the performance gain on using BSI in anomaly detection applications.

10. Conclusions

In this project, we evaluated the usage of bit-slicing and bitmaps (binning) to determine ranking queries with crossing attributes filtering in performance monitoring systems.

For a single attribute query, the bit-slicing approach is faster, with a performance gain on both large and small data set up to 170 times faster than the baseline approach array-bubble-sort algorithm, and 2 times faster than bitmap approach. Moreover, by using the BSI filtering algorithm introduced in this paper, the BSI algorithm is 3 times faster than the binning bitmap approach in the use case of crossing attribute filtering.

For a multiple attributes query, there is no applicable bitmap approach to compare with the BSI approach. Therefore, the experiment of multiple attributes query is performed with BSI and array-bubble-sort. Overall, the BSI approach is faster, with a performance gain up to one order of magnitude faster than the baseline approach array-bubble-sort algorithm.

For the multidimensional grouping, on the 1 to 9 number of dimension evaluations, the BSI approach is consistently two orders of magnitude faster than the baseline approach Java Stream API. Meanwhile, based on the linear regression forecasting, the BSI formula slope is two magnitudes smaller than the Java Stream API formula slope. That implies that the BSI approach will also outperform the Java Stream approach in the number of dimensions over 9.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] R. Akbarinia, E. Pacitti, P. Valduriez, Best position algorithms for top-k queries, in: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB Endowment, 2007, pp. 495–506.

[2] R. Akbarinia, E. Pacitti, P. Valduriez, Best position algorithms for efficient top-k query processing, Inf. Sci. 36 (2011) 973–989, https://doi.org/10.1016/j.is.2011.03.010.

[3] W.T. Balke, U. Güntzer, Multi-objective query processing for database systems, in: Proceedings of the 30th International Conference on Very Large Data Bases, VLDB Endowment, 2004, pp. 936–947.

[4] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, G. Weikum, Io-top-k: index-access optimized top-k query processing, in: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB Endowment, 2006, pp. 475–486.
[5] N. Bruno, L. Gravano, A. Marian, Evaluating top-k queries over web-accessible databases, in: Proceedings 18th International Conference on Data Engineering, IEEE, 2002, pp. 369–380.
[6] S. Chambi, D. Lemire, O. Kaser, R. Godin, Better bitmap performance with roaring bitmaps, Softw. Pract. Exp. 46 (2016) 709–719.
[7] K.C.C. Chang, S.w. Hwang, Minimal probing: supporting expensive predicates for top-k queries, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, ACM, 2002, pp. 346–357.
[8] A. Colantonio, R. Di Pietro, Concise: compressed 'n' composable integer set, Inf. Process. Lett. 110 (2010) 644–650.
[9] R. Fagin, R. Kumar, D. Sivakumar, Comparing top k lists, in: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Association for Computing Machinery, 2003, pp. 28–36, https://doi.org/10.5555/644108.644113.
[10] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, J. Comput. Syst. Sci. 66 (2003) 614–656.
[11] S. Fay, J. Xie, Probabilistic goods: a creative way of selling products and services, Mark. Sci. 27 (2008) 674–690.
[12] P. Gursky, T. Horvath, R. Novotny, V. Vaneková, P. Vojtas, Upre: user preference based search system, in: 2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2006 Main Conference Proceedings) (WI'06), IEEE, 2006, pp. 841–844.
[13] G. Guzun, G. Canahuate, Hybrid query optimization for hard-to-compress bit-vectors, VLDB J. 25 (2016) 339–354, https://doi.org/10.1007/s00778-015-0419-9.
[14] G. Guzun, G. Canahuate, Performance evaluation of word-aligned compression methods for bitmap indices, Knowl. Inf. Syst. 48 (2016) 277–304.
[15] G. Guzun, G. Canahuate, High-dimensional similarity searches using query driven dynamic quantization and distributed indexing, Distrib. Parallel Databases 38 (2020), https://doi.org/10.1007/s10619-019-07266-x.
[16] G. Guzun, G. Canahuate, D. Chiu, A two-phase mapreduce algorithm for scalable preference queries over high-dimensional data, in: Proceedings of the 20th International Database Engineering & Applications Symposium, ACM, 2016, pp. 43–52.
[17] G. Guzun, G. Canahuate, D. Chiu, J. Sawin, A tunable compression framework for bitmap indices, in: 2014 IEEE 30th International Conference on Data Engineering, IEEE, 2014, pp. 484–495.
[18] G. Guzun, J. Tosado, G. Canahuate, Slicing the dimensionality: top-k query processing for high-dimensional spaces, in: Transactions on Large-Scale Data-and Knowledge-Centered Systems XIV, Springer, 2014, pp. 26–50.
[19] I.F. Ilyas, G. Beskales, M.A. Soliman, A survey of top-k query processing techniques in relational database systems, ACM Comput. Surv. 40 (2008) 11.
[20] Q.S. Inc, Hybrid cloud data center 700k dataset [online], https://github.com/Foglight/BSI/, 2019. (Accessed 10 January 2019).
[21] D. Lemire, O. Kaser, K. Aouiche, Sorting improves word-aligned bitmap indexes, Data Knowl. Eng. 69 (2010) 3–28.
[22] D. Lemire, G. Ssi-Yan-Kai, O. Kaser, Consistently faster and smaller compressed bitmaps with roaring, Softw. Pract. Exp. 46 (2016) 1547–1569.
[23] P. O'Neil, D. Quass, Improved query performance with variant indexes, ACM SIGMOD Rec. 26 (1997) 38–49.
[24] Y. Qin, H. Fahimi Chahestani, Z. Song, Systems and methods for integrated modeling and performance measurements of monitored virtual desktop infrastructure systems, US Patent App. 15/201,657, 2019.
[25] Y. Qin, Z. Song, Z.H. Ji, Systems and methods for integrated modeling of monitored virtual desktop infrastructure systems, US Patent App. 10/200,252, 2019.
[26] D. Rinfret, P. O'Neil, E. O'Neil, Bit-sliced index arithmetic, in: ACM Sigmod Record, ACM, 2001, pp. 47–57.
[27] J.E. Tosado, G. Guzun, G. Canahuate, R. Mantilla, On-demand aggregation of gridded data over user-specified spatio-temporal domains, in: Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, ACM, 2016, p. 62.
[28] K. Wu, W. Koegler, J. Chen, A. Shoshani, Using bitmap index for interactive exploration of large datasets, in: 15th International Conference on Scientific and Statistical Database Management, IEEE, 2003, pp. 65–74.
[29] M.C. Wu, A.P. Buchmann, Encoded bitmap indexing for data warehouses, in: Proceedings 14th International Conference on Data Engineering, IEEE, 1998, pp. 220–230.
[30] G. Xiao, K. Li, K. Li, Reporting l most influential objects in uncertain databases based on probabilistic reverse top-k queries, Inf. Sci. 405 (2017) 207–226, https://doi.org/10.1016/j.ins.2017.04.028, https://www.sciencedirect.com/science/article/pii/S0020025517306746.
[31] G. Xiao, K. Li, X. Zhou, K. Li, Efficient monochromatic and bichromatic probabilistic reverse top-k query processing for uncertain big data, J. Comput. Syst. Sci. 89 (2017) 92–113, https://doi.org/10.1016/j.jcss.2016.05.010, https://www.sciencedirect.com/science/article/pii/S0022000016300459.
[32] A. Yu, P.K. Agarwal, J. Yang, Top-*k* preferences in high dimensions, IEEE Trans. Knowl. Data Eng. 28 (2016) 311–325.
[33] W. Zhang, X. Lin, Y. Zhang, W. Wang, J.X. Yu, Probabilistic skyline operator over sliding windows, in: 2009 IEEE 25th International Conference on Data Engineering, 2009, pp. 1060–1071, https://doi.org/10.1109/ICDE.2009.83.
[34] X. Zhou, K. Li, G. Xiao, Y. Zhou, K. Li, Top *k* favorite probabilistic products queries, IEEE Trans. Knowl. Data Eng. 28 (2016) 2808–2821, https://doi.org/10.1109/TKDE.2016.2584606.