Big Data Research ••• (••••) •••–•••

Contents lists available at ScienceDirect

Big Data Research

www.elsevier.com/locate/bdr

Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud ☆

Tian Guo [a,*], Thanasis G. Papaioannou [b], Karl Aberer [a]

[a] EPFL, Switzerland
[b] Center for Research & Technology Hellas (CERTH), Greece

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
Big data
Index
Key-value stores
MapReduce
Approximation
Query optimization

ABSTRACT

As the number of sensors that pervade our lives increases (e.g., environmental sensors, phone sensors, etc.), the efficient management of massive amount of sensor data is becoming increasingly important. The infinite nature of sensor data poses a serious challenge for query processing even in a cloud infrastructure. Traditional raw sensor data management systems based on relational databases lack scalability to accommodate large-scale sensor data efficiently. Thus, distributed key-value stores in the cloud are becoming a prime tool to manage sensor data. Model-view sensor data management, which stores the sensor data in the form of modeled segments, brings the additional advantages of data compression and value interpolation. However, currently there are no techniques for indexing and/or query optimization of the model-view sensor data in the cloud; full table scan is needed for query processing in the worst case. In this paper, we propose an innovative index for modeled segments in key-value stores, namely KVI-index. KVI-index consists of two interval indices on the time and sensor value dimensions respectively, each of which has an in-memory search tree and a secondary list materialized in the key-value store. Then, we introduce a KVI-index–Scan–MapReduce hybrid approach to perform efficient query processing upon modeled data streams. As proved by a series of experiments at a private cloud infrastructure, our approach outperforms in query-response time and index-updating efficiency both Hadoop-based parallel processing of the raw sensor data and multiple alternative indexing approaches of model-view data.

1. Introduction

Recent advances in sensor technology have enabled the vast deployment of sensors embedded in user devices that monitor various phenomena for different applications of interest, e.g., air/electrosmog pollution, radiation, early earthquake detection, soil moisture, permafrost melting, etc. The data streams generated by a large number of sensors are represented as time series in which each data point is associated with a time-stamp and a sensor value. These raw discrete observations are taken as the first citizen in traditional relational sensor data management systems, which leads to a number of problems. On one hand, in order to perform analysis of the raw sensor data, users usually adopt other third-party modeling tools (e.g., Matlab, R and Mathematica), which involve of tedious and time-consuming data extract, transform and load (ETL) processes [1]. Moreover, such data analysis tools are usually used for static data set and therefore cannot be applied for online processing of sensor data streams. On the other hand, unbounded sensor data streams often have missing values and unknown errors, which also poses great challenges for traditional raw sensor data management.

To this end, various model-based sensor data management techniques [1–4] have been proposed. Model-view sensor data management leverage time series approximation and modeling techniques to segment the continuous sensor time series into disjoint intervals such that each interval can be approximated by a kind of model, such as polynomial, linear or SVD. These models, for all the intervals (or segment models), exploit the inherent correlations (e.g. with time or among data streams) in the segments of sensor time series and approximate each segment by a certain mathematical function within a certain error bound [5–9]. Then, one can only materialize the models of the segments instead of the raw data and harvest a number of benefit:

First, model-view sensor data achieves compression over raw sensor data and therefore requires less storage overhead [10–12]. Second, due to the sampling frequency or sensor malfunction,

☆ This article belongs to Scalable Computing for Big Data.

* Corresponding author.
E-mail addresses: tian.guo@epfl.ch (T. Guo), thanasis.papaioannou@iti.gr (T.G. Papaioannou), karl.aberer@epfl.ch (K. Aberer).

http://dx.doi.org/10.1016/j.bdr.2014.07.005
2214-5796/

there may be some missing values at some time points. If one query involves such time points, then the relevant segment model can be used to estimate the values [10–12]. In some degree, model-view sensor data increases the data availability for query processing. Third, there usually exist outliers in raw sensor data, which has negative effect on the query results. Model-view sensor data removes the outliers in each interval via the segment model and historical data on upper and lower data bounds, thereby diminishing the effect of outliers in query results. Fourth, regarding the similarity search or pattern discovery in sensor time-series mining, the segment-based time series representation is a powerful tool for dimension reduction and search space pruning [9,13,14].

However, proposed model-based sensor data management approaches mostly employ the relational data model and process queries based on materialized views [2,3] on top of the modeled segments of sensor data. Nowadays, the amount of data produced by sensors is exponentially increasing. Moreover, the real-time production of sensor data requires the data management system to be able to handle high-concurrent model-view sensor data from massive sensors and this is difficult for traditional relational database to realize. To this end, recent prevalent cloud store and computing techniques provide a promising way to manage model-view sensor data [15–19].

The main focus of this paper is on how to manage the segment models of sensor data, namely model-view sensor data with the newly emerging cloud stores and computing techniques rather than how to explore more advanced sensor data segmentation algorithms.

In our approach, we exploit key-value stores and the MapReduce parallel computing paradigm [18,20], two significant aspects of cloud computing, to realize indexing and querying model-view sensor data in the cloud. We characterize the modeled segments of sensor time series by the time and the value intervals of each segment [16,17]. Consequently, in order to process range or point queries on model-view sensor data, our index in the cloud store should excel in processing interval data. Current key-value built-in indices do not support interval-related operations. The interval index for model-view sensor data should not only work for static data, but it should be dynamically updated based on the new arriving segments of sensor data. If traditional batch-updating or periodical re-building strategy was applied here [21,22], then the high speed of sensor data yielding might lead to a large size of the new unindexed data, even in short time periods and to significant index updating delay as well. As a result, the performance of queries involving both indexed and unindexed data would degenerate greatly. Therefore, the interval index in the cloud store should be able to insert an individual new modeled segment in an online manner.

The contributions of this paper are summarized as follows:

- Innovative interval index: We propose an innovative interval index for model-view sensor data management in key-value stores, referred to as *KVI-index*. *KVI-index* is a two-tier structure consisting of one lightweight and memory-resident binary search tree and one index-model table materialized in the key-value store. This composite index structure can dynamically accommodate new sensor data segments very efficiently.
- Hybrid model-view query processing: After exploring the search operations in the in-memory structure of the KVI-index for range and point queries, we introduce a hybrid query processing approach that integrates range scan and MapReduce to process the segments in parallel.
- Intersection search: We introduce an enhanced intersection search algorithm (*iSearch+*) that produces consecutive results suitable for MapReduce processing. We theoretically analyze the efficiency of (*iSearch+*) and find the bound on the redundant index nodes that it returns.
- Experimental evaluation: Our framework has been fully implemented, including the online sensor data segmentation, modeling, KVI-index and the hybrid query processing, and it has been thoroughly evaluated against a significant number of alternative approaches. As experimentally shown based on real sensor data, our approach significantly outperforms in terms of query response time and index updating efficiency all other ones for answering time/value point and range queries.

The remainder of this paper is organized as follows: Section 2 summarizes some related work on model-view sensor data management, interval index and index-based MapReduce optimization approaches. In Section 3, we provide a brief description of sensor-data segmentation, querying model-view sensor data and the necessity to develop interval index for managing model-view sensor data in key-value stores. The detailed designs of our innovative KVI-index and the hybrid query processing approach are discussed in Sections 4 and 5 respectively. Then, in Section 6, we present thorough experimental results to evaluate our approach with traditional query processing ones on both raw sensor data and modeled data segments. Finally, in Section 7, we conclude our work.

2. Related work

Time series segmentation is an important research problem in the areas of data approximation, data indexing and data mining. A lot of work has been devoted to exploit different types of models to approximate the segments of time series, such that the pruning and refinement framework can be applied to this segment-represented time series for the pattern matching or similarity search [9,13]. Some other researchers proposed techniques for managing the segment models that approximate sensor time series in relational databases. MauveDB designed a model-based view to abstract underlying raw sensor data; it then used models to project the raw sensor readings onto grids for query processing [3]. As opposed to MauveDB, FunctionDB only materializes segment models of raw sensor data [2]. Symbolic operators are designed to process queries using models rather than raw sensor data. However, both approaches in [3] and [2] do not take into account applying indices to improve query processing. Moreover, their proposed approach focuses on managing static dataset of time series rather than dynamic time series.

Also, each segment of time series could be characterized by its time and value intervals. Then, one should consider employing an interval index for processing queries over model-view sensor data. Two common used indices for interval data are segment tree [21] and interval tree [23]. As for segment tree, it is essentially a static data structure that needs an initialization process to construct elementary intervals based on the initial dataset [21]. Once a new interval outside of the domain of current segment tree arrives, the elementary intervals should be rebuilt, which is not suitable for the real time nature of sensor data streams [21]. Regarding the interval tree, individual interval is not divided and replicated during the insertion phase as in the segment tree and therefore the storage overhead is linear to the number of intervals to index [23–25]. However, it is a memory-oriented structure.

Some efforts [22,23,26,27] have also been done to externalize these in-memory index data structures. The relational interval tree (RI-tree) [26] integrates interval tree into relational tables and transforms interval queries into SQL queries on tables. This method makes efficient use of built-in B+-tree indices of RDBMS. Nevertheless, in this paper, we aim to design an interval index structure for model-view sensor data that is compatible with key-value stores and distributed query processing in the cloud.

Please cite this article in press as: T. Guo et al., Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud, Big Data Research (2014), http://dx.doi.org/10.1016/j.bdr.2014.07.005

In [28], they proposed an approach that enables key-value stores to support query processing of multi-dimensional data via integrating space filing order into row-keys, which is not fit for interval data. An index for multi-dimensional point data is proposed in [29] based on P2P overlay network, which is a different underlying architecture from our adopted key-value store. The latest effort to develop interval indices in the cloud utilizes MapReduce to construct a segment tree materialized in the key-value store [22]. This approach outperforms the interval query processing provided by HBase (http://hbase.apache.org/) and Hive (http://hive.apache.org/). However, the segment tree utilized in [22] is essentially a static index, as is also the case with [21]. Therefore, an index rebuilding phase needs to be periodically executed to include new data.

MapReduce parallel computing is an effective tool to process large scale of sensor data in cloud stores, but conventional MapReduce always conducts trivial full scan on the whole data set for the queries of any selectivity [19]. In order to enable MapReduce and indices to collaborate for query processing, many researchers proposed to integrate index techniques into MapReduce framework to avoid full data scan for low-selective queries [30–34]. One prior work is to construct a B+-tree over static data set in a distributed file system (e.g. HDFS) and materialize the index also in HDFS [34]. Then, the query processor can process this index file, which involves of multiple MapReduce jobs to access different layers of the tree and therefore is not efficient enough [34]. Moreover, the proposed HDFS based B+-tree is only effective for point data rather than interval data [34].

The authors in [31] integrate indices into each file split of the data file in HDFS, such that mappers can use the indices to only access predicate-qualified data records in each split. In [32], indices applicable to queries with predicates on multiple attributes were developed by indexing different record attributes in different block replicas of data files. These two kinds of index access methods in MapReduce are both on record-level. If we integrate an interval index into each file split guided by this direction, the MapReduce data preparation, starting overhead and Mapper waves are actually not decreased much and therefore the performance does not significantly improve. On the other hand, the update in one file split requires to rebuilt the whole local index of the split thereby having high latency for segment insertions. In [33], a split-level index is designed to decrease MapReduce framework overhead. As compared to above record-level indices in [31, 32], split-level indices eliminate irrelevant file splits before launching mappers, and the data transferring and starting up overheads are further decreased. This kind of approach is more efficient for decreasing the map function invocations via overriding the data reader of MapReduce [31,32], as compared to record-level optimizations.

3. Overview of model-view sensor data management

In this section, we discuss certain important issues for managing model-view sensor data in a key-value store. First, we explain what is model-view sensor data. Then, we discuss possible storage schemas for model-view sensor data and explain the necessity to develop a special index for it in key-value stores. Last, we describe the query types of our focus and some particular techniques for processing model-view sensor data queries.

3.1. Sensor time series segmentation

Sensor time series segmentation is a type of algorithms that fragment a time series stream into disjoint segments, and then approximate each data segment by a mathematical function or model, such that a specific error bound is satisfied [5–9]. Query processing can then be performed on the materialized segments instead of the raw sensor data, as in [2]. Sensor data segmentation and modeling algorithms have been extensively studied in [4–9].

Algorithm 1: Sensor time series segmentation.

```
Input: v_t, t, /* value and associated time point of one sensor reading */
  error_bound
1 begin
2   if currentSeg_error(anchor, t) < error_bound then
3     seg_[anchor,t] = segmentModel(v_anchor, ..., v_t);
4     /* approximate the sensor values between the time range [anchor, t]
        with the specified type of model (e.g., constant, linear, polynomial,
        etc.) */
5   else
6     l_t = anchor;
7     l_r = t − 1;
8     SegMaterialization(seg_[anchor,t])
9     /* output this new produced segment seg_[anchor,t] for materialization
        */
10    anchor = t;
```

Table 1
Symbol list.

Symbol	Semantics
l_t	Beginning timestamp of one segment model
r_t	Ending timestamp of one segment model
l_v	Minimum value of one segment model's value range
r_v	Maximum value of one segment model's value range
$p_0^i \cdots p_n^i$	Coefficient of each item in polynomial model

As segmentation algorithms are not the focus of this paper, we only present a general framework in Algorithm 1 for online time series segmentation [7], which is used in the dataset preparation phase of our sub-sequential experimental evaluation in Section 6. Table 1 summarizes the list of symbols that we employ for model segments.

This sensor time series segmentation program is invoked each time a new sensor reading comes. It first checks if the segment model over the sensor readings from the anchor time point to current time, namely in range $[anchor, t]$, satisfies the error bound. If yes, the current segment model is updated to $seg_{[anchor,t]}$. If no, the current time t is supposed to be the starting point *anchor* of a new segment. Then, it outputs the segment $seg_{[anchor,t-1]}$ for the materialization process for which our designed index approach is responsible. The segment $seg_{[anchor,t-1]}$ consists of the following information: time interval $[l_t, r_t]$, value range $[l_v, r_v]$ and model formula (for instance, it could be a sequence of coefficients for each item of one polynomial function). Under this framework, users can choose different categories of models to approximate the sensor data in segments. For instance, a time series shown in Fig. 1(a) is transformed into a sequence of linear functions with time dimension as the independent variable, which is depicted in Fig. 1(b). The associated value range in Fig. 1(b) indicates the values one segment model can cover within the time interval. Such set of functions, each with its associated $[l_t, r_t]$ and $[l_v, r_v]$ for the sensor data segments are referred to as the model-view sensor data over the original raw sensor data observations.

3.2. Storage model in key-value stores

3.2.1. HBase overview

A table in HBase consists of a set of regions, each of which stores a sequential range partition of the row-key space of the table [18,20]. An HBase cluster consists of a collection of servers, called region servers, which are responsible for serving a subset of regions of one table. The key-based data access method of HBase is realized by a three layered B+ tree maintained among the nodes

Please cite this article in press as: T. Guo et al., Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud, Big Data Research (2014), http://dx.doi.org/10.1016/j.bdr.2014.07.005

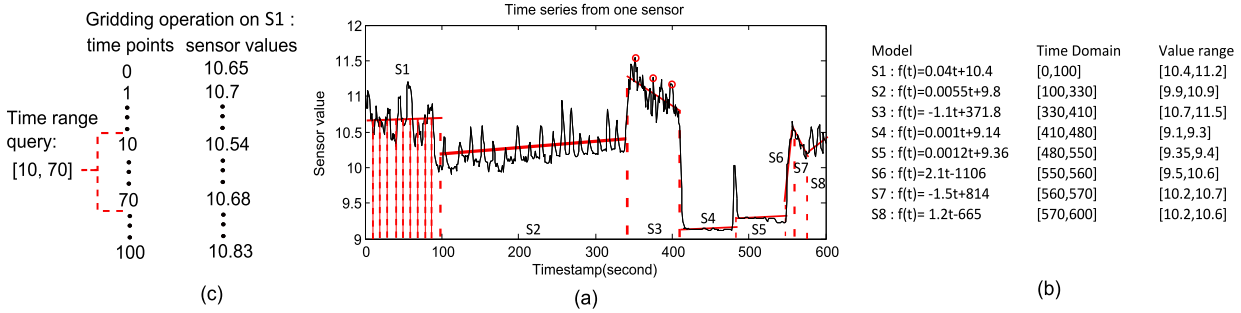


Fig. 1. Model-view sensor data.

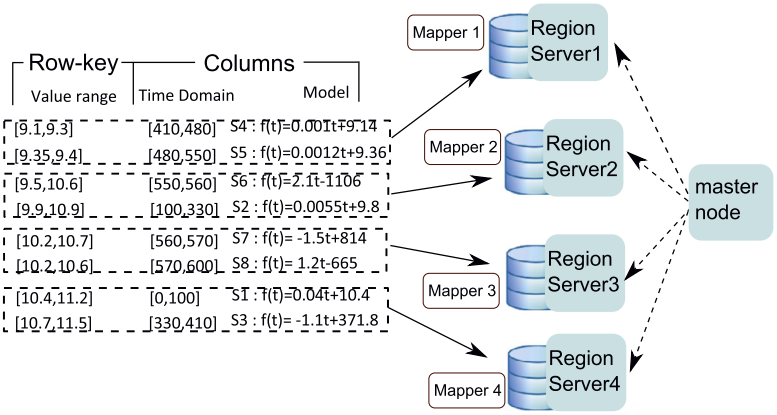


Fig. 2. Model-view sensor data in HBase key value store.

of HBase cluster. The two upper levels referred to as the ROOT and META regions are kept in the master node of an HBase cluster. The ROOT region plays the role of root node of one B+ tree, while the META table maintains a mapping of the regions of a table to region servers. The regions distributed among the other nodes constitute the lowest level of the B+-tree index and store the real data of a table. If a region's size exceeds a configurable limit, HBase can dynamically split it into two sub-regions, which enables the system to grow dynamically as data is appended. Regarding a MapReduce job on a table in HBase, the number of mappers is equal by default to that of regions of the table to process, which means that each mapper processes one region, as shown in Fig. 2. The number of reducers is configurable programmatically. And HBase allows MapReduce to only process key-based table partitions of interest, which can avoid trivial full-scan on a table for low-selective queries [18,20].

3.2.2. Interval-index in HBase

For the model-view sensor data in the key-value store, the time interval, the value range and the model formula (e.g. polynomial coefficients) can fully approximate one data segment. There are different possible ways to organize row keys and columns for storing and querying model-view sensor data.

One idea for storing segments in key-value stores could be to do it similarly to that of the raw sensor data management system such as Open-TSDB [15], which takes the time interval of one segment as the row key (rk). As rows are sorted on the row key in the key-value store, the starting points of time intervals are in ascending order. Therefore, although for time range or point queries the query processor knows when to stop the scan, it still needs to start the scan from the beginning of the table each time. The same problem happens to the table with value intervals as row-keys. For instance, in Fig. 2, when the value range is the row-key, the table is sorted according to the left boundaries of the value ranges of model-view sensor data. For a value query range $[9.2, 10]$, we can make sure after the fourth segment, namely $S2$ with value range $[9.9, 10.9]$, there is no qualified segment in the table. However, still scanning has to start from the beginning of the table, since the right boundaries of the value ranges are not in order.

In summary, simply incorporating time, value interval or model coefficients into the row-key cannot contribute to accelerating the query processing. A generic index in key-value stores specialized for model-view sensor data is imperative. Moreover, considering that model-view sensor data is produced in real-time, the index for model-view sensor data in key-value stores should be able to update on the fly efficiently [20]. Since each segment of model-view sensor data is characterized by its time and value ranges, we will concentrate on employing interval index to index both the time and value dimension of one segment.

However, another issue for designing the interval index for model-view sensor data in key-value stores is where to materialize the index. One possible way would be to store the index in separated files in the distributed file system (e.g. HDFS, etc.) on top of which the key-value store HBase is also built. In this approach, as in the work [34], each layer of the tree structure is stored in one file. Consequently, index searching needs to invoke multiple MapReduce jobs to iteratively process each level of the tree, which is not efficient. Another way is to integrate one index for each region to only index the local model-view sensor data in that region, such that mappers can first load the index to locate desired data in that region and then access it. However, this kind of approach does not decrease data preparation and starting overhead of MapReduce for the whole table, since still all the regions of the table are initialized to be processed by MapReduce [31,32]. Our proposed key-value based interval index (described in the next section) steps out of above ideas and takes advantage of both the in-memory and key-value parts to improve the query processing.

Please cite this article in press as: T. Guo et al., Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud, Big Data Research (2014), http://dx.doi.org/10.1016/j.bdr.2014.07.005

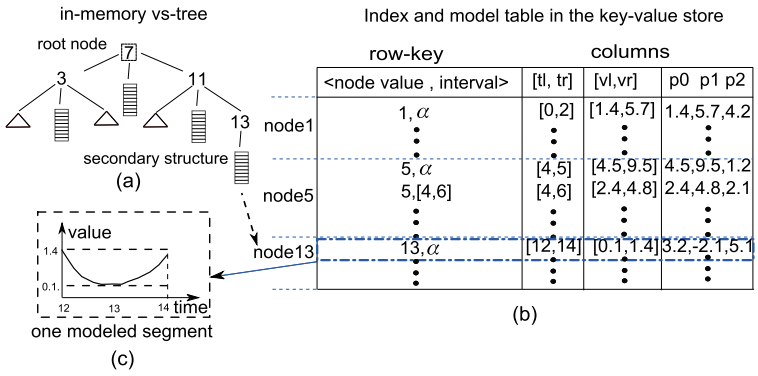


Fig. 3. Two-tier structure of KVI-index.

3.3. Querying model-view sensor data

Much work has been devoted to pattern discovery for time series data, but in this paper, we focus on the following four types of fundamental queries on model-view sensor data.

- Time point query: return the value of one sensor at a specific time point.
- Value point query: return the timestamps when the value of one sensor is equal to the query value. There may be multiple time stamps of which sensor values satisfy the query value.
- Time range query: return the values of one sensor during the query time range.
- Value range query: return the time intervals of which the sensor values are within the query value range. There may be multiple time intervals of which sensor values satisfy the query value range.

The generic process to query model-view sensor data queries comprises the following two steps:

- Searching of qualified segments: The qualified segments are the segments whose time (resp. value) intervals intersect the query time (resp. value) range or point. This step should make use of an interval index to locate all the qualified modeled segments in the segment model store.
- Model gridding: Qualified segments are too abstract and a finite set of data points are more useful as query results for end-users [2]. Therefore, model gridding is another necessary process [2,3]. Model gridding applies three operations to each qualified segment: (i) It discretizes the time interval of the segment at a specific granularity to generate a set of time points. (ii) It generates the sensor values at the discrete time points based on the model that approximates the segment. (iii) It filters out the sensor data that does not satisfy the query predicates. The qualified time or value points that result from gridding all qualified modeled segments are returned as query results.

As shown in Fig. 1(c), segment one $S1$ is found as one qualified segment for time query range $[10, 70]$, model-gridding discretizes the time interval of $S1$ and evaluates the values at each time point. Then, the qualified time-value pairs constitute the query results.

4. Key-value interval index

We will first present the design of the two-tier model index on key-value stores and then we will discuss the updating algorithm of the model index.

4.1. Structure of KVI-index

As each segment of model-view sensor data has a specific time and value range, instead of indexing the mathematical functions of segments, our idea is to take the time and value intervals as keys to index each segment, which allows the index to directly serve the queries proposed in Section 3. We propose the **k**ey-**v**alue represented **i**nterval index (KVI-index) to index time and value intervals of model-view sensor data. For a segment in Fig. 3(c), the time and value intervals are respectively indexed by the KVI-index, as depicted in Figs. 3(a) and (b). Our KVI-index adopts the idea from interval tree, since in-memory interval tree's primary–secondary structure is convenient for externalization to the key-value store [26]. Furthermore, the searching and scanning algorithm of the interval tree is suitable for the MapReduce computing paradigm.

Our KVI-index is a novel in-memory and key-value composite index structure. The virtual searching tree (*vs-tree*) resides in memory, while an index-model table in the key-value store is devised to materialize the secondary structure (*SS*) of each node in *vs-tree*.

4.1.1. In-memory structure

The in-memory virtual searching tree (*vs-tree*) is a standard binary search tree shown in Fig. 3(a). Each time (or value) interval is registered on only one node of *vs-tree*: the one with which the interval first overlaps along the searching path from root. This node is defined as the registration node τ for this interval. Each node of *vs-tree* has an associated secondary structure (*SS*), materialized in the key-value store, which stores the substantial information of the modeled segments registered at this node.

We apply space-partition strategy for *vs-tree*. The height of the *vs-tree* is denoted by h. We set the value of leftmost leaf node as 0. For negative sensor data values, we use simple shifting to have the data range start from 0 for convenience. Then, the domain of the *vs-tree* is $[0, R]$ and $R = 2^{(h+1)} - 2$. The value of root node is $r = 2^h - 1$. During the whole life of KVI-index, only the root value r is kept, since, due to the lightweight computability of the space-partition, the value of each node in the searching path from the root to the node that has the queried point or interval can be calculated in run time. All the operations on *vs-tree* are performed in memory and are thus very efficient. As the domains of time and value of the sensor data are different, two *vs-tree*s one for times and another for values are kept in memory simultaneously for answering time and value queries respectively.

4.1.2. Index-model table

Our novel index-model composite storage schema enables one table not only to store the modeled segments, but also to materi-

Please cite this article in press as: T. Guo et al., Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud, Big Data Research (2014), http://dx.doi.org/10.1016/j.bdr.2014.07.005

Algorithm 2: Time (or value) KVI-index updating.

```
Input: [l_v, r_v], [l_t, r_t], /* value and time intervals of one modeled segment
       M_i, r /* M_i denotes the coefficients of the modeled segment
1  begin
2      /* dynamic domain expansion
3      if (r_t > R) then
4          r = 2^(⌈log(r_t+2)⌉−1) − 1 /* expand to new root value
5      /* registration node search
6      node = 2^(⌊log r_t⌋+1) − 1;
7      h = log(node) − 1;
8      while (h ≥ 0) do
9          if (l_t ≤ node and r_t ≥ node) then
10             break; /* node is the registration node
11         else
12             if (l_t > node) then
13                 node = node + 2^h;
14             if (r_t < node) then
15                 node = node − 2^h;
16             h = h − 1;
17     /* materialized into the index-model table.
18     if the SS of node has been initialized then
19         rowkey = ⟨node|l_t|r_t⟩;
20     else
21         rowkey = ⟨node|α⟩;
22     insert [l_v, r_v], [l_t, r_t], M_i into the table.
```

alize the structural information of the *vs-tree*, i.e., the *SS*s for each tree node.

The index-model table is shown in Fig. 3(b). Each row corresponds to only one modeled segment of sensor data, e.g., the data segment shown in the black dotted rectangle in Fig. 3(c). A row key consists of the node value and the interval of an indexed modeled segment at that node. The time interval, value interval and the model coefficients are all stored in different columns of the same row. And the *SS*s of each node correspond to a consecutive range of tuples in the index-model table. For instance, the rows corresponding to the *SS*s of node 1, node 5 and node 13 in *vs-tree* are illustrated in Fig. 3(b). Analogously, we have two index-model tables that correspond to time and value *vs-trees* respectively.

4.2. KVI-index updates

The complete modeled segment updating algorithm of KVI-index is shown in Algorithm 2. It includes two processes:

(1) *Registration node searching*: locate the node τ at which one time (value) interval $[l_t, r_t]$ should be registered.
(2) *Materialization of modeled segments*: construct the row-key based on the τ and materialize one modeled segment's information into the columns of the corresponding row.

4.2.1. Registration node searching (rSearch)

This algorithm first involves a *domain-expansion process* to dynamically adjust the domain of the *vs-tree* according to the specific domain of the sensor data. Then, the registration node can be found on the validated *vs-tree*.

Lemma 1. *For a modeled segment M_i with time (value) interval $[l_t, r_t]$ (resp. $[l_v, r_v]$), its registration node lies in a tree rooted at $2^{\lceil \log(r_t+2) \rceil - 1} - 1$.*

Proof. The domain of one tree rooted at $2^{\lceil \log(r_t+2) \rceil - 1} - 1$ is $[0, 2^{\lceil \log(r_t+2) \rceil} - 2]$. As $2^{\lceil \log(r_t+2) \rceil} - 2 \geq 2^{\log(r_t+2)} - 2 = r_t$, the registration node of interval $[l_t, r_t]$ must be in a tree rooted at $2^{\lceil \log(r_t+2) \rceil - 1} - 1$. □

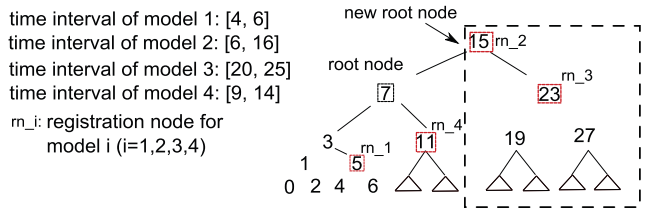


Fig. 4. Registration node searching of KVI-index.

Lemma 2. *For a modeled segment M_i with time (value) interval (l_t, r_t), if the right end-point r_t satisfies $r_t > R$, the domain of vs-tree needs to expand.*

Proof. The current *vs-tree*'s height is h and $l_t \leq r$, the interval $[l_t, r_t]$ rides over the root node r. Assume that we do not expand the domain and hang $[l_t, r_t]$ on r. When a new model M_j with a time (or value) interval $[l'_t, r'_t]$ (or $[l'_v, r'_v]$) and $l'_t > R$ comes, the root value has to increase to $r' = 2^{\lceil \log r'_t + 2 \rceil - 1} - 1$ (resp. $r' = \cdots$) as $[l'_t, r'_t]$ (resp. $[l'_v, r'_v]$) intersects no node of current *vs-tree*. Then between r and r', there is one node with value $2^{(h+1)} - 1$. The interval (l_t, r_t) is stored at node $r = 2^h - 1$ and $r_t > 2^{(h+1)} - 2 \Rightarrow r_t \geq 2^{(h+1)} - 1$, therefore registering the interval $[l_t, r_t]$ on node r contradicts with the interval registration rule. □

Using Lemmas 1 and 2, KVI-index is able to dynamically decide when and how to adjust the domain $[0, R]$ of *vs-tree*. The complete *rSearch* can be illustrated by Fig. 4. When *model*1 is to be inserted, the *vs-tree* rooted at *node*7 is still valid. *model*1 is registered at *node*5. However, when *model*2 arrives, its right end-point, i.e., 16, exceeds the domain $[0, 14]$. The *vs-tree* is expanded having 15 as new root and the new extended domain is the area enclosed by the dotted block in Fig. 4. Then, the sub-sequential *model*2 and *model*3 can be updated successfully.

The rSearch process can be further optimized via adaptively adjusting the starting point based on the interval to index. In this way, rSearch does not need to always search from root node, thereby shortening the length of searching path. Based on Lemma 1, for one interval (l_t, r_t), we can start to search for registration node from node $r_0 = 2^{\lfloor \log tr \rfloor + 1} - 1$ rather than root node r, which can be referred to Line 7 in Algorithm 2. Also, the nodes outside of sub-domain $[0, 2^{\lfloor \log tr + 1 \rfloor + 1} - 2]$ of *vs-tree* will not become the registration node, because the interval (tl, tr) does not intersect with them. For example, the *model*4 in Fig. 4. The adaptive searching finds that the subtree rooted at node 7 is the minimum one covering *model*4's interval $[9, 14]$. Therefore, node 7 is the starting point for registration node searching and the length of searching path is only 1.

4.2.2. Materialization of modeled segment

When materializing model M_i into the *SS* of a node τ, the row-key may be chosen in two ways:

- Upon initialization of the *SS* of node τ: when no modeled segment has been stored at τ's *SS*, the row key is chosen as $\langle \tau, \alpha \rangle$ for model M_i. Here, α is a postfix of row key to indicate that this row is the starting position of τ's *SS* in the table.
- Upon updating the *SS* of node τ: when the *SS* of τ has already been initialized, the time interval $[l_t, r_t]$ (resp. $[l_v, r_v]$ for value interval) to be indexed will be incorporated into the row key, i.e., $\langle \tau, l_t, r_t \rangle$ (resp. $\langle \tau, l_v, r_v \rangle$ in the index-model table for values). In this way, different modeled segments stored in the same *SS* of a node do not overwrite each other.

Please cite this article in press as: T. Guo et al., Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud, Big Data Research (2014), http://dx.doi.org/10.1016/j.bdr.2014.07.005

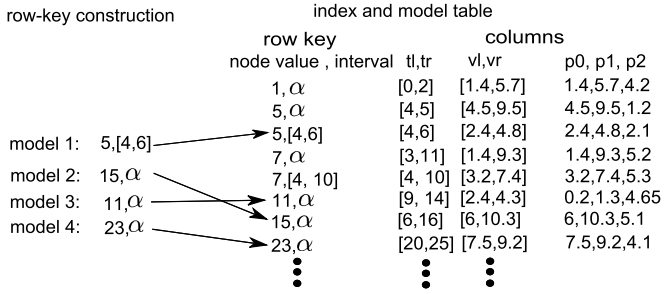


Fig. 5. Modeled segment materialization of KVI-index.

The selection of specific α should make sure that the binary representation of $\langle \tau, \alpha \rangle$ is in front of any other $\langle \tau, l_t, r_t \rangle$. This design is useful for query processing. For instance, if the query processor requires to access all the modeled segments stored at registration node 5, then we know that all the corresponding modeled segments lie in the rows within the row-key range $[\langle 5, \alpha \rangle, \langle 6, \alpha \rangle)$. For example, take the *model*1 and *model*2 in Fig. 5. First, the KVI-index checks whether the starting modeled segments of *node*5 and *node*15, namely rows with key $\langle 5, \alpha \rangle$ and $\langle 15, \alpha \rangle$, exist. Then, the row key $\langle 5, 4, 6 \rangle$ is constructed for *model*1 as the *SS* of *node*5 has been initialized, whilst KVI-index constructs the row key $\langle 15, \alpha \rangle$ for *model*2.

4.2.3. Complexity analysis

In this section, we analyze the computational and communication complexities of updating operation of KVI-index. The complexity analysis of KVI-index includes in-memory and key-value part.

- In-memory
  The *rSearch* on *vs-tree* can run within $O(\log(R))$ time. The space expansion costs $O(1)$ time. Only the root value r of *vs-tree* is kept in memory. The values of other nodes on *vs-tree* can be calculated due to the computability of *vs-tree*'s space-partition. Therefore, the space complexity of *vs-tree* is $O(1)$.
- In key-value store
  The update operation on *vs-tree* does not generate any network I/O cost. For an n-th order polynomial model of one sensor data segment, $(2+n)$ put operations are conducted to materialize one model. Therefore, the time complexity is $O(1)$ in terms of network I/O, as n is constant. Moreover, one segment model's time (or value) interval and coefficients are only materialized once into the index-model table, thus the space cost is $O(N)$ for time (or value) index.

5. Query processing via KVI-index and MapReduce

For querying model-view sensor data, the searching process of qualified modeled segments (defined in Section 3) in KVI-index includes two steps:

- *Intersection and point search*: The intersection search on *vs-tree* is used for range queries, while point search is employed for point queries. They are responsible for collecting the nodes that accommodate qualified modeled segments in their secondary structures *SS*s.
- *Modeled-segment filtering*: Due to the rule for interval registration at the nodes of the *vs-tree*, the *SS* of a node may contain some intervals irrelevant to queried range or point. In KVI-index, the *SS*s of all nodes found by the search operation are accessed to filter out unqualified segments.

Algorithm 3: *iSearch*$^+$ of *vs-tree*.

```
Input: time query range [l_t, r_t], root value r
Output: node set S_0 and D
1  begin
2      /* construct S_0
3      node = r; h = log(r) − 1;
4      while (h ≥ 0) do
5          if (l_t ≤ node and r_t ≥ node) then
6              break; /* node is the registration node
7          else
8              S_0 = S_0 ∪ node
9              if (l_t > node) then
10                 node = node + 2^h;
11             if (r_t < node) then
12                 node = node − 2^h;
13             h = h − 1;
14     /* construct D.
15     l_s = 2^⌊log(l_t)⌋, r_s = R − 2^⌊log(R−r_t)⌋, D = [l_s, r_s]
```

After the above two steps, model gridding component fetches the coefficients of each qualified modeled segment and performs gridding. Next, we first describe an enhanced intersection search algorithm on *vs-tree* that benefits KVI–Scan–MapReduce query processing, introduced later in this section. We then present the point search algorithm on *vs-tree*. Subsequently, we introduce our novel hybrid KVI–Scan–MapReduce query processing. Last, we theoretically analyze the enhanced intersection search algorithm of KVI-index.

5.1. Intersection and point search

5.1.1. Enhanced interval intersection search

Algorithm 3 presents the *iSearch*$^+$. Given a time (resp. value) range query $[l_t, r_t]$, *iSearch*$^+$ first calls the *rSearch* to find the registration node τ of $[l_t, r_t]$. The nodes on the searching path from the root node to the one preceding τ form a node set denoted by $\mathcal{S}_0$. The *iSearch*$^+$ stops at the node, which is closest to the left-end point l_t. All the nodes along the left-descending path form a node set, denoted by $\mathcal{S}_l$, while the node with the minimum value in this path is denoted by l_s. Analogously, $\mathcal{S}_r$ is the node set from the right-descending path and r_s is the node with the maximum value in this path. Any node outside the range $[l_s, r_s]$ and the set $\mathcal{S}_0$ does not have any qualified modeled segments.

The traditional intersection search would return the node set $\mathcal{C} = \mathcal{S}_0 \cup \mathcal{S}_l \cup \mathcal{S}_r \cup [l_t, r_t]$ for further modeled-segment filtering and gridding. Our *iSearch*$^+$ outputs the discrete node set $\mathcal{S}_0$ and a consecutive range of nodes $\mathcal{D} = [l_s, r_s]$. For example, take the range query in Fig. 6(a). *node*7 is the registration node of query range $[6, 10]$. The traditional *iSearch* returns the discrete node sets shown in the solid boxes of Fig. 6(a), while our *iSearch*$^+$ returns a range of nodes $[3, 11]$ and $\mathcal{S}_0 = \{15\}$. We will see how the output of *iSearch*$^+$ benefits the hybrid query processing later in Section 5.2.

5.1.2. Point search

We denote the point search by *sSearch* as it functions as the stabbing search of interval tree. The *sSearch* is a binary search that records the nodes along the descending path. We present the *sSearch* in Fig. 6(a). For example, when querying the sensor value of time point 24, the node set $\mathcal{S}_0 = \{15, 7, 11, 9, 10\}$ is returned by *sSearch*. Since there is no split searching, as in *iSearch*$^+$, only one node set is produced here. We denote this node set by $\mathcal{S}_0$ as well, so as to facilitate the description of the hybrid KVI–Scan–MapReduce query processing that follows next.

Please cite this article in press as: T. Guo et al., Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud, Big Data Research (2014), http://dx.doi.org/10.1016/j.bdr.2014.07.005

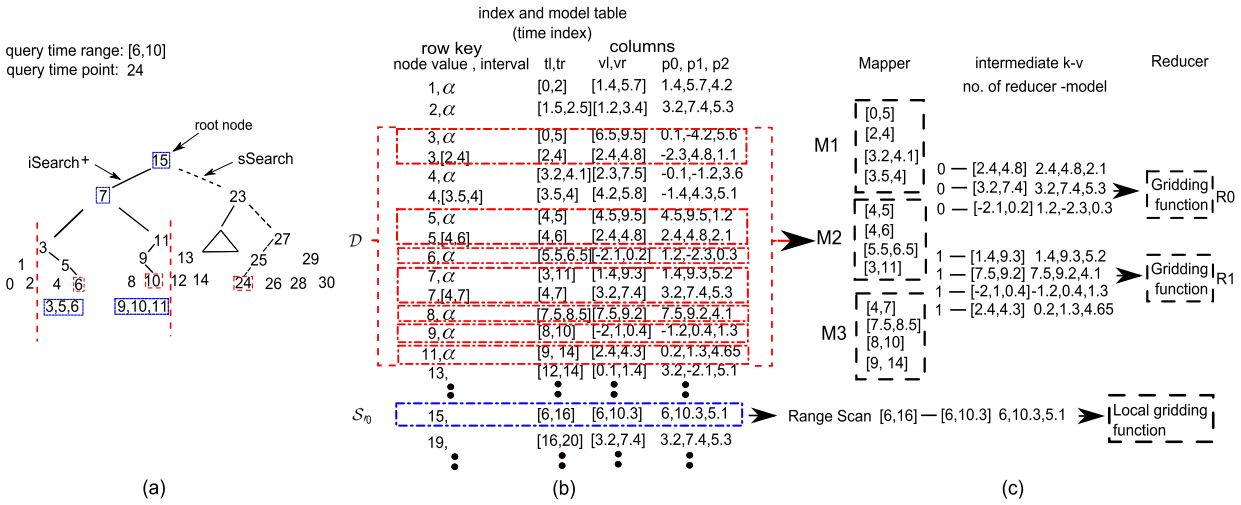


Fig. 6. Workflow of KVI–Scan–MapReduce approach. (a) *iSearch*$^+$ and *sSearch*. (b) *SS* location distribution for the range query. (c) Hybrid processing.

5.2. KVI–Scan–MapReduce query processing

The conventional clustered index for one-dimensional data can exactly locate a consecutive range of qualified data. Then, the query processor just needs to do range scan on these qualified data. However, as for querying model-view sensor data, the challenge is how to tackle the large amount of segment models from the *SSs* distributed across the index-model table. We first analyze the location distribution of the *SSs* of the nodes found by *iSearch*$^+$ and *sSearch* in the index-model table. The characteristics of this distribution inspired us to propose the hybrid KVI–Scan–MapReduce query processing approach.

5.2.1. SS location distribution

There are two cases for the *SS* distribution in the index-model table, described below.

- $\mathcal{S}_0$: The *SSs* of $\mathcal{S}_0$ are non-consecutive and sparsely distributed in the index-model table. The node value is the primary part of the row-key; thus, the distance between *SSs* of $\mathcal{S}_0$ depends on the numerical difference of node values. As $\mathcal{S}_0$ includes the nodes from root node to the one preceding the τ, the intra-distances between any consecutive nodes in $\mathcal{S}_0$ are 2^{h-i}, where $i = 0, \ldots, h - d_\tau$ is the position of the node in the descending search path $\mathcal{S}_0$ and d_τ is the depth of τ. Obviously, the intra-distances in $\mathcal{S}_0$ are greater than those for other nodes below τ in the search path.
- $\mathcal{D}$: The *SSs* of $[l_s, r_s]$ are clustered around the ones of $[l_t, r_t]$ in the index-model table. The *SSs* of $[l_t, r_t]$ are all adjacent in the index-model table. The *SSs* of $[l_s, r_s]$ are bounded by those of the sub-tree rooted at τ and the nodes in $[l_s, r_s]$ are a superset of the nodes in $[l_t, r_t]$. The deeper the registration node τ is located, the tighter the set of the *SSs* of $[l_s, r_s]$ over those of $[l_t, r_t]$.

For example, take the time (or value) query range $[6, 10]$ in Fig. 6(a). The registration node is *node*7. Then, $\mathcal{S}_0 = \{15\}$ and $\mathcal{D} = [3, 11]$. The sub-tree rooted at *node*7 covers the node range $\mathcal{E} = [0, 14]$ and $\mathcal{D} \subset \mathcal{E}$. From Fig. 6(b), the *SSs* of $\mathcal{D}$ are clustered around those of $[6, 10]$ and bounded by the *SSs* of $\mathcal{E}$. However, *node*15's *SS* is located far away from those of $[3, 11]$.

If *SSs* of $\mathcal{S}_0$ and $\mathcal{D}$ are processed via straightforward random access and range scan provided by key-value stores, the entire modeled-segment filtering and gridding processes are conducted locally at the application side. For a table of multiple or hundreds of GBs, the communication and computation costs are prohibitively high for the application side even for low-selective queries.

The modeled segment filtering–gridding processing matches MapReduce's filtering-aggregation paradigm. Considering the research results from [31–33], for CPU non-intensive workload, I/O cost, network latency and starting-up overhead of mappers are dominant in the execution time of MapReduce programs. If the *SSs* of $\mathcal{S}_0$ and $\mathcal{D}$ are all processed by MapReduce, a lot of time is wasted for mappers that process irrelevant *SSs* in the index-model table. This is because MapReduce will access the continuous regions of the table including the *SSs* of nodes between the $\mathcal{S}_0$ and $\mathcal{D}$ due to the sequential data feeding mechanism in the mapping phase. For example, in Fig. 6(b), the *SSs* of $\mathcal{D} = [3, 11]$ and $\mathcal{S}_0 = \{15\}$ are distant in the table. Hence, MapReduce will launch many unnecessary mappers for the irrelevant *SSs* of nodes between 11 and 15, in order to process the *SSs* of $\mathcal{S}_0$ and $\mathcal{D}$.

5.2.2. Hybrid model filtering and gridding

As discussed above, simply using range scan or MapReduce to process *SSs* are both problematic. Our idea is to design a hybrid KVI–Scan–MapReduce paradigm that combines range scan and MapReduce for processing *SSs*, as follows:

- (1) $\mathcal{S}_0$: the height of *vs-tree* is bounded by $\log(R)$, and thus the amount of computation on $\mathcal{S}_0$ is limited. As the *SSs* of $\mathcal{S}_0$ are sparsely distributed in the index-model table and each *SS* of $\mathcal{S}_0$ can be considered to be a small range of clustered index, the random-access- and range-scan-based model filtering and gridding is suitable.
- (2) $D = [l_s, r_s]$: the successive range $[l_s, r_s]$ delimits a tight boundary of the sub-index-model table over the relevant *SSs* that are suitable for processing with MapReduce.

This hybrid paradigm eliminates the Map-phase processing of *SSs* of irrelevant nodes between $\mathcal{S}_0$ and $\mathcal{D}$ and the nodes between the elements of $\mathcal{S}_0$. Moreover, it is non-intrusive for both the key-value store and MapReduce. Regarding the time (or value) point query, it only produces the node set $\mathcal{S}_0$ without $\mathcal{D}$, hence, only range-scan-based model filtering and gridding is needed.

Suppose that the number of reducers is P and each reducer is denoted by $0, \ldots, P - 1$. For range queries, the partition function f is used to assign the qualified modeled segments into different reducers. It is designed on the basis of query time (resp. value) range $[l_t, r_t]$ (resp. $[l_v, r_v]$) and the time (or value) interval $[l_i, r_i]$ of each modeled segment i. The idea is that each of the reducers

is in charge of one even sub-range $\frac{r_t - l_t}{P}$. Such a partition function f is given in Eq. (1).

$$f(r_i) = \begin{cases} l_t \le r_i \le r_t & \lfloor \frac{(r_i - l_t) * P}{r_t - l_t} \rfloor \\ r_i \ge r_t & P - 1 \end{cases} \quad (1)$$

The functionalities of mappers and reducers are depicted in detail below.

- **Mapper**: Each mapper gets the time (resp. value) interval $[l_i, r_i]$ of one modeled segment i to check whether it intersects with the query time (resp. value) range. For the qualified modeled segments, the intermediate key is derived by the partition function $f(r_i)$. The model coefficients $\langle p_i^1, \ldots, p_i^n \rangle$ are the value part of the intermediate key-value pair.
- **Reducer**: One reducer receives a list of qualified modeled segments $\langle p_0^1, \ldots, p_0^n \rangle, \langle p_1^1, \ldots, p_1^n \rangle, \ldots$. For each modeled segment $\langle p_i^1, \ldots, p_i^n \rangle$, the reducer invokes a model-based gridding function to compute discrete values as query results.

Regarding the scan-based model filtering and gridding, as *SSs* in $\mathcal{S}_0$ are located in different regions of the index-model table, the query processor makes use of thread pool to process each *SS* of $\mathcal{S}_0$ in parallel. Fig. 6 shows the workflow of the hybrid KVI–Scan–MapReduce approach. For a time (or value) range query $[6, 10]$, *iSearch*$^+$ constructs the node set $\mathcal{S}_0 = \{15\}$ and $\mathcal{D} = [3, 11]$. Then, the *SSs* of the nodes in $\mathcal{D}$ are sent to MapReduce. The *SS* of *node*15, enclosed by the bottom dot-dashed block, is processed via range scan.

5.3. Theoretical analysis

One point to carefully consider is that *iSearch*$^+$ may generate redundant nodes, because the *iSearch*$^+$ aims to find a tight and consecutive range of *SSs* for MapReduce. For instance, in Fig. 6(a), the *SS* of *node*4 is not accessed by the *iSearch*$^+$, but is in the sub-table processed by MapReduce.

Theorem 1. *For a range query $[l_t, r_t]$, the redundant nodes in $[l_s, r_s]$ returned by iSearch$^+$ are bounded.*

Proof. Assume h to be the height of the registration node τ. Consider the left-descending path from τ to the node l_0 closest to l_t. Let d be the depth from which the descending path turns right, namely the value $w < l_t$ of the current node. Then, based on the *iSearch*$^+$ algorithm, w is the left boundary of accessed node range and $w = \tau - \sum_{i=1}^{d} 2_{h-i}$.

For a certain value of d, the worst case happens when the descending process continues to go right until reaching l_0, as the nodes between w and l_0 are all redundant ones. The number of nodes returned by *iSearch*$^+$ under this case is given by:

$$U = \tau - \left(\tau - \sum_{i=1}^{d} 2^{h-i} \right) \quad (2)$$

The nodes between τ and w are all included into the output range $\mathcal{D}$ of *iSearch*$^+$. The number of nodes returned by the conventional *iSearch* is given by:

$$V = h - d + \left\{ \tau - \left(\tau - \sum_{i=1}^{d} 2^{h-i} + \sum_{i=d+1}^{h} 2^{h-i} \right) \right\} \quad (3)$$

Therefore, the number of redundant nodes returned by *iSearch*$^+$ is given by:

$$\begin{aligned} f = U - V &= d + \sum_{i=d+1}^{h} 2^{h-i} - h \\ &= d + 2^{h-d} - h - 1 \end{aligned} \quad (4)$$

Eq. (4) is a function of d and is monotonous decreasing in d's domain $[1, h]$. Consequently, when $d = 1$, the function f reaches the maximum value, namely, the number of redundant nodes from *iSearch*$^+$ attains the maximum value f_{max} that is given by:

$$f_{max} = 2^{h-1} - h. \quad (5)$$

As the total number η of nodes of the sub-tree of the left child of τ is $2^h - 1$, hence

$$f \le \frac{1}{2}\eta - \log(\eta + 1) + \frac{1}{2}. \quad (6)$$

In summary, the total number of redundant nodes in the range $[l_s, r_s]$ is bounded. □

The worst case happens when the endpoints l_t and r_t are the preceding and succeeding nodes of τ, namely $l_t = \tau - 1$ and $r_t = \tau + 1$. However, for most of the cases, the redundant nodes returned from *iSearch*$^+$ are very limited.

6. Experimental evaluation

First, we compare model-view sensor data query processing with conventional one over raw sensor data. Then, we show that our KVI–Scan–MapReduce (*KSM*) approach outperforms other model-view sensor data querying approaches. Finally, we experimentally explore the factors that affect the performance of KVI–Scan–MapReduce.

6.1. Setup

We employ accelerometer data from mobile phones as sensor data set. The size of raw sensor data is 22 GB including 200 million data points. After modeling, the modeled segments of the sensor data take 12 GB, while there are around 25 million modeled segments.

We developed our system using the versions of HBase and Hadoop in Cloudera CDH4.3.0. The experiments are performed on our own cluster that consists of 1 master node and 8 slaves. The master node has 64 GB RAM, 3 TB disk space (4 × 1 TB disks in RAID5) and 12 cores, each of which is 2.30 GHz (Intel Xeon E5-2630). Each slave node has 6 cores 2.30 GHz (Intel Xeon E5-2630), 32 GB RAM and 6 TB disk space (3 × 2 TB disks). Nodes are connected via 1 GB Ethernet. In the experimental results, we refer to query selectivity as the ratio of the number of qualified modeled segments over that of total modeled segments.

The data set contains discrete accelerometer data from mobile phones and is a sequence of tuples each of which has one timestamp and three sensor values representing the coordinates. The size of the raw sensor data set is 22 GB including 200 million data points. We simulate the sensor data emission, in order to segment and update sensor data into the KVM-index in an online manner. We implement an online sensor data segmentation component [5] applying the *PCA* (piecewise constant approximation) [1], which approximates one segment with a constant value (e.g., the mean value of the segment). Since how to segment and model sensor data is not the focus of this paper, other sensor time series segmentation approaches could also have been applied here. Provided that the segments are characterized by the time and value intervals, our KVM-index and related query-processing techniques are able to manage them efficiently in the cloud. Finally, there

Please cite this article in press as: T. Guo et al., Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud, Big Data Research (2014), http://dx.doi.org/10.1016/j.bdr.2014.07.005

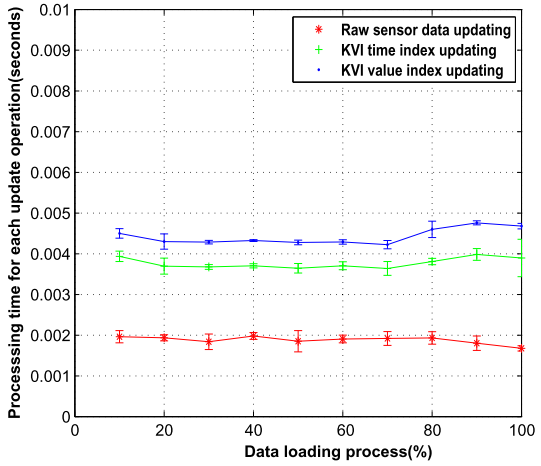


Fig. 7. Sensor data updating performance.

are around 25 million sensor data segments (nearly 15 GB) uploaded into the key-value store. Regarding the segment gridding, we choose 1 second as the time granularity which is application-specific.

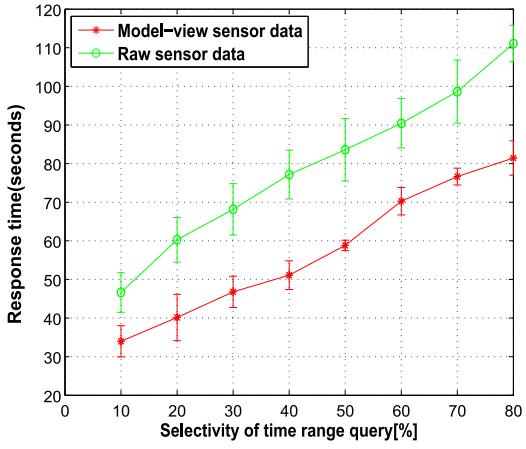
6.2. Index updating

Fig. 7 shows the average updating time of each segment and the average insertion time of each raw sensor data point during the data uploading phase. Both time and value index keep relatively stable updating efficiency. The updating of the value KVI-index is a little slower than the time KVI-index. As discussed before, since the domain of the value *vs-tree* is smaller than that of the time *vs-tree*, the value index performs more *SS* updating operations (discussed in Section 4.2) than the time index and therefore incurs more network I/O cost. The raw sensor data insertion is the fastest but the amount of data to update is much larger than model-view approach. This is because model-view sensor data achieves data compression over the raw sensor data thereby decreasing the amount of data to upload.

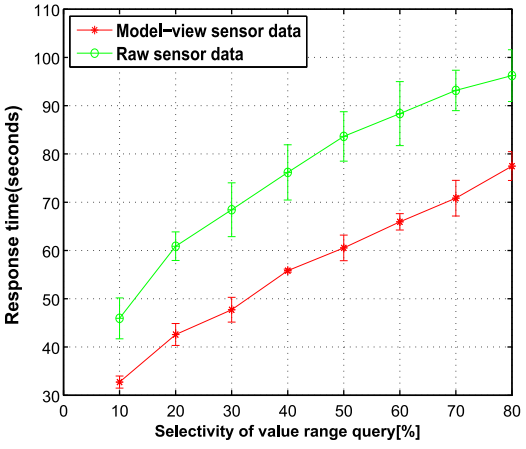
6.3. Model-view sensor data vs. raw sensor data

We create two tables, which take the time-stamp and sensor value as the row-keys respectively, such that the query range or point can be used as keys to locate the qualified data points. Then, the query processor invokes the MapReduce to access the large size of data points for getting query results.

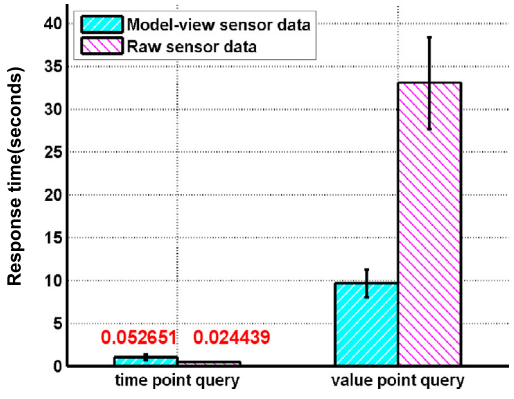
Figs. 8(a), (b) and (c) present the query response times for time range, value range and point queries respectively. As shown in Figs. 8(a) and (b), the model-view approach takes around 30% less time than the raw sensor data method for both time and value range queries. Although the raw sensor data based methods apply MapReduce to directly access the qualified tuples via the row-key based range scan, the amount of raw sensor data to process is much larger than that of the model-view approach. In Fig. 8(c), the processing time of the raw data based method is 2× less than that of the model-view one in time point queries, because the raw data method can use the query time point as index key to directly access the relevant data points, while our *KSM* requires to perform model filtering and gridding. For value point queries, the model-



(a) Time range queries



(b) Value range queries



(c) Point queries

Fig. 8. Query performance comparison of raw data and model-view approaches on range and point queries.

Please cite this article in press as: T. Guo et al., Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud, Big Data Research (2014), http://dx.doi.org/10.1016/j.bdr.2014.07.005

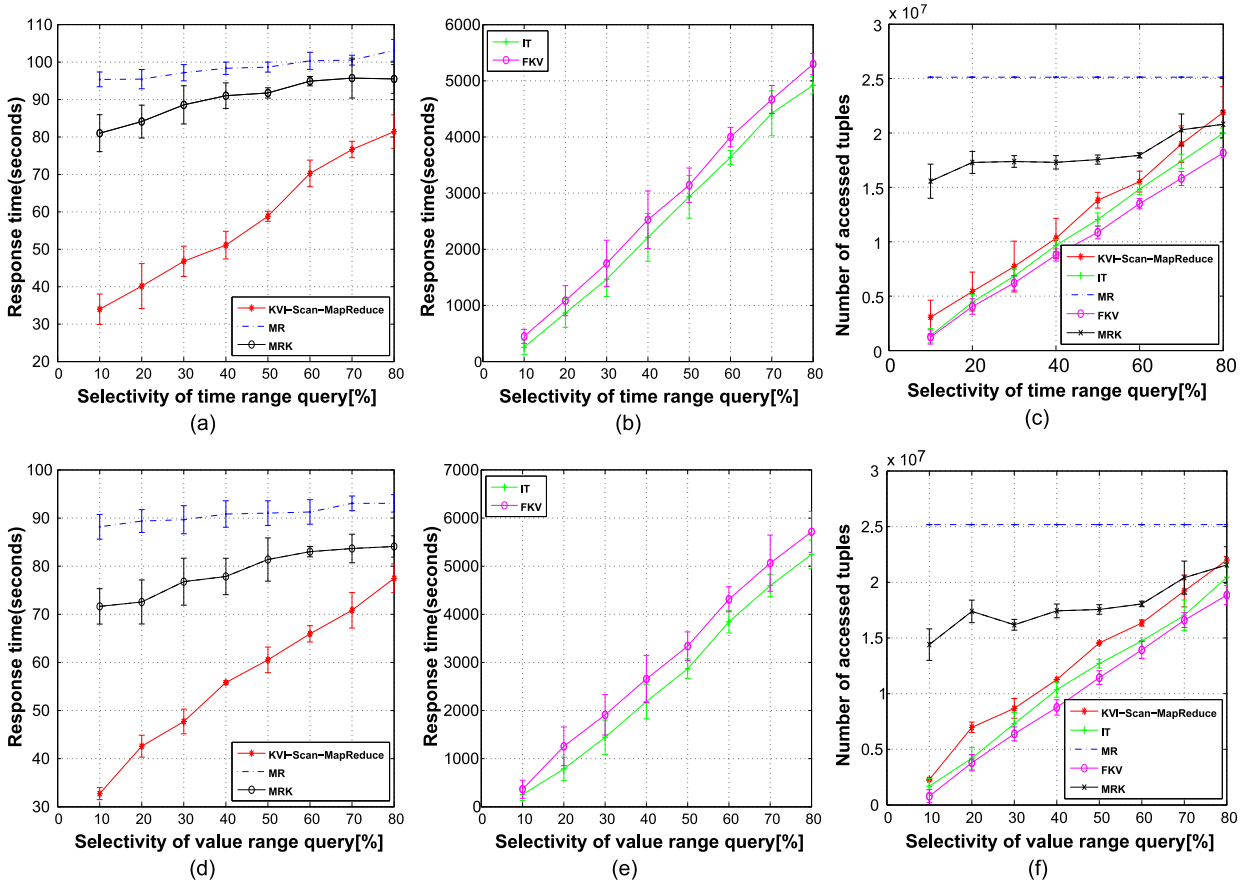


Fig. 9. Query performance comparison of different model-view approaches. (a)–(c): time range queries, (d)–(g): value range queries.

view approach has nearly 3× less time than the raw data method. As, normally, there is a large size of data points with the queried value, MapReduce is used to access this qualified sensor data set. In the model-view approach, the point query processing only uses random access and range scan to get qualified modeled segments for gridding locally, and thus it saves the time on starting MapReduce to access data.

6.4. Comparison of model-view approaches

There are four baseline approaches for querying model-view sensor data, namely:

MapReduce (MR). This approach utilizes MapReduce without support from any index. It always works on the whole index-model table to filter the qualified modeled segments in the mapping phase and perform the model gridding in the reduce phase.

Interval tree (IT). We implemented the traditional query processing operations of the interval tree [23,26] by adding another table to store the *SS* of each node sorted by the right end-point of intervals. Each index, time or value, has two associated tables. During the intersection or point search on *vs-tree*, the query processor decides which table to access based on the relation between the query range (or point) and the node value. In this way, the query processor can stop scanning once it encounters one unqualified modeled segment, due to the monotonicity of end-points of modeled segments. *IT* makes use of random access and range scan to sequentially filter the qualified modeled segments and make gridding locally.

MapReduce+KVI (MRK). The idea of *MRK* is to leverage KVI-index to avoid having MapReduce to process the whole table. In *MRK*, MapReduce is designed to work over one continuous sub-index-model table including all the *SSs* of the accessed nodes in search operations. For instance, in Fig. 6(a), for a time (or value) range query [6, 10], *MRK* invokes MapReduce to work on the sub-table within the row-key range $[\langle 3, \alpha\rangle, \langle 16, \alpha\rangle]$. The same idea applies for point queries. As compared to our hybrid *KSM* approach, *MRK* is a lightweight indexing-MapReduce plan, as it processes many irrelevant *SSs* of nodes between $\mathcal{S}_0$ and $\mathcal{D}$.

Filter of key-value store (FKV). Some key-value stores such as HBase provide a filter functionality to support predicate-based row selection [20]. The filter transmits the filtering predicate to each region server and then all servers scan their local data in parallel. Afterwards, they return the qualified tuples. Our filter-based query processing also works on the index-model table, as the filtering predicates can be directly applied to the columns. The query processor waits until all region servers finish scans and then it retrieves each returned qualified modeled segment to conduct gridding locally.

6.4.1. Range query

Figs. 9(a), (b) and (c) present the performance of time range queries. As depicted in Fig. 9(a), *KSM* outperforms *MR* up to 3× for the low-selective time range queries. As the query selectivity increases, the amount of *SSs* for scan based processing decreases and that for MapReduce approaches the entire table. Therefore, the response time increases and approaches that of *MR*. The response time of *MR* increases little. As increasing query selectivity leads to ascending gridding workload in reduce phase, these results show that the overhead from model gridding is not dominant in *MR*. The response time of *MRK* is more than that of *KSM*, but less than that of *MR*. As *MRK* utilizes the KVI-index to localize a consecutive sub-index-model table covering all the *SSs* of nodes found by intersection search, it processes fewer modeled segments than *MR*'s full table scanning. Yet, as compared to *KSM*, *MRK* processes more

Please cite this article in press as: T. Guo et al., Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud, Big Data Research (2014), http://dx.doi.org/10.1016/j.bdr.2014.07.005

redundant modeled segments. Moreover, as the sub-table in *MRK* covers a large range, the processing time of *MRK* increases little for low-selective queries.

Fig. 9(b) exhibits the performance of *IT* and *FKV* approaches. As *FKV* needs to wait for each region server of HBase to finish the local data scanning, its total response time is a little longer than that of *IT* approach. They both consume much more time than all *MR*, *MRK* and *KSM*, as they apply sequential accessing of modeled segments.

We also analyze the number of modeled segments accessed by each approach in Fig. 9(c). These experiments show how different access methods of modeled segments affect the performance. *MR* works on the entire table, thus, the number of accessed segments is the same. From the point of view of the application, only qualified modeled segments are returned for gridding, thus *FKV* processes no redundant modeled segments and consumes the least amount of modeled segments. Since *IT* scans the *SS* of one node until encountering an unqualified model, the total number of accessed segments is a little larger than that of *FKV*. Our *KSM* processes larger number of segments than both *IT* and *FKV* due to the continuous and redundant range of *SS*s found by *iSearch*$^+$. However, the results also verify our theoretical analysis that the amount of redundant modeled segments is bounded. *MRK* accesses more segments than *IT*, *FKV* and *KSM*, as it adds the *SS*s between $\mathcal{S}_0$ and $\mathcal{D}$ to form a continuous sub-table for MapReduce. Referring to Fig. 9(a) and Fig. 9(b), although *KSM* approach consumes more segments than *IT* and *FKV*, its hybrid paradigm is the most efficient.

Figs. 9(d), (e) and (f) present the value range query performance. The different query processing approaches exhibit similar patterns as for the time range queries, so we skip the detailed analysis.

6.4.2. Point query

The time and value point query processing performance are shown in Fig. 10. *IT* wins both for time and value point queries. The response time of *KSM* is a little greater than *IT*, but outperforms the other approaches, because *IT* is able to access all qualified modeled segments in one *SS*. However, the *KSM* scans the whole *SS* entries of a node to find the qualified ones. Because of the invocation of MapReduce and redundant modeled segments in the sub-table, *MRK* takes more time than both *IT* and *KSM*. But, as *MRK* does not work on the entire table as *MR* does, it takes about 2× less time than *MR*. *FKV* consumes the most time as it needs to wait for server-side full table scan before gridding operations. Since the size of the domain of the sensor data values is smaller than that of the time domain, nodes in value *vs-tree* accommodate more modeled segments than time *vs-tree* nodes. Thus, the response times of value point queries of *IT*, *FKV* and *KSM* approaches are all more than those of time point queries. For *MR* and *MRK*, the processing time differences between time and value queries are insignificant, as the time spent on model filtering and gridding is not dominant.

6.5. Insights into KVI–Scan–MapReduce

Section 6.5.1 discusses effect of the searching depth on the performances of in-memory *iSearch*$^+$, sequential scan based and MapReduce based model filtering and gridding. At last, we will see the workload constitutions within the KVI–Scan–MapReduce paradigm in Section 6.5.2.

6.5.1. Searching depth

In Figs. 11, 12 and 13, we present the results from time and value range queries of 10% selectivity and 10% to 50% *iSearch*$^+$ depth. The percentage of *iSearch*$^+$ depth here means the ratio of

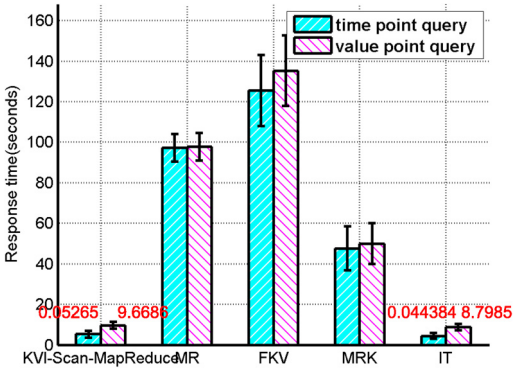


Fig. 10. Query performance of point queries.

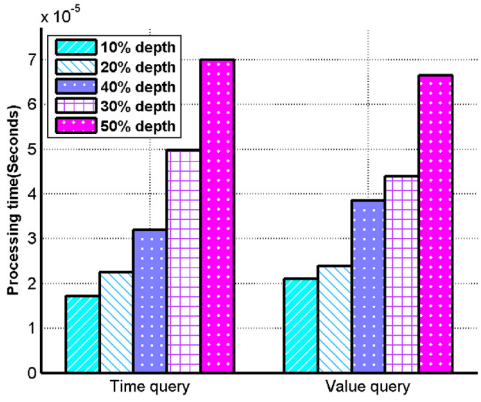


Fig. 11. Effect on *iSearch*$^+$.

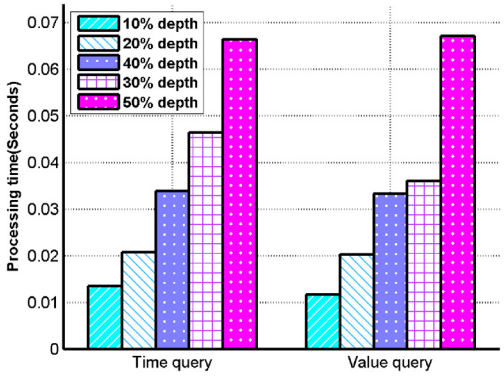


Fig. 12. Effect on sequential scan.

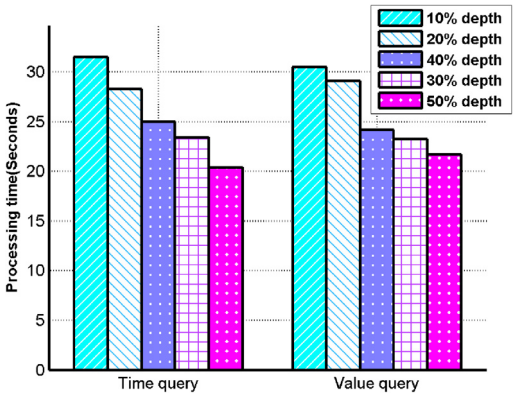


Fig. 13. Effect on MapReduce.

Please cite this article in press as: T. Guo et al., Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud, Big Data Research (2014), http://dx.doi.org/10.1016/j.bdr.2014.07.005

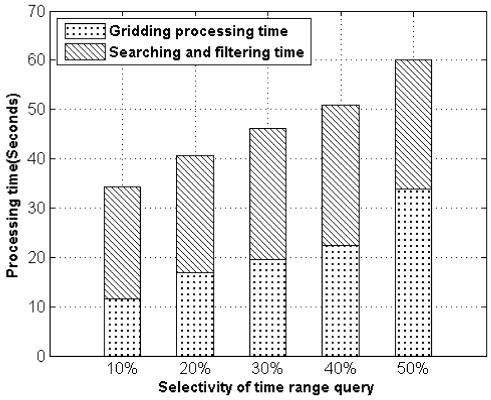


Fig. 14. Query processing time constitution of time range queries.

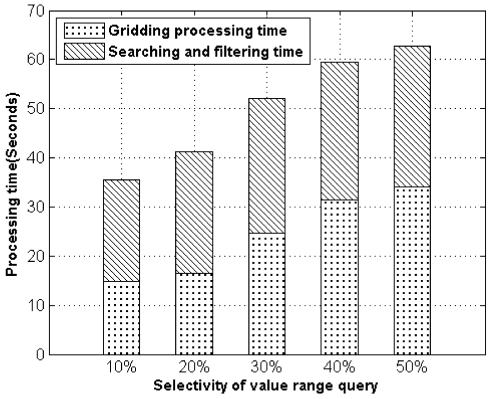


Fig. 15. Query processing time constitution of value range queries.

registration node searching depth over the height of *vs-tree* in *iSearch*$^+$. As *iSearch*$^+$ depth increases, the number of *SSs* for range scan based model processing increases. Therefore, the time consumed by *iSearch*$^+$ and range scan based model processing both increases, shown in Figs. 11 and 12. As for the MapReduce part, the deeper the level of registration node τ is, the smaller the space covered by the sub-tree rooted at τ is. Then the range of *SSs* in $\mathcal{D}$ is tighter over the range of *SSs* of query range and results in less redundant *SSs* for MapReduce, which is theoretically analyzed in Section 5.3. From Fig. 13, we can see a salient decreasing trend of MapReduce processing time.

6.5.2. Workload constitution

This experiment aims to reveal how much time the KVI–Scan–MapReduce spends on model gridding, which is a difference of model-view approach from raw sensor data approach. Figs. 14 and 15 show the results from time and value range queries of selectivity from 10% to 50%.

From Figs. 14 and 15, the time on model gridding accounts for 1/3–1/2 of the total query processing time. As the majority of gridding work is done in the reduce phase and the amount of qualified segment models sent to reducers depends on the query selectivity, the time spent on model gridding increases as the query selectivity increases. If the model gridding can adapt to users' different requirements for query results, the performance of the KVI–Scan–MapReduce scheme can be further optimized.

7. Conclusion

To the best of our knowledge, this is the first work to explore the key-value representation of an interval index for model-view based sensor data management. Different from conventional external-memory index structure with complex node merging and split mechanisms, our *KVI-index*, resident partially in memory and partially materialized in the key-value store, is easy to maintain in the dynamic sensor data generation environment. Moreover, we proposed a hybrid query processing approach, namely *KVI–Scan–MapReduce*, integrating the KVI-index, range scan and MapReduce for model-view sensor data in key-value stores. Extensive experiments in a real testbed showed that our approach outperforms in terms of query response time and index updating efficiency not only query processing methods based on raw sensor data, but also all other approaches considered based on model-view sensor data for time/value range and point queries. As a future work, we plan to explore how to process time and value composite queries and join queries based on the KVI-index.

Acknowledgement

This work is supported by the European Commission (EC), Project FP7-ICT-2011-7-287305 OpenIoT.

References

[1] S. Sathe, T.G. Papaioannou, H. Jeung, K. Aberer, A survey of model-based sensor data acquisition and management, in: Managing and Mining Sensor Data, Springer, 2013.
[2] A. Thiagarajan, S. Madden, Querying continuous functions in a database system, in: SIGMOD, 2008.
[3] A. Deshpande, S. Madden, MauveDB: supporting model-based user views in database systems, in: SIGMOD, 2006.
[4] T. Papaioannou, M. Riahi, K. Aberer, Towards online multi-model approximation of time series, in: MDM, 2011.
[5] T. Guo, Z. Yan, K. Aberer, An adaptive approach for online segmentation of multi-dimensional mobile data, in: Proc. of MobiDE, SIGMOD Workshop, 2012.
[6] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, E. Keogh, Querying and mining of time series data: experimental comparison of representations and distance measures, VLDB Endow. 1 (2008) 1542–1552.
[7] E. Keogh, S. Chu, D. Hart, M. Pazzani, An online algorithm for segmenting time series, in: Proceedings of the IEEE International Conference on Data Mining, ICDM 2001, IEEE, 2001, pp. 289–296.
[8] Y. Cai, R. Ng, Indexing spatio-temporal trajectories with Chebyshev polynomials, in: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, ACM, 2004, pp. 599–610.
[9] Q. Chen, L. Chen, X. Lian, Y. Liu, J.X. Yu, Indexable PLA for efficient similarity search, in: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB Endowment, 2007, pp. 435–446.
[10] Saket Sathe, Thanasis G. Papaioannou, Hoyoung Jeung, Karl Aberer, A survey of model-based sensor data acquisition and management, in: C.C. Aggarwal (Ed.), Managing and Mining Sensor Data, Springer US, 2013.
[11] A. Bhattacharya, A. Meka, A. Singh, Mist: distributed indexing and querying in sensor networks using statistical models, in: VLDB, 2007.
[12] A. Deshpande, C. Guestrin, S.R. Madden, J.M. Hellerstein, W. Hong, Model-driven data acquisition in sensor networks, in: Proceedings of the Thirtieth International Conference on Very Large Data Bases, in: VLDB Endowment, vol. 30, 2004, pp. 588–599.
[13] E. Keogh, K. Chakrabarti, M. Pazzani, S. Mehrotra, Locally adaptive dimensionality reduction for indexing large time series databases, SIGMOD Rec. 30 (2) (2001) 151–162.
[14] J. Shieh, E. Keogh, iSAX: indexing and mining terabyte sized time series, in: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2008, pp. 623–631.
[15] OpenTSDB, http://opentsdb.net/, 2011.
[16] T. Guo, T.G. Papaioannou, K. Aberer, Model-view sensor data management in the cloud, in: 2013 IEEE International Conference on Big Data, IEEE, 2013, pp. 282–290.
[17] T. Guo, T.G. Papaioannou, H. Zhuang, K. Aberer, Online indexing and distributed querying model-view sensor data in the cloud, in: The 19th International Conference on Database Systems for Advanced Applications, 2014.
[18] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: a distributed storage system for structured data, ACM TOCS 26 (2008) 4.
[19] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (2008) 107–113.
[20] L. George, HBase: The Definitive Guide, O'Reilly Media, Inc., 2011.
[21] C.P. Kolovson, M. Stonebraker, Segment indexes: dynamic indexing techniques for multi-dimensional interval data, SIGMOD Rec. 20 (2) (1991).

Please cite this article in press as: T. Guo et al., Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud, Big Data Research (2014), http://dx.doi.org/10.1016/j.bdr.2014.07.005

[22] G. Sfakianakis, I. Patlakas, N. Ntarmos, P. Triantafillou, Interval indexing and querying on key-value cloud stores, in: ICDE, 2013.
[23] L. Arge, J. Vitter, Optimal dynamic interval management in external memory, in: 37th Annual Symposium on Foundations of Computer Science, 1996.
[24] R. Elmasri, G.T.J. Wuu, Y.-J. Kim, The time index: an access structure for temporal data, in: VLDB, 1990.
[25] T. Bozkaya, Z.M. Özsoyoglu, Indexing valid time intervals, in: DEXA, 1998.
[26] H.-P. Kriegel, M. Pötke, T. Seidl, Managing intervals efficiently in object-relational databases, in: VLDB, 2000.
[27] H.-P. Kriegel, M. Pötke, T. Seidl, Object-relational indexing for general interval relationships, in: Advances in Spatial and Temporal Databases, Springer, 2001.
[28] S. Nishimura, S. Das, D. Agrawal, A.E. Abbadi, MD-HBase: a scalable multi-dimensional data infrastructure for location aware services, in: MDM, 2011.
[29] J. Wang, S. Wu, H. Gao, J. Li, B.C. Ooi, Indexing multi-dimensional data in a cloud system, in: SIGMOD, 2010.
[30] M.-Y. Iu, W. Zwaenepoel, HadoopToSQL: a MapReduce query optimizer, in: EuroSys, 2010.
[31] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, J. Schad, Hadoop++: making a yellow elephant run like a cheetah (without it even noticing), VLDB Endow. 3 (2010) 515–529.
[32] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, J. Schad, Only aggressive elephants are fast elephants, VLDB Endow. 5 (2012) 1591–1602.
[33] M.Y. Eltabakh, F. Özcan, Y. Sismanis, P.J. Haas, H. Pirahesh, J. Vondrak, Eagle-eyed elephant: split-oriented indexing in Hadoop, in: EDBT, 2013.
[34] H.-C. Yang, D.S. Parker, Traverse: simplified indexing on large map-reduce-merge clusters, in: DASFAA, 2009.

Please cite this article in press as: T. Guo et al., Efficient Indexing and Query Processing of Model-View Sensor Data in the Cloud, Big Data Research (2014), http://dx.doi.org/10.1016/j.bdr.2014.07.005