Research Article

An Efficient Data Analysis Framework for Online Security Processing

Jun Li¹ and Yanzhao Liu²

¹Information Security Department, Beijing Information Science and Technology University, Beijing 100192, China
²China Information Technology Security Evaluation Center, Beijing, China

Correspondence should be addressed to Yanzhao Liu; liuyz@itsec.gov.cn

Received 12 February 2020; Accepted 8 February 2021; Published 1 April 2021

Academic Editor: Roberto Nardone

Copyright © 2021 Jun Li and Yanzhao Liu. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Industrial cloud security and internet of things security represent the most important research directions of cyberspace security. Most existing studies on traditional cloud data security analysis were focused on inspecting techniques for block storedata in the cloud. None of them consider the problem that multidimension online temp data analysis in the cloud may appear as continuous and rapid streams, and the scalable analysis rules are continuous online rules generated by deep learning models. To address this problem, in this paper we propose a new LCN-Index data security analysis framework for large scalable rules in the industrial cloud. LCN-Index uses the MapReduce computing paradigm to deploy large scale online data analysis rules: in the mapping stage, it divides each attribute into a batch of analysis predicate sets which are then deployed onto a mapping node using interval predicate index. In the reducing stage, it merges results from the mapping nodes using multiattribute hash index. By doing so, a stream tuple can be efficiently evaluated by going over the LCN-Index framework. Experiments demonstrate the utility of the proposed method.

1. Introduction

Industrial cloud security service has drawn increasing attention in recent years. A wide spectrum of industrial applications and online industrial control business are using cloud-fog computing as their fundamental solution for the unprecedented amount of data problem [1–5].

Despite the successes of traditional cloud, existing traditional cloud protecting services were merely focused on designing scalable inspecting techniques for static block data in the cloud. For many emerging online industrial protection applications in the cloud, data often come in the form of multidimension continuous temp tuple streams, and it is urgent to develop scalable stream-based inspecting techniques for cloud security computing.

Example 1. Let us consider an online traffic management and inspecting system as shown in Figure 1. The essential goal of the system is to analyze security surrounding traffic information for all connected users. In this system, on the one hand, all traffic data are monitored by on-street cameras and uploaded to analysis nodes in the cloud; on the other hand, all the connected users will get online continuous surrounding traffic security information from the cloud by inspecting all queries. Note that different users may get different levels of security services based on their rules or queries.

(1) The online analysis queries may be very large and complex in the cloud. For example, in the job recommendation application, there are usually more than one million job applicants and each applicant may have more than a hundred attribute items. (2) The query set has a dynamically changing nature. For example, in the online web monitoring, web masters may need to add new queries to the query set and delete outdated queries. (3) A timely response is demanded for all queries, even though the stream data are very complex. For example, in the web monitoring application, it is often the case that the tuples have a size of more...
than 40 bytes with flowing speed faster than $10^6$ tuples per second, and the system is supposed to return all the tuples that match the monitoring queries.

In front of the above new characteristics of stream-based query in the cloud, how can we efficiently evaluate all the upcoming tuples with respect to all the registered queries? Traditionally, in data stream query systems, a centralized indexing structure will be built on a master server. After that, for each upcoming tuple, the system traverses the centralized index structure to answer the queries. However, such a centralized method cannot be used in our new problem setting. This is because the size of the indexing structure increases with the query number. Besides, there will be also a bottleneck of the system response. Therefore, in the cloud systems, we are unable to build a single centralized indexing structure to unify all the scalable and complex queries.

On the other hand, in the cloud, the conventional distributed methods are also impractical. This is because we may need to frequently register new queries (or delete outdated queries) in the cloud, which makes it very difficult to decide how much computing power should be assigned to the cloud. Conventional distributed methods have the obvious downside of lacking elasticity.

Given the limitations of existing methods, in order to solve the stream query problem in data streams, the following three challenges should be addressed: (1) Scalability: traditional data stream processing studies [6–10] usually make an assumption that the query number is no more than two thousand, whereas query number often exceeds a million in the cloud. (2) Elastic compute power: traditional distributed stream processing solutions [6–10] usually predefine the number of computing nodes, whereas it is very difficult in the cloud, because of the dynamic changing nature of query numbers. (3) Real time processing: it is necessary to process all the queries in a real time manner.

In light of the above challenges, in this paper we design a new LCN-Index data stream framework for online security analysis in the cloud. LCN-Index uses the MapReduce computing paradigm to deploy all the continuous queries. In the mapping stage, it decomposes each attribute into a batch of predicate sets which are then deployed onto a mapping node using interval predicate index. In the reducing stage, it merges the intermediate results from all the mapping nodes using multiattribute hash index. If stream processing overload is detected, the master will request more nodes from power provider. By doing so, a stream tuple can be efficiently evaluated by going over the LCN-Index framework. Experiments demonstrate the utility of the proposed method. The rest of the paper is organized as follows. Section 2 introduces the LCN-Index framework. Sections 3, 4, and 5 theoretically study the workflow, structure, and key analysis procedure of LCN-Index. Section 6 introduces the related work, and Section 7 conducts experiments and comparisons.
to demonstrate the effectiveness of LCN-Index. We conclude the paper in Section 8.

2. The Framework of LCN-Index

2.1. The Map Function of LCN-Index.

2.2. The Reduce Function of LCN-Index.

3. The Workflow of the LCN-Index Framework

The essential goal of the paper is to develop an efficient index framework that can support scalable and complex query in the cloud.

3.1. Architecture. The overall system architecture is shown in Figure 2. In an offline process, before cloud stream querying, continuous queries are decomposed and indexed. The query set decomposition module is responsible for decomposing all queries into different predicate sets according to attributes. Predicate sets are then indexed by the Mapper index builder and Reducer index builder. The Mapper index builder is responsible for building interval predicate index (LCN-Index). The Reducer index builder is responsible for building multiattribute index. Our approach works with any existing scheme for indexing interval predicates, e.g., [11, 12]. During runtime, given a coming tuple, the LCN-Index of Mapper is used to retrieve the matching predicates, and the multiattribute index of Reducer is used to retrieve all satisfied queries by merging all matching predicates. In addition, for every coming tuple, the job of Mapper Evaluator is to retrieve the matching predicates using LCN-Index. The job of Reducer Evaluator is to verify if any query can be satisfied from predicates retrieved by Mapper Evaluator. Section 3 describes the structure of LCN-Index in Mapper, whereas Section 4 describes the efficient merging algorithm in Reducer by using multiattribute index.

3.2. Workflow. Figure 2 shows the processing flow of cloud stream. More specifically, we divide query set Q into predicate sets and build LCN-Index based on segregated predicate sets. These LCN-Indexes are used by Mapper Evaluator in the map nodes (Figure 3). We also build multiattribute indexes based on the mapping information between queries and predicates. The multiattribute indexes are used by Reducer Evaluator to retrieve all satisfied queries. For every coming stream tuple, we integrate the process procedure and MapReduce model. More specially,
input tuples are first dispatched to different LCN-Indexes (Mapper Evaluator) according to attributes. The LCN-Index is used to retrieve the matching predicates. The resulting matches are shuffled according to tuple identifier and sent to Reducer Evaluator. Then, the Reducer Evaluator uses the multiattribute indexes to merge all intermediate matching results and retrieve satisfied queries. In Mapper Evaluator module, LCN-Index is an interval predicate index which is used to find all satisfied predicates for attribute/value pair: scalability, elasticity, latency.

Our driving applications are cloud stream querying applications, which have to support millions of continuous queries and billions of tuples a day. To solve scalability, we decompose query set into independent predicate sets, which make the indexes distributed into the cloud easily. To solve elasticity, Map and Reduce nodes keep performance status heartbeats with master server. If stream processing overload is detected, the master will request more nodes from power provider (1 in Figure 1). Otherwise, the master will release the underusage nodes. To solve latency, for every incoming tuple, we combine the stream querying procedure with the MapReduce [4] model. MapReduce offers the capabilities of a massive and efficient key/value process framework. As mentioned above, the key idea that speeds up every incoming item’s querying procedure is to combine MapReduce [4] model with our index framework.

Below, the content of Section 4 describes the details of LCN-Index. Section 5 describes the efficient merging algorithm of Reducer Evaluator by using multiattribute index.

4. The Index Strategy

In this section, we will first introduce the basic workflow and index schema of LCN-Index in our cloud data stream querying system in Section 4.1. Specifically, our focus is the Search and Insert operations. Therefore, we discuss the Search operation in Section 4.2. Insert and Delete operations are also introduced in the appendix.

In cloud data stream querying problem, a large scale of continuous range queries can be registered against a data stream. Usually, an efficient main memory-based index is needed, especially if the cloud stream is rapid. We propose the LCN-based index for efficient processing of continuous queries in a cloud stream environment. The LCN-Index is centered around a set of predefined virtual containment-encoded intervals. The intervals are used to decompose predicate intervals and then perform efficient Search.
operations. In fact, LCN-Index is motivated by the CEIIndex [12]. The major differences between them are as follows:

1. LCN-Index mainly has enhanced search capability, especially supporting all predicates with quality expression, whereas CEI-Index is designed to only index simple interval predicates.

2. The problem of CEI-Index [12] focuses on the queries with single interval predicate, whereas LCN-Index is designed to index complex queries where WHERE clause is a conjunction of interval predicates.

The native method for indexing the interval predicates is comparing all predicate’s boundaries maintained in the index. Time cost of this method is usually \( o(\log n) \). \( n \) is the number of predicates. In contrast, we use the indirect indexing approach to index the interval predicates. It is based on the concept of Standard Interval Unit (SIU). We predefine and label a set of Standard Interval Units. We decompose each predicate into one or more SIUs. We give every predicate a sole ID, named PredId. We insert the PredId into the ID lists associated with the decomposed SIUs. Given a target point value \( X \), the search procedure is very simple; we conduct the search procedure indirectly via the SIUs. It is needless to compare value \( X \) with any predicate boundaries. Through the predicate interval decomposition, the search result is union of all SIUs’ ID list. We prove that only a small number of SIUs cover random value \( X \). Therefore, the search time is independent of predicate number. In the rest of this paper, we assume that the query is the conjunction of predicates, and we assume that the predicate is the interval predicate of Int type. Any other type could be transferred to the Int type.

4.1. The Structure of LCN-Index. Figure 4 shows an example of local ID labeling corresponding to an interval predicate. Assume that the range of continuous attributes \( A \) is \([0, r)\). First, we partition \( r \) into \( r/L \) segments of length \( L \), where \( L \) is a power of 2. Every segment is denoted as \( S_i \), where \( i = 0, 1, \ldots, (r/L) − 1 \). Here, we assume that \( r \) is a multiple of \( L \). If not, it is easy to expand \( r \). The value range of segment \( S_i \) is \([iL, (i + 1)L]\). We treat boundaries of segments as guiding posts. For every segment, we define a \( 2L − 1 \) Standard Interval Unit (SIU) as follows:

1. Build 1 SIU of length \( L \), corresponding to the entire segment
2. Build 2 SIUs of length \( L/2 \) to partition the segment into two pieces
3. Build 4 SIUs of length \( L/4 \) to partition the segment into four pieces
4. Build the partition process until the length of every SIU is 1

For example, there is 1 SIU of length 8, 2 SIUs of length 4, 4 SIUs of length 2, and 8 SIUs of length 1. All \( 2L − 1 \) SIUs are defined to have a special relationship among them. The SIUs with length 1 are contained in SIUs with length 2, which are in turn contained in SIUs with length 4 and so on.

In this paragraph, we introduce the labeling process on these SIUs of one segment. Every SIU has a unique ID which is composed of two parts: the segment ID and the local ID. Every segment is assigned a unique ID as a global identifier among all segments. The segment ID of segment \( S_i \), where \( i = 0, 1, \ldots, (r/L) − 1 \), is simply defined as \( i + 2il \), where \( i \) is the local ID. The local ID assignment follows the labeling of a perfect binary tree. The SIU with length \( L \) is assigned to 1. The SIUs with length \( L/2 \) are, respectively, assigned to 2 and 3. Figure 4 shows the assigning process of local ID in one segment. Note that we assign \( 2L \) local IDs in every segment.

In this way, all SIUs in the same segment are organized as a perfect binary tree. The SIU with the local ID 1 is the root node of this tree, which contains two child SIUs with length of \( L/2 \), respectively; all leaf nodes have length of 1. The ID list structure in all leaf nodes not only store the ID of predicate, but also store the identifying label that indicates whether this predicate is a equality predicate. Figure 5 shows the perfect binary tree of one segment in Figure 4. The leftmost leaf node in Figure 5 shows the ID list structures. We can easily determine whether a satisfied predicate is equality predicate through the label corresponding to the predicate ID. The perfect binary tree in one segment has many efficient properties which make the Search operation in LCN-Index more efficient. We will introduce the searching and inserting algorithm in the next subsection.

4.2. The Search Operation on Mapper. The Search operation is used to efficiently find all satisfied predicates for every real time attribute/value pair coming. Algorithm 1 shows the details of Search algorithm. For every attribute/value pair \((a, v)\), where \( a \) is used to denote the attribute ID and \( v \) is the value, the Search algorithm first computes the segment ID using

\[
S_{id} = \frac{v}{L}.
\]  

(1)

Then, the algorithm uses (2) to compute the local ID of leftmost unit-length SIU.

\[
ID_l = v - S_{id} \times L + L.
\]  

(2)

Based on the property of perfect binary tree, we can simply check exactly \((k + 1)\) SIUs that overlap data value \( v \). Hence, the search results are merged into ID lists of these \((k + 1)\) SIUs. We can simply locate the \((k + 1)\) SIUs by
dividing the unit-length local ID by 2. The Search algorithm is efficient and easy. We speed up the Search algorithm by translating all complex floating points into integers. The dividing of local ID is finished by logical shifts. The Search algorithm is independent of the indexed predicate number.

Figure 6 shows an example of Search algorithm with input value $a/v$. Our algorithm first computes the local ID of the unit-length SIU that overlaps $v$. In this case, it is $S_1$ as the $k$ is set to 2. Then, we compute all local IDs of the left $k$ SIUs. In this case, they are $S_2$ and $S_1$. Last, we compute the search result by merging all ID lists of these three SIUs ($S_0$, $S_2$, $S_1$). Figure 7 also verifies that the results indeed contain $P_1$, $P_2$, $P_3$.

5. Data Stream Query Processing

In this section, we show how to merge all search results of LCN-Indexes corresponding to different attributes. In this section, we focus on the Merge algorithm in Reducer. Multiattribute index can be applied to process intermediate search results merging. Figure 2 shows the details of data stream querying workflow. For every coming stream item, we divide the item into individual attribute/value pairs. The key of every pair corresponds to the attribute contained in the stream. As shown in Figure 2, we integrate MapReduce programming model with the data stream processing, which split every stream item’s process into two phases: Map and Reduce. In the Map phase, different key-value pairs are searched in different LCN-Indexes and all intermediate results are shuffled to Reducer according to the stream ID, which causes all intermediate results of the same stream item to be processed by the same Reducer. In the Reduce phase, we merge the results of intermediate search into Reducer. Given a stream item, to guarantee the completeness of the result, we need to retrieve all possible predicate IDs stabbed by any attribute value in Mapper. To achieve this goal, efficient algorithms are proposed to merge the predicate result into Reducer. In this section, we focus on the efficient merging algorithm that is defined in Reduce function. In the rest of this paper, we assume that the input values are different search results of LCN-Indexes. Through the shuffling scheme of MapReduce, all intermediate search results with the same stream ID could be dispatched into the same Reducer. In Section 5.1, we will introduce the scheme of multiattribute index which is used to efficiently merge all intermediate search results of LCN-Index on Mappers. The details of efficient merging algorithm will be introduced in Section 4.2.

5.1. The Scheme of Index in Reducer. The two most important schemes in the Reduce are the following: (1) selecting most common equality or inequality predicates as the trigger predicates; (2) building multiattribute index based on all these trigger predicates and mapping all these predicates with queries. More precisely, given a set of queries $Q$ and the attribute set contained in the stream $C$, the attributes in $C$ can be divided into two classes: (1) discrete and (2) continuous. First, we select all predicates of discrete attributes as predicate set. Then, we cluster all predicates into different sets according to the attribute and its popularity. In the end, by using multiattribute hashing function, we build indexes based on these predicate sets. For every coming intermediate result from Mapper, merging incurs a lookup per hash table of the multiattribute indexes to find the trigger predicates.

We consider trigger predicates defined as a conjunction of equality or inequality predicates. A trigger predicate is defined by a pair $<\text{identifier}, \text{predicate}>$, where $\text{identifier}$ is an identifier and $\text{predicate}$ is a set of e-quality or inequality predicates which are pairwise different over their attributes. The set of attributes occurring in the $\text{predicate}$ is called Hash Combination. Let $TP$ be a set of access predicates. In order to test these predicates against incoming events of stream item we use multiattribute hashing function to build indexes. Each index is intended to check trigger predicates having a certain schema. More precisely, a multiattribute index over a set of predicates is defined by a pair $<A, h>$, where $A$ is a set of attributes that have equality predicates and $h$ is a hash function which takes the coming event and returns the trigger predicates entry.

5.2. The Merging Algorithm in Reducer. The Reduce phase is responsible for merging all intermediate search results from Mappers. The merging algorithm in Reducer uses a set of multiattribute indexes, a predicate result bit vector, an event list, and a vector of references to queries cluster lists, called a query cluster. The data structures that are used in merging algorithm are depicted in Figure 8. The multiattribute index is used to compute the set of queries satisfied by a given incoming search result from Mapper. We build multiattribute index based on the Distribute attribute whose predicates are registered with equality expression. Equality predicates shared among one or more queries are selected to be inserted into the multiattribute index. We call all these equality predicates trigger predicates. A trigger predicate $p$ is associated with a list of queries clusters. When triggering an equality predicate, we need to check every query in the queries clusters associated with the trigger predicate. The multiattribute index is used in the merging algorithm which will be introduced in the next subsection. The predicate result bit vector is used to record the result of all predicates. The query cluster is used to check all satisfied queries sharing the same predicate. At the Reduce phase, we first build multiattribute index and deploy it in Reducer. Then, for every coming intermediate search result from Mapper, we check its attribute ID and search in the multiattribute index.
to check all query clusters. If all predicate results of a query in query cluster are true, we say this query is matched and output its ID.

Figure 8 provides a detailed description of a query cluster for queries having the same equality predicate $p$. A query cluster is a vector of a collection of query structures. The query structure of each query is organized as follows: it consists of a collection of all predicate results and a bit denoting the query identifier. Entry $[i,j]$ of the query cluster contains a bit vector reference to the $i$th predicate of the $j$th query in query cluster. If all bit vector entries referenced at column $j$ are true, we say the $j$th query in query cluster is true. The most important problem is how to merge all intermediate results from Mappers. Algorithm 2 gives details of the merging process of all attribute/value pairs. As described above, the multiattribute index is built based on all equality predicates of discrete attributes. The discrete attribute/value pairs of one stream item are directly dispatched to Reducer node according to their stream ID. The discrete value pair is searched in the multiattribute indexes to trigger predicate inspection. Algorithm 2 shows the details of merging all predicate results from Mappers. We denote every coming predicate search result as an event $e$. The merging algorithm is executed each time a new intermediate search result comes in. First, the predicate result bit vector is initialized to “false.” Then, the merging algorithm begins a two-step procedure. The first step uses the multiattribute indexes to compute the satisfied trigger predicates, the algorithm sets to true all corresponding bits in the predicate result bit vector. We say event $e$ satisfies a query $q$ if the status of every predicate in $q$ is satisfied after trigger event $e$ coming by. Therefore, the merge result problem is as follows: given a set of predicate search result events $e$ and a set of queries $Q$, find all queries that satisfied the event set. The algorithm data structures are depicted in Figure 6. Recall that a query $q$ is defined by an ID and a set of predicates. An event is an instance of $e$. Algorithm 3 shows the whole procedure of Reduce function. Firstly, we use the relationship between query and predicates to cluster all queries for every trigger predicate. A predicate $p$ may also be associated with a reference to a list of query clusters. We say predicate $p$ is a trigger predicate for all queries in query clusters lists of $p$. We guarantee that queries in cluster list triggered by $p$ need to be checked if and only if $p$ is satisfied. Inside the cluster list, queries are grouped into query clusters by size. Secondly, for every coming event $e$, we execute the result merging algorithm. The result merging algorithm first inspects the event ID; if it is the first predicate search result of a stream item, we allocate a data structure just like in Figure 7 for this new stream tuple and initialize all bit vectors of predicates to 0. Thirdly, for current trigger predicate ID, we inspect all queries in the associated query clusters. If we find all predicates of any query result true, then we add the ID of all these queries to the output (Algorithms 4 and 5).

6. Related Work

This paper complements ideas developed in cloud data management, Publish/Subscribe systems, and data stream querying.

6.1. Cloud Data Management

In the cloud in [2, 3] proposed a data management system called epIC to build scalable data storage system. However, our work is different from theirs: our work focuses on the cloud stream querying problem where large scale queries are continuous, whereas their work [2, 3] focuses on analytical jobs on large scale dataset in the cloud. Recently, there exist some works on cloud stream processing, e.g., [13, 14], while [13] tried to combine the MapReduce with the stream processing in IBM’s system S. They propose the DEDUCE, as a new middleware to support MapReduce model. In the DEDUCE, they provide language to support cutting the stream processing data-flow into MapReduce procedures. However, their work is different from ours. In particular, our work focuses on indexing scalable continuous queries to speed up the stream’s querying procedure, whereas DEDUCE focuses on cutting simple workflow into MapReduce procedures. Reference [14] proposes a new processing framework to support large scale data streams in the cloud, while focusing on how to split the queries to support parallelization instead of indexing these scalable queries.

6.2. Publish/Subscribe Systems

Publish/Subscribe systems are an active area of research [11, 15]. Subscriptions expressing subscribers’ interest in events are continually evaluated against publications representing events. The approaches are distinguished by the data formats they process and by the algorithmic design. Common among the approaches is the determination of a match based on the publication processed. A Boolean expression uses two types of primitives: $\epsilon$ and $<$ predicates, and queries in Publish/Subscribe systems are often a disjunctive normal form (DNF) or conjunctive normal form (CNF) of Boolean expressions, which is different from our queries with expression $SELECT \ast \quad FROM \ast \quad WHERE \ast$.

6.3. Data Stream Querying

For each up-to-date stream record, data stream querying model traverses all its continuous queries to verify the record’s key/value pairs. Reference [7] aims to speed up the join operator for every incoming stream item; in their STREAM system, they always assume that the number of queries does not exceed 2000, so it is impractical to use their methods to solve cloud stream querying problem. However, a line of indexing methods during the last few decades have been proposed to index texts, images, and microclusters on data streams for anytime query and clustering, e.g., [12, 16–19]. In particular, [16, 17, 19] focus on multidimension index strategy, and [12] focuses on only interval index on one single attribute. However, none of the existing works considers the problem of indexing for scalable cloud stream querying. Our work can be taken as a pioneer work on this direction.

7. Experiments

In this section, we will conduct extensive experiments on both synthetic and real world data streams to evaluate the performance and scalability of our index framework for each
up-to-date stream record in the cloud. Our testing infra-
structure includes 16 Hadoop machines which are con-
ected together to simulate cloud computing platforms. The
communication bandwidth between nodes is 1 Gbps. Each
machine has a 3.00 GHz Intel Core 2 CPU, 4G memory, and
a 500G disk. Machines ran Red Hat application server 5.2
OS. Different sizes of cloud computing systems can be
simulated by our infrastructure. We conducted 10 simul-
ation experiments, ranging from 100 nodes to 1500 nodes.
Each time, 100 nodes are considered to be added to the cloud
computing system. In our index framework, we use one
machine to play the role of master and dispatch different
attribute/value pairs of cloud data stream. Each of the other
15 machines simulates 100 to 1500 nodes.

7.1. Benchmark Data. In order to test the efficiency of the
LCN-Index framework, we used three real world datasets
crawled from Internet. Table 1 lists the information of
datasets. In particular, the stock dataset is crawled from
stock-analysis websites (http://www.econ.yale.edu/shiller/
data.htm), which is used to simulate cloud stock stream
monitor applications. The spam detection and malicious
URL detection datasets are crawled from application-level
routers to simulate cloud web traffic monitoring applica-
tions. All of our queries are generated using Zipf distri-
bution, which is well known as a good fit for keyword
popularity in text-based searches. With Zipf distribution, the
popularity of the i-th most popular predicate is inversely
proportional to its rank i, i.e., $p_i \propto \frac{1}{i^\theta}$. The query number of
all three real world datasets is 1000000.

7.2. Benchmark Methods. For comparison purpose, we
implement a distributed R-Tree index described in [17]. We
use "DistributeRTree" to denote the index. In the Distrib-
uteRTree, every query is taken as a multidimensional matrix
and inserted into the DistributeRTree. The DistributeRTree
maintains a large R-Tree over the network. However, the
search cost of DistributeRTree may be much higher for
stream processing when we insert more queries. Therefore,
the total number of queries of DistributeRTree is less than
that of our system.

7.3. Measurements. Two important measurements will be
used here. (1) Time cost. By using an index framework that
integrates LCN-Index and MapReduce to divide the query
index according to attributes, LCN-Index is supposed to
achieve a much lower computation overhead, whereas the
DistributeRTree is based on the idea of conventional data
stream processing procedure, which is supposed to achieve a
much higher cost than LCN-Index. (2) Scalability. As the
framework of LCN-Index divides the query set into different
dedicate sets according to attributes, LCN-Index supports
more scalable queries than DistributeRTree.

7.4. Experimental Results. We compared the two index
strategies under different parameters. For example, different
query numbers in, different nodes, different attributes,
different L, and different query width w. Unless otherwise
mentioned, the parameters are set as follows. The default
query number is set to 100000. The default node number is
set to 100.

7.4.1. The Impact of Query Number. Figure 9 shows per-
formance comparison with DistributeRTree index strategy
under different query scale on real world data stream sets.
Query number is the most important parameter to evaluate
the performance. By comparison on these datasets, our
LCN-Index always outperforms the DistributeRTree. When
the query scale is enlarged, throughput of DistributeRTree
obviously degrades as the property of R-Tree search cost.
In LCN-Index, a stream item is processed by different Mapper
in parallel, while in the DistributeRTree, we cannot apply the
parallel Search algorithm, because the query is taken as
matrix and randomly distributed in the node cloud.
Therefore, it is obvious that LCN-Index will significantly
support the large scale continuous queries.

7.4.2. The Impact of Attributes. For cloud data stream que-
rying problem, the Reducer of LCN-Index only indexes equality
or inequality predicates associated with the Distribute attribute,
so the problem is the impact of discrete attributes and con-
tinuous attributes. To answer this question, we conducted a
series of experiments with different attributes numbers. Figure 7
shows the impact of attribute number. From the results, we can
observe that the LCN-Index framework will degrade as we
increase the discrete attributes, because we need to do more
index scanning in the merging algorithm of the Reduce phase,
whereas the number of continuous attributes does not impact
the performance of LCN-Index, because our LCN-Index ab-
solutely supports parallelization.

7.4.3. The Impact of Query Width w. To investigate whether
the width of query impact LCN-Index efficiency, we com-
pare the LCN-Index and DistributeRTree framework with
some famous standard datasets. From Figure 10, we can
come to important conclusions: (1) LCN-Index can significa-
tically reduce the stream querying cost. For example, in the
spam detection data stream, when the width w equals 7,
LCN-Index needs 487146 ms to process a stream item, while
the DistributeRTree needs 1397197 ms. (2) As we increase
the width w, the querying costs of LCN-Index and Dis-
tributeRTree both increase, but the cost of DistributeRTree
increases more quickly than that of LCN-Index. This is
because the LCN-Index of one Mapper only maintains
predicates of query set that belong to the same attribute, and
the increasing width w only impacts the merging algorithm
in Reducer. In contrast, as the width increases, Distrib-
uteRTree’s cost increases faster because more nodes have to
be split and deployed in the network, which makes the
Search algorithm more costly.

7.4.4. The Impact of L. In this part, we compare the impact
of L on three datasets. We use L to denote the length of
segment in LCN-Index. All LCN-Indexes are built and
Algorithm 1: The total execution procedure of LCN-Index.

*Input:* query set \( Q \), stream \( S \)

Step 1: partition \( Q \) into a batch of predicate sets \( P \).

Step 2: build index on \( P \).

Step 3: deploy these indexes on Mapper and Reducer

Step 4: for every incoming tuple \( t \leftarrow \text{stream } S \)

while \( t \neq \text{empty} \) do

(1) Search in the interval index in Mapper;

(2) Merge all results from Mapper and output satisfied queries;

\[ P = \text{LCN-Index.search}(v); \]

\[ \text{// search } v \text{ in LCN-Index;}; \]

Figure 6: Example of LCN-Index Search operation; search is performed via the SIUs.

![Figure 6: Example of LCN-Index Search operation; search is performed via the SIUs.](image)

Figure 7: The impact of number of attributes.

(a) The number of discrete attributes

(b) The number of continuous attributes

- Stock
- Intrusion
- Spam
As we increase $L$, the total index storage cost decreases. This is because more predicates are stored in lesser SIUs. The search time increases as $L$.

**Algorithm 2:** The map function of LCN-Index.

**Algorithm 3:** The Reduce function of LCN-Index.

**Algorithm 4:** The Search operation of LCN-Index.

**Algorithm 5:** The merging algorithm in Reducer.
becomes bigger. This is because we need to inspect more SIUs’s list when \( L \) increases. From the results, we can observe the following: LCN-Index performs better than DistributeRTree. For example, in the Syn-10 dataset, LE-Tree is nearly three times faster than DistributeRTree. Therefore, we can safely say that, compared to DistributeRTree framework, LCN-Index is more suitable for cloud data stream querying. It is obvious that the performance of LCN-Index, by indexing the query according to attribute to support parallelization, will be more scalable than that of DistributeRTree which simply distributes the R-Tree in the network for processing.

### 8. Conclusions

Industrial cloud security is a new challenge. This paper presents a new elastic cloud data analysis system that supports scalable multidimension continuous queries inspecting. In online data analysis framework, we proposed a new indexing schema to efficiently process every incoming online data tuple. The key idea of the data analysis framework is to integrate MapReduce model and industrial communication tuple filtering procedure. Experiments on both synthetic and real world industrial streams show that our online data analysis framework is efficient, elastic, and scalable.
Data Availability
No data were used to support this study.

Conflicts of Interest
The authors declare that they have no conflicts of interest.

Acknowledgments
This work was supported by the Joint Funds of the National Natural Science Foundation of China (Grant no. U1936111).

References
[1] D. Abouzeid, K. Bajda-Pawlowski, and A. S. Hadoopdb, “An architectural hybrid of map reduce and DBMS technologies for analytical workloads,” in Proceedings of the VLDB, Lyon, France, August 2009.
[2] J. Wang, S. Wu, H. Gao, J. Li, and B. Chin Oo, “Indexing multi-dimensional data in a cloud system,” in Proceedings of the SIGMOD, Indianapolis, IN, USA, June 2010.
[3] S. Wu, D. Jiang, B. Chin Ooi, and K.-L. Wu, “Efficient b-tree based indexing for cloud data processing,” in Proceedings of the VLDB, Singapore, September 2010.
[4] J. Dean and S. G. Mapreduce, “Simplified data processing on large clusters,” in Proceedings of the OSDI, San Francisco, CA, USA, October 2004.
[5] P.-A. Chaiken, B. Jenkins, and J. Zhou, “Scope: easy and efficient parallel processing of massive data sets,” in Proceedings of the VLDB, Auckland, New Zealand, August 2008.
[6] R. Avnur and J. H. Eddies, “Continuously adaptive query processing,” in Proceedings of the SIGMOD, Dallas, TX, USA, May 2000.
[7] S. Babu and J. W. Streamon, “An adaptive engine for stream query processing,” in Proceedings of the SIGMOD, Paris, France, June 2004.
[8] J. W. Chris Olston and J. Jiang, “Adaptive filters for continuous queries over distributed data streams,” in Proceedings of the SIGMOD, San Diego, CA, USA, June 2003.
[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, “Niagracq: a scalable continuous query system for internet databases,” in Proceedings of the SIGMOD, Dallas, TX, USA, May 2000.
[10] A. R. Chen Liu and S. Parthasarathy, “Near-optimal algorithms for shared filter evaluation in data stream systems,” in Proceedings of the SIGMOD, Vancouver, Canada, June 2008.
[11] S. Whang and H. Molina, “Indexing boolean expressions,” in Proceedings of the VLDB, Lyon, France, August 2009.
[12] K. L. Wu and P. S. Yu, “Interval query indexing for efficient stream processing,” in Proceedings of the CIKM, Washington, DC, USA, November 2004.
[13] K.-L. Vibhore Kumar and H. Andrade, “Deduce: at the intersection of map reduce and stream processing,” in Proceedings of the EDBT, Lausanne, Switzerland, March 2010.
[14] M. P. Vincenzo Gulisano and R. Jimenez-Peris, “Streamcloud: a large scale data streaming system,” in Proceedings of the ICDCS, Genova, Italy, June 2010.
[15] A. Machanavajjhala, E. Vee, M. Garofalakis, and J. Shanmugasundaram, “Scalable ranked publish/subscribe,” in Proceedings of the VLDB, Auckland, New Zealand, August 2008.
[16] P. Ciaccia, M. Patella, P. Zezula, and M-tree, “An efficient access method for similarity search in metric spaces,” in Proceedings of the VLDB, Athens, Greece, August 1997.