2015 IEEE 35th International Conference on Distributed Computing Systems Workshops

A Scalable Distributed Private Stream Search System

Peng Zhang*†, Yan Li‡, Qingyun Liu*† and Hailun Lin*
*Institute of Information Engineering, Chinese Academy of Sciences
†National Engineering Laboratory for Information Security Technologies
‡National Computer Network Emergency Response Technical Team Beijing, China
{pengzhang, liuqingyun}@iie.ac.cn, liyan@cert.org.cn

***Abstract*—With the coming of the era of big data, people are more concerned about data privacy. On the one hand, the users are more eager for fresh and low-latency search results than ever before. On the other hand, they do not want to open the search criteria. To this end, this paper proposes a scalable distributed private stream search system, in which the search criteria is hidden by homomorphic encryption technique with three buffers. Most importantly, the system adopts shared-nothing architecture to support the horizontal scalability, and partitions the stream into segments to achieve parallel query and bitmap index-based storage. Experimental results show the effectiveness and efficiency of our method on private stream search.**

***Keywords*—private stream search; paillier cryptosystem; bitmap index; big data**

I. Introduction

Big data refers to the massive data generated in the cyberspace and available on the Internet. It has a few typical features: such as volume, variety and velocity [1]. Therefore, with the coming of the era of big data, most of the applications must provide users with fresh, low latency search results. However, the disk access is difficult to meet the requirement. From 1980s to 2009, the maximum transmission rate of disk is improved by 50 times, but the access delay of disk is only improved by 2 times. If it is measured by capacity/bandwidth (Jim Gray rule [2]), the access delay seems worse. As a typical representative of low-latency and high throughput, the memory access can speed up the query exponentially. Therefore, it can be an excellent solution to implement efficient query for these applications. However, when large-scale datasets have to be loaded in memory, it will cost a lot of money. But with the decrease of the memory price, memory access gradually becomes available. Nowadays, many organizations have proposed excellent query solutions to make full use of memory, such as MapUpdate [3], D-Streams [4], RAMCloud [5] and Spark [6]. However, the user might want to protect the privacy of his queries for a variety of reasons ranging from personal privacy to protection of commercial interests. So how to implement the private search is a new challenge.

Different from searching on encrypted data, for private search, the data is unencrypted but the search query is encrypted [7]. Actually, private search is most closely related to the single-database private information retrieval(PIR) [8]. The incompatibility between previously proposed PIR schemes and the private search is that PIR schemes have thus far required communication dependent on the size of the entire database rather than the size of the portion retrieved. In some streaming settings, a private search scheme with communications independent of the size of the stream or database is desirable. Another difference between the PIR and private search settings is that most PIR constructions model the database to be searched as a long bitstring and the queries as indices of bits to be retrieved. In contrast, the private search allows queries based on a search for keywords within text.

Nowadays, the hot topics of private search are about how to model encrypted query to improve the capability of semantical security (e.g. [9], [10]), but the scalability of private search is ignored. For this purpose, we propose a scalable distributed private search system, where the query is encrypted through Paillier encryption system [11]. Moreover, the system adopts shared-nothing architecture to support the horizontal scalability, and partitions the stream into segments to achieve parallel query.

In general, the main contributions of this paper are two-fold:

- We propose to use segment-based scalable distributed query. Specifically, our method partitions the stream into segments and implements the distributed query and bitmap index-based storage, which improves the query performance and scalability.
- We propose to exploit homomorphic encryption technique with three buffers to implement private search.

The remainder of this paper is organized as follows. In Section II, we give a review of related work. Section III presents an overview of the architecture, including data model and stream query. Section IV is devoted to the experimental results. Finally, the paper is concluded in Section V.

II. Related Work

The problem of private search on streaming data was first introduced by Ostrovsky and Skeith [12]. They gave two solutions. One is based on the Paillier cryptosystem [11] and allows to search for documents satisfying a disjunctive condition $k_1 \vee k_2 \vee \ldots \vee k_{|K|}$, i.e., containing one or more classified keywords. Another is based on the Bonehet cryptosystem [13] and can search for documents satisfying $(k_{11} \vee k_{12} \vee \ldots \vee k_{1|K_1|}) \wedge (k_{21} \vee k_{22} \vee \ldots \vee k_{2|K_2|})$, an AND of two sets of keywords. This paper also gave a solution to search for documents satisfying a condition $k_1 \vee k_2 \vee \ldots \vee k_{|K|}$. Like the idea in [12], an encrypted dictionary is used. However, rather than using one large buffer and attempting to avoid collisions like [12], This paper stores the matching documents in three buffers and retrieves them by solving linear systems. Yi et al. [14] proposed a solution to search for documents containing more than *t* out of *n* keywords, so-called (*t*,*n*) threshold searching, without increasing the dictionary size. The solution is built on the state of the fully homomorphic encryption technique and the buffer keeps at most *m* matching documents

978-1-4673-7303-6/15 $31.00 © 2015 IEEE
DOI 10.1109/ICDCSW.2015.31

without collisions. Searching for documents containing one or more classified keywords like [12], [7] can be achieved by (*1*,*n*) threshold searching. However, the existing solutions for private search on streaming data have not considered scalability, for this purpose, this paper proposes a scalable distributed private search system, and adopts shared-nothing architecture and encryption technique with three buffers to support private search under different input load.

III. The Architecture

In our system, the fundamental storage unit is segment, and each table is divided into a collection of segments, where each segment contains about 10 thousand lines, some of them are shown in Table I. The system simplifies the data distribution, storage and queries with a timestamp column, and partitions data sources into well defined time intervals, typically an hour or a day, and may further partition according to values from other columns to achieve the desired segment size. The segment's identifier is composed of data source identifier, the time interval of the data, a version string that increases whenever a new segment is created, and a partition number. The metadata of segment is used for concurrency control, read operation always access to the data in the segments with the latest version identifier for the time range.

In our system, most of segments are unchangeable historical segments. These segments are stored permanently in a distributed file system, such as S3 or Hadoop Distributed File System(HDFS) [15]. All historical segments have their metadata to describe their attributes such as the size, the compression format and the storage location. The historical segment can be updated through the creation of a new historical segment that obsoletes the older one. The segment covered very recent intervals is a changeable real-time segment. The real-time segment is incrementally updated after new data are injected, and can support query during incremental indexing process. The incremental indexing only works by calculating the aggregate value of the interesting metric (e.g. summary of impression and revenue in Table I). This often brings an order of magnitude compression without sacrificing the numerical accuracy. Of course, this is at the cost of not supporting queries over the non-aggregated metrics.

A. Data Query

Figure 1 depicts the architecture of our proposed system. The query involves the following types of nodes: historical compute node, real-time compute node, broker node and coordination node. Each node performs a specific function. Specifically, the real-time compute node is responsible for data injection, storage and responses to queries for the most recent data. Similarly, the historical compute node is responsible for loading and responding to queries for historical data. Data in our system is stored in the storage node. The storage node may be a historical compute node or a real-time compute node. A query will firstly be sent to the broker node, which is responsible for finding and routing the query to the storage nodes containing related data, the storage nodes execute their portion of the query in parallel and return the results to the broker node, then the broker node receives the results and merges them, and finally returns the final result to the users. The broker node, compute node and real-time compute node are considered as queryable nodes. In addition, our system also has a coordination node to manage the segment assignment, distribution and replication, but the coordination node is unqueryable node, it is mainly used to maintain the stability of the cluster. The coordination node depends on the external MySQL database and the Apache Zookeeper [16] to achieve coordination. Even though the query is forwarded via HTTP, intra-cluster communication is over Zookeeper.

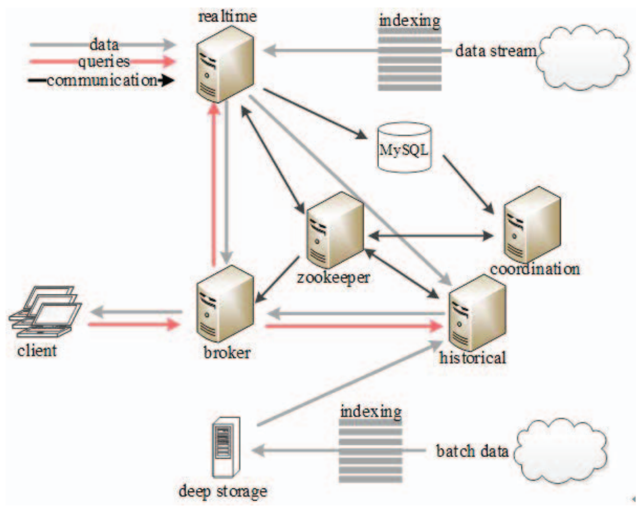

Fig. 1. An overview of the system architecture

1) Historical Compute Node: The historical compute node is the main worker of our system and does not depend on external components. It loads historical segments from a permanent storage and make them queryable. Since historical compute nodes do not know each other, there is no competition of single point between the nodes. The historical compute node only needs to know how to perform its assigned tasks. To help other service discovery historical compute nodes and the segments they provide, each historical compute node maintains a connection with the Zookeeper. The historical compute node creates a temporary node under specifically configured Zookeeper paths to publish its online status and served segments. A historical compute node loads new segments or drops existing segments by creating a temporary znode directory under a special "load queue" path associated with the historical compute node.

Figure 2 shows a simple interaction of three historical compute nodes and Zookeeper. Each historical compute node has an associated load queue path. When a historical compute node comes online, it will publish its served segments in the path. In order to make the segment queryable, a historical compute node must firstly possess a local copy of this segment. Before a historical compute node starts to download a segment from HDFS, it firstly checks the local disk directory (also known as cache) to determine whether this segment has been in the local storage. If the cache information of this segment does not exist, then the historical compute node will download metadata of this segment from Zookeeper. The metadata includes information about where the segment is located in HDFS and how to decompress and process the segment. Once the historical compute node completes this process of this segment, it will publish the status that it can serve this segment in Zookeeper. At this moment, this segment is queryable.

TABLE I. THE SEGMENT EXAMPLE

Timestamp	Publisher	Advertiser	Gender	Country	Impressions	Clicks	Revenue
2014-01-01 T01:00:00Z	sina.com	baidu.com	Male	China	1800	25	15.70
2014-01-01 T01:00:00Z	sina.com	baidu.com	Male	China	2912	42	29.18
2014-01-01 T01:00:00Z	yahoo.com	google.com	Male	USA	1953	17	17.31
2014-01-01 T01:00:00Z	yahoo.com	google.com	Male	USA	3914	170	34.01

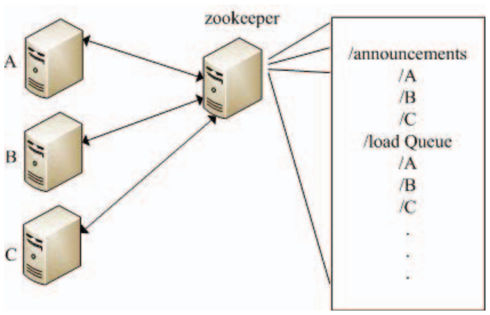

Fig. 2. The interaction of three historical compute nodes and Zookeeper

2) Real-Time Compute Node: The real-time compute node encapsulates the functions of real-time data stream injection and query. Data indexed via real-time compute node can be queried immediately. The real-time compute node consumes data, so it needs a corresponding producer to provide data. Typically, for the purpose of data persistence, a message queue, such as Kafka [17], should be placed between the producer and the real-time compute node, as shown in Figure 3.

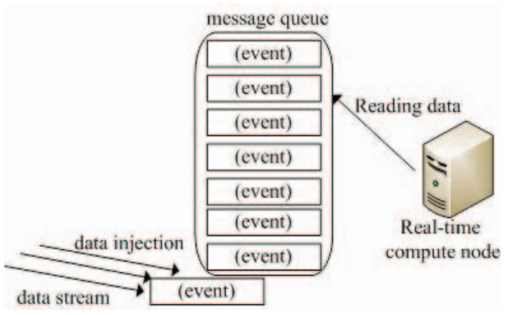

Fig. 3. An example of message queue

As shown in Figure 3, a message queue can be regarded as a buffer for incoming data stream. The message queue can maintain offsets indicating the location that the real-time compute node has read to and the real-time compute node can periodically update this offsets. The message queue can also be seen as a backup storage for recent data stream. The real-time compute node injects data by read message from the message queue. The time from message creation to the message queue storage to message consumption may be about hundreds of milliseconds. The real-time compute node maintains an in-memory index buffer for all injected messages. When new message is injected into the message queue, these indexes are incrementally created and can also be directly queried. The real-time compute node persists periodically these indexes into disk.

After persist, a real-time compute node uses the offset of last message of the most recently persisted index to update the message queue. Each persisted index is unchangeable. If a real-time compute node fails, it starts to recover, which just needs to reload any index which has been persisted to disk and then reads the message queue from the point which the last offset is committed. Periodically committing offsets can reduce the amount of re-scanned data after a real-time compute node fails. The real-time compute node maintains a comprehensive view of current index being updated and of all index persisted to disk. This comprehensive view allows all indexes on a node can be queried. Periodically, the real-time compute node will assign a background task to search all persisted indexes of a data source. This task builds a historical segment while merges all indexes. The real-time compute node will upload this segment to HDFS and simultaneously provide a signal to the historical compute nodes to indicate the segment could be queried. When a real-time compute node transforms a real-time segment into a historical segment, there is no data loss.

Figure 4 shows the real-time compute node data persistence process. Similar to the historical compute nodes, the real-time compute node also publishes segments in the Zookeeper. Unlike the historical segments, the real-time segments can represent a period of time that extends to the future. For instance, a real-time compute node publishes a segment is being serving, which contains the data for the current hour. Before the end of this hour, the real-time compute node will collect data all the time.

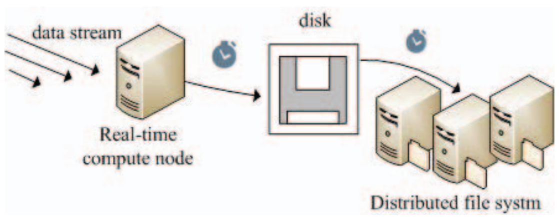

Fig. 4. Real-time compute node data persistence process

For example, every 10 minutes, the real-time compute node will refresh and persist the memory index to the disk. At the end of the current hour, the real-time compute nodes prepare to provide data for the next hour by creating a new index and publishing a new segment for the next hour. The real-time compute node does not immediately merge and build a historical segment for the previous hour until after some window times have passed. With a window time, the real-time compute node can disperse the data points to come and reduce the risk of data loss. At the end of this window time, the real-time compute node will merge all persisted indexes, and build a historical segment for the previous one hour, and then send the historical segment to historical compute nodes to serve. Once the segment on the historical compute node can be queried, then the real-time compute node will delete all the information of this segment and publish it will never serve this segment.

The real-time compute node is highly scalable. If the injection rate of a given data source exceeds the maximum capacity of the real-time compute node, additional real-time compute nodes will be added. Multiple real-time compute nodes simultaneously consume the data from the same data stream, and each real-time compute node is only responsible for a part of the data source. This naturally creates partitions across nodes. Each real-time compute node publishes the real-time segment it is serving and each real-time segment has a partition number. The data from all real-time compute nodes will be merged at the broker node.

3) Broker Node: The broker node functions as query router, it can route query to the queryable node, such as the historical compute node and real-time compute node. The broker node gathers the metadata published in Zookeeper about what segments exist and where the segments. The broker node is also responsible for merging the query results from each node, before returning final result to the users. In addition, the broker node provides data persistence layer through maintaining a cache for recent results. When multiple compute nodes fail and all copies of a segment are lost, if the information has already been stored in the cache, the segment results can still be returned.

In order to send the query to the correct compute node, the broker node builds a global view from the information in Zookeeper. The system uses Zookeeper to maintain all information of the historical compute nodes and real-time compute nodes in a cluster and the information of segments these nodes are serving. For each data source in Zookeeper, the broker node builds a timeline of the segments for the data source and the nodes serving them segments. A timeline is composed of segments, and represents which segments contain data in which time range. The system may contain multiple segments that have the same data source and time interval, but have different version number. The timeline view always presents the segment with the latest version number for a time range. If the intervals of two segments overlap, the segment with the latest version has higher priory. When a query of a specified data source and time interval is received, the broker nodes will perform a lookup on the timeline associated with the data sources and time interval, and search the segments that contain data for the query. The broker node finds the related compute nodes for these segments by mapping, and then forwards the query to these nodes.

4) Coordination Node: The coordination node is primarily responsible for the management and distribution of segments (including loading new segments, dropping outdated segments), and the management of the replicated segments and load balancing of segments. The coordination node periodically checks the current status of the cluster. At running time, the coordination node compares the expected state of the cluster and the actual state of the cluster to make decision. The coordination node maintains a Zookeeper connection to obtain information of all nodes in the cluster. Meanwhile, the coordination node also maintains a MySQL database connection to get the information of operational parameters and configuration.

An important piece of information located in the MySQL database is the segment table, which contains all historical segments that should be served. This table can be updated by any service creating the historical segments. MySQL database also contains a rule table to manage how segments are created, destroyed and replicated in the cluster. When the coordination node is ready to assign work to a compute node, it does not communicate with the compute nodes directly, instead, the Zookeeper node creates a temporary znode directory, which contains the information about what the compute node should do. Each compute node maintains a connection with Zookeeper to listen new assigned work.

B. Data Storage

The system adopts column-oriented storage format. Column-oriented storage could make the CPU more efficient as a result of only needed data are loaded and scanned. Now, the system supports different column types. According to these types, the system can reduce the cost of storing a column on memory and disk by using different compression methods. As shown in Table I, the publisher, advertiser, gender and country column contains only are strings. String column is dictionary encoding. Dictionary encoding is a common method to compress data. For data shown in Table I, we can map each publisher into a unique integer identifier as follows:

$$sina.com \rightarrow 0$$
$$yahoo.com \rightarrow 1$$

The mapping transforms the publisher column as an integer array, and the array indices represent the rows of raw data set. For the publisher column, we can transform publishers as follows: [0, 0, 1, 1]. The integer array of this result is very suitable for compression. The generic compression algorithms based on encoding are very common in column-oriented storage. In the system, we use the LZF compression algorithm [18]. Similar compression methods can be applied to the numeric columns. For example, the clicks and revenue column in the table can be transformed into an array, respectively:

$$clicks \rightarrow [25, 42, 17, 170]$$
$$revenue \rightarrow [15.70, 29.18, 17.31, 34.01]$$

In this case, we compress the original value instead of the encoded dictionary representations. In addition, the system creates additional indices for the string column to support any filters set. These indices are compressed and operated in their compressed form. Filters can be represented by the Boolean expression of multiple indices. Boolean operations on compressed indices can improve performance and save space. Consider the publisher column in Table I, for each unique publisher, we can get some information which row of the table the publisher is seen. We store the information in a binary array, which represents the row by the array indices. If the publisher is seen in a certain row, the array indices will be marked as 1, for example:

$$sina.com \rightarrow rows[0, 1] \rightarrow [1][1][0][0]$$
$$yahoo.com \rightarrow rows[2, 3] \rightarrow [0][0][1][1]$$

The sina.com appears at 0 and 1 column. The mapping of column values to the row indices forms an inverted index. In order to know which rows contain sina.com or yahoo.com, we join the two arrays with or. The result can be computed as follows:

$$[1][1][0][0] \vee [0][0][1][1] = [1][1][1][1]$$

C. Private Search Scheme

We have expanded the broker to implement the private search scheme, which works with four steps:

Step 1: Client Query Construction Procedure

In the step, a public dictionary of potential keywords $D = \{w_1, w_2, \ldots, w_{|D|}\}$ is assumed to be available. Constructing the encrypted query for some disjunction of keywords $K \subseteq D$ then proceeds as following. The client generates a key pair, then for each $i \in \{1, \ldots, |D|\}$, defines $q_i = 1$ if $w_i \in K$ and $q_i = 0$ if $w_i \notin K$. The values $q_1, q_2, \ldots, q_{|D|}$ are encrypted and put in the array $Q = (E(q_1), E(q_2), \ldots, E(q_{|D|})$, which forms the final encrypted query. The client then sends Q and the public key n to the broker node.

Step 2: Broker Stream Search Procedure

In addition to the public key and Q, the client may provide the broker with the parameter t, the number of segments to process before returning the results, and the parameters l_F, l_I and k, which affect correctness and performance. The broker must maintain three buffers as it processes the segments in its stream. These buffers are hereafter referred to as the *data buffer*, the *c-buffer*, and the *matching-indices buffer* denoted F, C, and I respectively. Each of these is an array of elements from the ciphertext space $Z \times n^2$, with F and C of length l_F and I of length l_I. For simplified notation here and in subsequent explanations, we assume that each segment is at most n bits and therefore fits within a single plain text in $Z \times n^2$. For longer segments requiring s elements of Z_n, we would let F be $l_F \times s$ array and subsequent operations involving a segment updating F are performed blockwise.

The *data buffer* will store the matching segments in an encrypted form which can then be used by the client to reconstruct the matching segments. In particular, the data buffer will contain a system of linear equations in terms of the content of the matching segments in an encrypted form. This system of equations will later be solved by the client to obtain the matching segments.

The *c-buffer* stores in an encrypted form the number of keywords matched by each matching segment. We call the number of keywords matched for a segment the *c-value* of the segment. The *c-buffer* will be used in reconstruction of the matching segments from the *data buffer* by the client. As in the case of the *data buffer*, the *c-buffer* stores its information in the form of a system of linear equations. The client will later solve the system of linear equations to reconstruct the *c-values*.

The *matching-indices buffer* is an encrypted Bloom filter that keeps track of the indices of matching segments in an encrypted form. More precisely, the *matching-indices buffer* will be an encrypted representation of some set of indices $\{\alpha_1, \alpha_2, \ldots, \alpha_r\}$, where $\{\alpha_1, \alpha_2, \ldots, \alpha_r\} \subseteq 1, \ldots, t$. Here r is the number of segments which end up matching the query.

Each of these buffers begins with all its elements initialized to encryptions of zero. We now detail how they are updated as each segment is processed. To process the ith segment f_i, the broker takes the following steps.

Step 2.1: compute encrypted *c-value*. First, the broker looks up the query array entry $Q[j]$ corresponding to each word w_j found in the segment. The product of these entries is then computed. Due to the homomorphic property of the Paillier cryptosystem [11], this product is an encryption of *c-value* of the segment, i.e., the number of distinct members of K found in the segment. That is:

$$\prod_{w_j \in W_i} Q[j] = E(\sum_{w_j \in W_i} q_j) = E(c_i)$$

Where W_i is the set of distinct words in the ith segment and c_i is defined to be $|K \bigcap W_i|$. Note in particular that $c \neq 0$ if and only if the segment matches the query.

Step 2.2: update *data buffer*. The broker computes $E(c_i f_i)$ using the homomorphism property of the Paillier cryptosystem.

$$E(c_i)^{f_i} = E(c_i f_i) = \begin{cases} E(c_i f_i) & \text{if } f_i \text{ matches the query} \\ E(0) & \text{otherwise} \end{cases}$$

The broker multiplies the value $E(c_i f_i)$ into a subset of the locations in the *data buffer* according to the following procedure. Let G be a family of pseudo-random functions that map $Z \times Z$ to $\{0, 1\}$. Randomly select $g \xleftarrow{R} G$. The algorithm multiplies $E(c_i f_i)$ into each location j in the *data buffer* where $g(i, j) = 1$. Suppose for example we are updating the third location in the *data buffer* with the second segment. Assume that the first segment was also multiplied into this location, i.e., $g(1, 3) = g(2, 3) = 1$.

Each of the two segments may or may not match the query. Suppose in this example that f_1 matches the query, but f_2 does not. Before processing f_2 we have that $D(F[3]) = c_1 f_1$. After multiplying in $E(c_2 f_2)$, $D(F[3]) = c_1 f_1 + c_2 f_2$. But $c_2 = 0$ since f_2 does not match, so it is still the case that $D(F[3]) = c_1 f_1$ and the *data buffer* is effectively unmodified. This mechanism allows the *data buffer* to accumulate linear combinations of matching segments while discarding all non-matching segments.

Step 2.3: update *c-buffer*. The value $E(c_i)$ is multiplied into each of the locations in the *c-buffer* in a similar fashion as $E(c_i f_i)$ was used to update the *data buffer*. In particular, the broker multiplies the value $E(c_i)$ into each location j in the *c-buffer* where $g(i, j) = 1$.

Step 2.4: update *matching-indices buffer*. The broker then multiplies $E(c_i)$ further into a fixed number of locations in *matching-indices buffer*. This is done using essentially the standard procedure for updating a Bloom filter. Specifically, we use k hash functions $h_1, \ldots, h_k$ to select the k locations where $E(c_i)$ will be added. For optimal efficiency, the client should select the parameter k as $k = \lfloor \frac{l_I log_2}{m} \rfloor$, where m is the number of segments they expect to retrieve. The locations of the *matching indices buffer* that a matching segment i is multiplied into are take to be $h_1(i), h_2(i), \ldots, h_k(i)$. Again, if the f_i does not match and $c_i = 0$, so the *matching-indices buffer* is effectively unmodified.

After completing the aforementioned steps for a fixed number of segments t in its stream, the broker sends its three buffers back to the client. Also, the broker should return the function g.

Step 3: Client Segment Reconstruction Procedure

The construction process works with following three-step: decrypt buffers, reconstruct matching indices and reconstruct *c-values* of matching segments. Next, we elaborate on each step:

Step 3.1: decrypt buffers. The client first decrypts the values in the three buffers using the Paillier decryption algorithm with its private key K_{priv}, obtaining decrypted buffers F, C and I.

Step 3.2: reconstruct matching indices. For each of the indices $i \in \{1, 2, \ldots, t\}$, the client computes $h_1(i), h_2(i), \ldots, h_k(i)$ and checks the corresponding locations in the decrypted *matching-indices buffer*; if all these locations are non-zero, then i is added to the list $\{\alpha_1, \alpha_2, \ldots, \alpha_\beta\}$ of potential matching indices. Note that if $c_i \neq 0$, then i will be added to this list. However, due to the false positive feature of Bloom filters, we may obtain some additional indices. Now we may check for overflow, which occurs when the number of false positives plus the number of actual matches r exceeds l_F. At this point if $\beta < l_F$, we continue to add indices to the list until its length equals l_F. Here the function pick denotes the operation of selecting an arbitrary member of a set. Note that we will not run out of indices since $t > l_F$.

Step 3.3: reconstruct *c-values* of matching segments. Given our superset of the matching indices $\{\alpha_1, \alpha_2, \ldots, \alpha_{l_F}\}$, the client next solves for the values of $\{c_{\alpha_1}, c_{\alpha_2}, \ldots, c_{\alpha_{l_F}}\}$. This is accomplished by solving the following system of linear equations $A \cdot \overrightarrow{c} = C'$, where A is the matrix with the i, jth entry set to $g(\alpha_i, j)$, C' is the vector of values stored in the decrypted *c-buffer*, and $\overrightarrow{c}$ is the column vector denoted as $\overrightarrow{c} = (c_{\alpha_i})_{i=1,\ldots,l_F}$. Now the exact set of matching indices $\{\alpha_1', \alpha_2', \ldots, \alpha_r'\}$ may be computed by checking whether $c_{\alpha_i} = 0$ for each $i \in \{1, \ldots, l_F\}$. Before proceeding, we replace all zeros in the vector $\overrightarrow{c}$ with ones.

As an example of Step 3, suppose that there are four spots in the decrypted *c-buffer*, seven segments are processed, and we have established the following list of potentially matching indices: $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4\} = \{1, 3, 5, 7\}$. Then given

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}, C = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 3 \end{pmatrix}$$

We may compute:

$$\begin{aligned} c_{\alpha_1} &= c_1 = 1 \\ c_{\alpha_2} &= c_3 = 2 \\ c_{\alpha_3} &= c_5 = 1 \\ c_{\alpha_4} &= c_7 = 0 \end{aligned}$$

We then see that there were three matching segments ($r = 3$): f_1, f_3 and f_5.

Step 4: Reconstruct Matching Segments

In the step, the content of the matching segments $f_{\alpha_1'}, f_{\alpha_2'}, \ldots, f_{\alpha_r'}$ may be determined by solving the linear system $A \cdot diag(\overrightarrow{c}) \cdot \overrightarrow{f} = F'$, where

$$diag(\overrightarrow{c}) = \begin{pmatrix} c_1 & 0 & \cdots \\ 0 & c_2 & \\ \vdots & & \ddots \end{pmatrix}$$

We directly compute $\overrightarrow{f} = diag(\overrightarrow{c})^{-1} \cdot A^{-1} \cdot F'$. Note that $diag(\overrightarrow{c})$ is never singular, because we previously ensured that no zeros appear in $\overrightarrow{c}$. The content of the matching segments appears as $f_{\alpha_1'}, f_{\alpha_2'}, \ldots, f_{\alpha_r'}$ the other entries in $\overrightarrow{f}$ will be zero. Continuing the example above (and making up a value of F), this corresponds to solving the following equations:

$$\begin{aligned} f_1 + f_5 &= 32 \\ f_1 + 2f_3 + f_7 &= 32 \\ f_1 + f_7 &= 10 \\ 2f_3 + f_5 &= 44 \end{aligned}$$

Thereby determining that $f_1 = 10$, $f_3 = 11$, $f_5 = 22$ and $f_7 = 0$, but this value will be ignored.

IV. Experiments

To test the performance of our system, we created a large test cluster with 80GB data including millions of rows. This data set includes more than a dozen dimensions, and the cardinalities ranges from double digits to tens of millions. We calculate three aggregation metrics for each row (count, sum, average). This data set is firstly divided on the time stamp, and then on dimension value to create thousands of segments, each segment is about 10,000 lines. Testing benchmark cluster contains 6 compute nodes, and each node has 16 cores, 16GB of RAM, 10GigEFA Ethernet and 1TB of disk space. Overall, the cluster contains 96 cores, 96GB of RAM, as well as enough fast Ethernet and enough disk space. The query statements in Table II describe the purpose of each query.

1. The scope of timestamp of queries covers all data;

2. Each machine has 16GB of RAM and 1TB of disk and 16 cores. The machine is configured to use 15 threads to process queries and to memory map the data instead of loading it into the Java heap.

Figure 5 shows the cluster scanning rate, and Figure 6 shows the core scanning rate. In Figure 5, we find the results of the expected linear scaling based on the result of the 5 nodes cluster. In particular, we inspect the performance of the marginal revenue decreases with the scale of the cluster increasing. Under the expected linear scaling; Query 1 on a cluster with 55 nodes would achieve scanning rate of 37 million rows per second. In fact, the scanning rate is 25 million rows per second. However, the Query 2-6 keep a linear speedup until up to 30 nodes, while in Figure 6 the core scanning rate of the query remains almost stable.

According to the Amdahls Law, the increase of the speed of a parallel computing system is often limited by the time requirements for the sequential operations of the system. In Table 2, the first query is a simple counting, achieving scan rate of 330 thousands lines per second per core. In fact, we consider that the cluster with 55 nodes is actually over provisioned for the test datasets, which explains the growth is slower than the cluster with 30 nodes. Concurrency model of our system is based on the segment: one thread scan a segment. If the number of segments on a node modulo the number of cores is small (such as 17 segments and 15 cores), during the last round of calculation, some of the core will be idle. When more aggregation metrics are added, we find performance degrade. This is because our system uses a column-oriented storage format. For the count(*) query, the system has to check

TABLE II. THE QUERY STATEMENT

Query No.	Query
1	SELECT count(*) FROM _table_ WHERE timestamp $\geq$? AND timestamp $<$?
2	SELECT count(*), sum(metric1) FROM _table_ WHERE timestamp $\geq$? AND timestamp $<$?
3	SELECT count(*), sum(metric1), sum(metric2), sum(metric3), sum(metric4) FROM _table_ WHERE timestamp $\geq$? AND timestamp $<$?
4	SELECT high_card_dimension, count(*) AS cnt FROM _table_ WHERE timestamp $\geq$? AND timestamp $<$? GROUP BY high_card_dimension ORDER BY cnt limit 100
5	SELECT high_card_dimension, count(*) AS cnt, sum(metric1) FROM _table_ WHERE timestamp $\geq$? AND timestamp $<$? GROUP BY high_card_dimension ORDER BY cnt limit 100
6	SELECT high_card_dimension, count(*) AS cnt, sum(metric1), sum(metric2), sum(metric3), sum(metric4) FROM _table_ WHERE timestamp $\geq$? AND timestamp $<$? GROUP BY high_card_dimension ORDER BY cnt limit 100

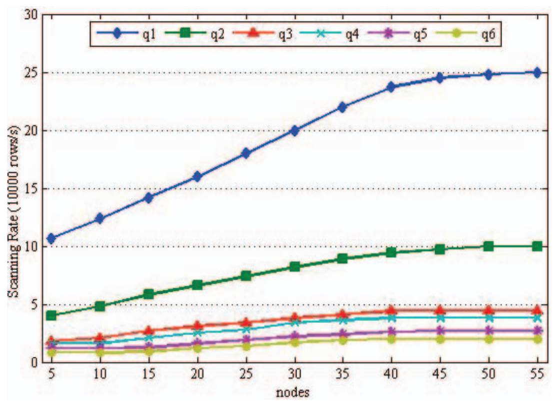

Fig. 5. The cluster scanning rate

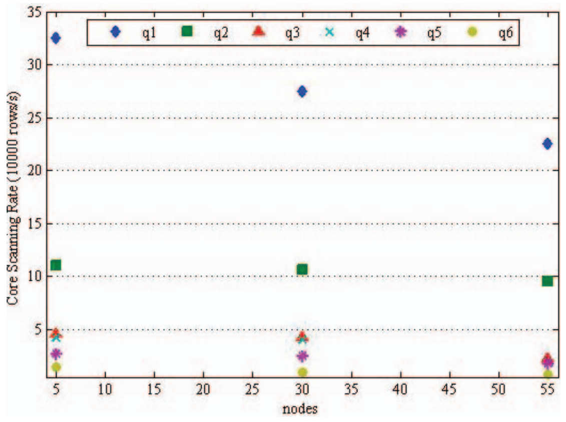

Fig. 6. The core scanning rate

timestamp column to determine whether it satisfies the where clause. When we add metrics, the system has to load those metric values and scan over them, which takes up the memory being scanned.

The last experiment compares the time consumption of average aggregate function running in encryption search system [19] to that running in our system with the scale of the input increasing, as shown in Figure 7. Since the encryption search system [19] cannot support dynamic scalability, its time consumption keeps high increasing rate, on the contrary, our system is dynamically scalable according to the input scale, so the time consumption keeps stable over time.

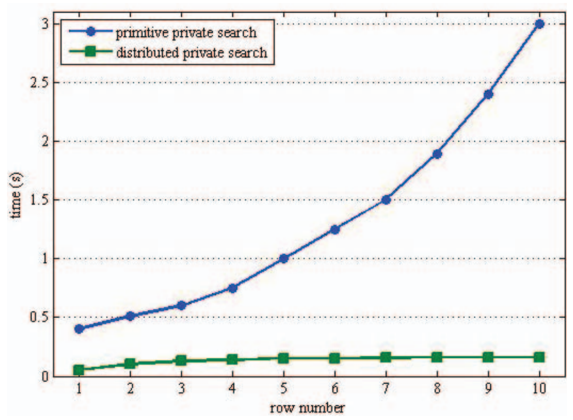

Fig. 7. The time consumption of average aggregate function

V. CONCLUSIONS

In this paper, we propose a scalable distributed private stream search system, which adopts the shared-nothing architecture to support the scalability. The experiments show the system has good performance and scalability on online aggregation queries. Moreover, the query can be encrypted through Paillier encryption to protect search criteria.

ACKNOWLEDGMENT

The research work is supported by Supported by Strategic Priority Research Program of the Chinese Academy of Sciences under Grant (No.XDA06030602), China Postdoctoral Science Foundation under Grant (No. 2013M541076) and National Natural Science Foundation under Grant (No.61402464, 61370025).

REFERENCES

[1] X. Meng and X. Ci, “Big data management: Concepts, techniques and challenges,” *Journal of computer research and development*, vol. 50, no. 1, pp. 146–169, 2013.

[2] G. Jim and G. Goetz, “The five-minute rule ten years later, and other computer storage rules of thumb,” *ACM Sigmod Record*, vol. 26, no. 4, pp. 63–68, 1997.

[3] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan, “Muppet: Mapreduce-style processing of fast data,” *PVLDB*, vol. 5, no. 12, pp. 1814–1825, 2012.

[4] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, “Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters,” in *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing*, 2012.

[5] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. M. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, “The case for ramclouds:

scalable high-performance storage entirely in dram," *Operating Systems Review*, vol. 43, no. 4, pp. 92–105, 2009.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, 2010.

[7] J. Bethencourt, D. X. Song, and B. Waters, "New techniques for private stream searching," *ACM Transactions on Information and System Security*, vol. 12, no. 3, 2009.

[8] S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Outsourced symmetric private information retrieval," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, 2013, pp. 875–888.

[9] S. Yekhanin, "Private information retrieval," *Communications of the ACM*, vol. 53, no. 4, pp. 68–73, 2010.

[10] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, p. 18, 2014.

[11] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques*, 1999, pp. 223–238.

[12] R. Ostrovsky and W. E. S. III, "Private searching on streaming data," in *Proceedings of the 25th Annual International Cryptology Conference*, 2005, pp. 223–240.

[13] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-dnf formulas on ciphertexts," in *Proceedings of the Second Theory of Cryptography Conference*, 2005, pp. 325–341.

[14] X. Yi and C. Xing, "Private (t, n) threshold searching on streaming data," in *Proceedings of International Confernece on Social Computing Privacy, Security, Risk and Trust*, 2012, pp. 676–683.

[15] M. Bhandarkar, "Hadoop: a view from the trenches," in *Proceedings of The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013, p. 1138.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the USENIX Annual Technical Conference*, 2010.

[17] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB)*, 2011.

[18] A. Colantonio and R. D. Pietro, "Concise: Compressed ncomposable integer set," *Information Processing Letters*, vol. 110, no. 16, pp. 644–650, 2010.

[19] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Financial Cryptography and Data Security*, 2010, pp. 136–149.