RC25456 (WAT1404-022) April 8, 2014
Computer Science

IBM Research Report

FCCE: Highly Scalable Distributed Feature Collection and Correlation Engine for Low Latency Big Data Analytics

Douglas Schales, Xin Hu, Jiyong Jang, Reiner Sailer, Marc Stoecklin*, Ting Wang

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

*IBM Research Divison
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

FCCE: Highly Scalable Distributed Feature Collection and Correlation Engine for Low Latency Big Data Analytics

Douglas Schales, Xin Hu, Jiyong Jang, Reiner Sailer, Marc Stoecklin, Ting Wang
{schales, huxin, jjang, sailer, tingwang}@us.ibm.com, mtc@zurich.ibm.com
IBM T. J. Watson Research Center, IBM Zurich Research Lab

ABSTRACT

In this paper, we present the design, architecture, and implementation of a novel analysis engine, called FCCE, that finds correlations across a diverse set of data types spanning over large time windows with very small latency and with minimal access to raw data. FCCE scales well to collecting, extracting, and querying features from geographically distributed large data sets. FCCE has been deployed in a large production network with over 150,000 workstations for over 2 years, ingesting more than 2 billion events per day and providing low latency query responses for various analytics. We explore two real use cases and applications to demonstrate how we utilize the deployment of FCCE on large diverse data sets in the cyber security domain: 1) detecting fluxing domain names of potential botnet activity and identifying all the devices in the production network querying these names, and 2) detecting advanced persistent threat infection. Both evaluation results and our experience with real-world applications show that FCCE yields superior performance over existing approaches, and excels in the challenging cyber security domain by correlating multiple features and deriving security intelligence.

1. INTRODUCTION

Over the last few years, the database community has witnessed an important trend in data management. Many new data-driven applications demand for more flexible database schemes that process, compute, and combine information on-the-fly, as new data is flowing in and needs to be prepared for future queries. For several decades, relational database management systems (DBMSs) were the de-facto standard for such applications. With the recent shift of the industry towards data-driven analytics, where the paradigm "write-a-lot, read-a-little" is central in many applications, a new type of database management systems has emerged, so-called NoSQL databases. Many Internet applications are benefiting from these new type of databases, including online bidding, social network applications, or multi-player online gaming, where data requires to be exchanged in real time across participating parties. Beyond those applications, many other domains started to map data-intensive real-time exchange or data caching tasks onto the NoSQL paradigm.

One domain that has particularly benefited from these novel type of databases is *cyber security analytics*. Over the last decade, cyber security has become a big data problem, for which a wide variety of real-time and offline data sources have to be combined, stored, analyzed, and interpreted in order to identify complex on-going and past attacks and threats. Distributed key-value stores offer excellent performance and flexibility to cope with the high volume of events, accommodate various sensor formats, and retrieve events from different data sources and time ranges in a scalable fashion. However, current implementations of key-value stores only offer a limited capability to correlate data at query time across the data stores. For example, annotating the attributes queried from one data source using data from other sources requires resource-intensive sequential querying of the second source to retrieve the annotations. Moreover, repeating the annotation process over additional data sources renders the query logic complex and less usable.

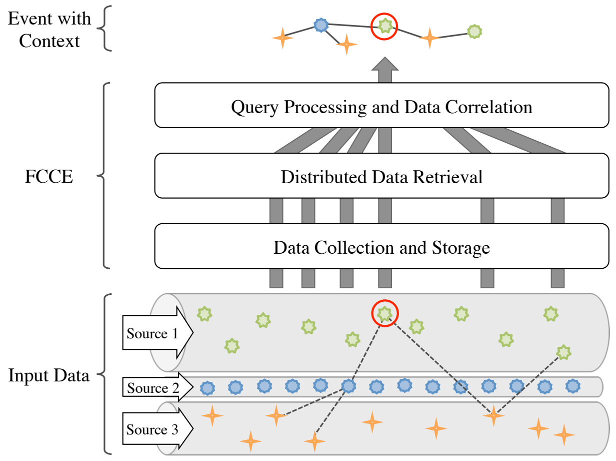

Event with Context
FCCE
Query Processing and Data Correlation
Distributed Data Retrieval
Data Collection and Storage
Input Data
Source 1
Source 2
Source 3

Figure 1: Processing, storing, and correlating real-time and historical data from multiple input sources by means of FCCE

In this paper, we address the challenge of collecting high volume data from arbitrary data sources, and storing it efficiently for high-performance and low-latency data retrieval, streaming, and correlation using a novel distributed key-value store (see Figure 1). A first problem we address is the segregation of raw data into pieces and the derivation of meaningful meta data representations, which we call *features*, that are expressive and relevant to support correlation at retrieval time. In addition to the feature representation, we strive for an approach that enables efficient access to the entire raw data stored. The second problem we focus on is the data correlation mechanism at query time; this mechanism enables a user to define and configure correlation criteria ahead of query time while leaving the complexity of resolving and annotating attributes to the underlying database management system.

The main contributions of this paper are:

- We present the design and architecture of the *Feature Collection and Collection Engine* (FCCE), a highly-scalable, low-latency, distributed key-value data management system. FCCE is optimized to extract, normalize, store, retrieve, and correlate features from diverse data sources. FCCE is designed to support geographically distributed data sources, and does not require continuous connectivity between the data sources. FCCE also offers resilience against failures of individual nodes within the distributed engine architecture.
- We propose a novel approach to integrate correlation criteria directly into the query engine of our key-value store to expedite the query response time and reduce computational and I/O overhead.
- We implement and deploy FCCE in a large production network with 150,000+ user workstations and network servers, ingesting more than 2.3 billion events per day, and yielding a total raw compressed data storage of 43 TB over 2 years and a total feature storage size of 500GB/month (uncompressed). Our measurements show that our engine can ingest, extract, and normalize features from more than 3 billion events per day (e.g., Netflow, DNS messages, DHCP logs, firewall logs) with the performance being I/O bound and scaling largely linearly with the number of hardware nodes deployed.
- We evaluate FCCE with a large-scale real-world network data to show low-latency query response time, scalability, and real-time processing capability. We also demonstrate practical impacts by applying FCCE to two cyber security analytics use cases: (i) botnet behavior detection and (ii) post-incident cyber threat investigation.

The remainder of the paper is organized as follows. We describe the applications and challenges of NoSQL data stores in the cyber security domain in §2. We detail the data model in FCCE and interfaces for storing, retrieving, and correlating features in §3. We present the architecture of FCCE in §4, and its implementation in §5. The performance of FCCE is evaluated in §6 along with several real-world use cases and applications. We discuss relevant literature and existing work in §7, and conclude in §8.

2. NOSQL IN CYBER SECURITY

Corporations and governmental organizations have been challenged by a drastic shift towards highly sophisticated and targeted cyber security threats. Attackers are increasingly applying stealthy attack techniques to hide their presence or, at least, reduce the probability of being detected, e.g., by concealing their attack steps over multiple machines, exploiting different application protocols, or spreading their activities over long time frames. Many of these threats are referred to as advanced persistent threats (APT). Yet the chance to be able to detect and investigate such complex attack patterns requires collecting, storing, and analyzing events from a variety of vantage points, different data sources, and multiple abstraction layers, such as raw network packets and network flow exports, DNS messages, DHCP requests, ARP packets, wireless authentication logs, IDS/IPS alerts, external blacklists, and so on.

The monitoring data, exported typically at rates of many thousands of events per second, needs to be collected, stored, and made available for real-time and historical analysis. With such a load, a wide variety of relevant data types, and varying collection delays (e.g., network flow summaries are exported near real-time versus authentication logs are pulled periodically), cyber security threat investigation has turned into a big data problem and demands for careful selection of suitable data management technologies. Many events collected become only meaningful when they are put into context and correlated with different data sources over potentially large time windows (e.g., weeks or months) in order to assess the big picture of on-going and past activities and to filter out false alarms having little or no impact.

The data flow in cyber security applications is of the type "write-a-lot, read-a-little", as data is collected at a high speed (e.g., 100K events/sec or more). Moreover, the data exported is static and does not change (e.g., by means of updates) at a later point in time. For example, log files contain unchanged records of what happened. Existing (relational) databases and their underlying indices are optimized for scenarios where data may be changed and indices are reorganized and rebuilt over time; however, this is achieved at the expense of more overhead during data insertion or update. In addition, cyber security applications need to deal with heterogeneous data sources in different formats and collecting intervals. For example, some information is streamed in near real-time whereas other data is exported periodically. Finally, a lot of data relevant for cyber security provide contextual insights, such as associations between IP addresses and hardware MAC addresses, or ownership between devices and users. Such data is similar to (time-dependent) mapping between keys and values. Compared with a traditional DBMS, NoSQL database management systems, and in particular key-value stores, provide superior performance that is crucial for cyber security analytics.

In the remainder of this section, we provide an example of a typical query we perform in many cyber security analytics applications. Figure 2 illustrates how we analyze the scope of the impact of a known malicious or suspicious external machine (given its Fully Qualified Domain Name) in 5 stages, whereby the output of one correlation stage is fed as an input into the next stage(s).

a) We look up all the IP addresses related to the investigated external domain name.

b) We find all other names resolving to any of those IP addresses, both historically and in real-time. This expands our knowledge from a single system that has been reported to host malicious activities (e.g., by blacklists or external investigation reports) or detected locally as the source of malicious activity (e.g., by botnet analytics) to the larger network infrastructure related to this system.

c) We look up all the IP addresses that have been returned for any of those names during the investigation time period (e.g., one month). At this point we have expanded our knowledge about the external infrastructure that may be related to the incident and we have

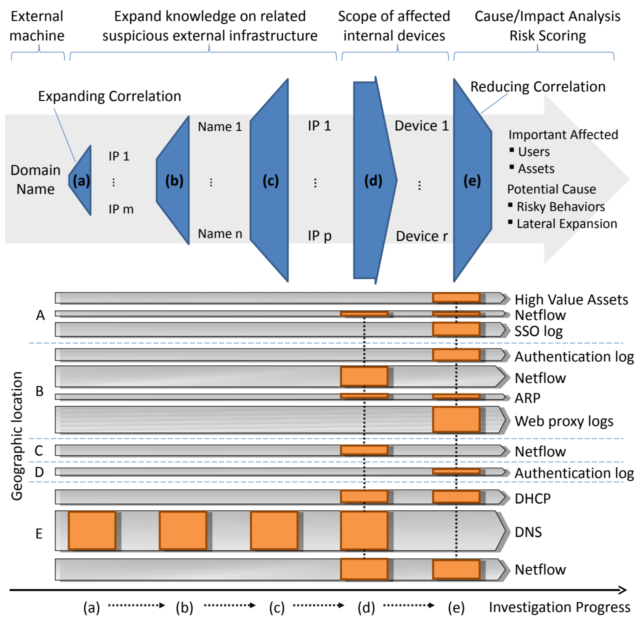


Figure 2: Security threat impact analysis of a known malicious/suspicious external host by automatically extracting correlated events from different locations, data sources, and time periods.

looked three times at the DNS data collected during this time period while changing inputs. The operations are hard to parallelize.

d) We transfer analysis from the 'outside' to the 'inside' of the monitored network and determine all internal devices that either looked up any of the external names (DNS messages) and/or connected (network flows) to any of the external IP addresses. Flows and lookups deliver internal IP addresses, that are automatically translated into MAC addresses (using historical DHCP /ARP information) and then collapsed to the different devices (e.g., unifying wireless and wired MAC addresses to a single machine).

e) We can lookup which credentials have been used on those devices (single-sign on, authentication logs) and that may have been exposed on those suspicious devices, or which high value assets have been accessed from those devices (network flows, high value asset information). We can reduce the number of further investigated devices by prioritizing them based on the privileges of the user credentials or the accessed servers hosting valuable assets.

As we will see in more details later, FCCE can perform such tasks with only several seconds latency.

3. CONCEPT

In this section we present a set of key concepts of FCCE, including the data model, idempotence, and APIs, and discuss the rationale behind the design choice to make FCCE simple yet suitable for the target applications.

3.1 Data Model

As introduced in §1, the design of FCCE centers around the concept of *features*. A feature essentially specifies the relationship between a pair of *key* and *value*, both of which may comprise multiple attributes. Each type of features has a unique name, typically given by concatenating the names of the key and value attributes.

EXAMPLE 1. IPByNameDate *names the type of features specifying the DNS resolution relationship of* Domain Name + Date *(key) and* IP Address *(value), e.g.,* www.example.com + 20130101 → 1.2.3.4.

FCCE presents a simplified relational data model to a user. Conceptually, data is organized by tables and rows. Each table stores one type of features. Each row is individually identified by a key, and may contain multiple values. Please note that our concrete implementation does not strictly follow this conceptual model for storage and retrieval efficiency. We discuss implementation details in §5.

3.2 Idempotence

Conventional key-value stores typically differentiate the values associated with the same key by assigning different version numbers. In contrast, FCCE treats the values associated with the same key as a set.

The rationale behind the simplification is as follows. First, the loss of ordering and duplicate information is often nonconsequential for applications that continuously correlate live and historical data. Second, one can always impose the ordering information by appending timestamps to the values. Third, most importantly, the order in which values are stored in FCCE is transparent to read operations.

This leads to the *idempotence* property of FCCE, and this property significantly simplifies the logic of collecting features in environments wherein diverse data arrives along multiple overlapping paths originated from geographically distributed and heterogeneous data sources (details in §5).

3.3 API

At the low level, FCCE provides a set of flexible APIs for storing, retrieving, and correlating features.

put(feature, ⟨K, V⟩): put stores key-value pair (K, V) to feature table. FCCE handles a series of key-value pair inputs (e.g., inputs from a file or a feature collector).

get(feature, ⟨K, ...⟩): get reads the values associated with K from feature table. Note that ⟨...⟩ indicates that FCCE can take a sequence of keys as an input (e.g., read a list of keys from a file, data stream, or standard input) and fetch all of their associated values.

Although its basic form seems similar to the get function in conventional key-value stores, FCCE differs in providing two powerful mechanisms that empower simple APIs to support efficient correlation operations between features.

a) **Query pipeline:** This mechanism allows passing the features retrieved from one query to another query as inputs. In other words, with query pipeline, one is able to chain multiple get functions sequentially, thereby generating the intersection of multiple feature stores.

EXAMPLE 2. *Table* NameDateByClient *stores the domain names queried on specific dates by each client while table* IPByNameDate *stores the resolved IP addresses of the domain names on a specific date. Therefore, one may have the following expression which chains two get functions:*

get(IPByNameDate, get(NameDateByClient, 1.2.3.4))

By correlating both features, one can identify all the IP addresses to which client 1.2.3.4 *tries to connect.*

b) **Query modifier:** On top of the basic get operation, FCCE offers a rich set of options for fine-grained controls of its behavior. We refer the options as query modifiers. A subset of important modifiers include:

- ++select ⟨field⟩: This allows selecting a subset of specific fields from each record.
- ++subscribe: This enables the subscription mode, which consumes data directly from live data streams (details in §4).
- ++where ⟨expression⟩: This filters the query results based on the evaluation of the expression.
- ++duplicate: This allows duplicate records to be output for saving the cost of de-duplication (details in §5).

EXAMPLE 3. *The following expression identifies the IP addresses in the subnet of* 1.2.0.0/16*, which have been visited by the client* 1.2.3.4.

get(IPByNameDate, get(NameDateByClient, 1.2.3.4), '++where', 'in 1.2.0.0/16')

4. ARCHITECTURE

In designing FCCE, we strive to achieve the following five objectives:

Scalability: It should scale up to ingest extremely high-volume data input (e.g., in the order of hundreds of thousands of events per second).

Low query latency: It should respond to queries in fairly limited time windows (e.g., in the order of nanoseconds).

Versatility: It should be easily adaptable to handle new data types (e.g., various network monitoring data).

Fault tolerance: It should be resilient against the failures of single points in the system.

Usability: It should provide a rich and interactive interface for accessing the features in the store.

In the rest of the section, we first give an overview of the system architecture of FCCE and then detail the design decisions of its core components that help achieve the aforementioned objectives.

As introduced in §3, in FCCE, we extract meta data representations called features. An extracted feature is conceptually a key-value pair, where a key essentially points to a specific "bucket" of feature values. For example, when we collect features from DHCP requests, an IP address as a key can be mapped to a set of hardware MAC addresses that were

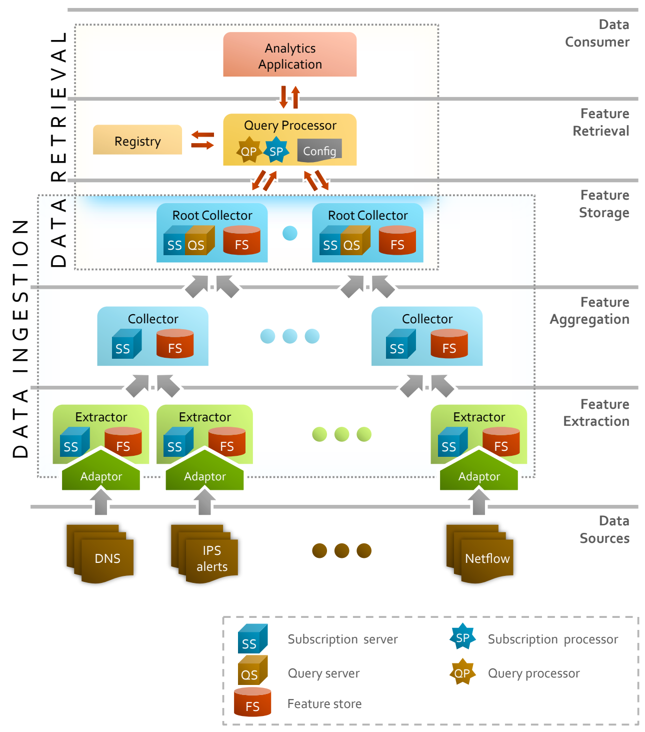


Figure 3: Architecture of FCCE: Organization of components and flow of data in FCCE during data ingestion (bottom box) and retrieval (top box).

associated with the IP address. We treat the feature values in each bucket as a mathematical *set*. The collection of such sets forms *feature store* (FS), which is implementable using off-the-shelf key-value data stores. The use of mathematical sets to aggregate features allows to ingest data without considering its temporal ordering, which is especially beneficial for distributed environments. Furthermore, the use of sets enables to efficiently aggregate different data sources that may become available at different time frames.

FCCE entails a complete framework for ingesting, aggregating, storing, as well as retrieving data. The overall architecture of FCCE is sketched in Figure 3. The data ingestion part can yet be subdivided into *feature extraction* (ingesting and processing raw data to abstract features), *feature aggregation* (collecting and merging features from different data inputs), and *feature storage* (storing the aggregated results). For data retrieval, a *feature retrieval* layer provides an interface for efficiently querying features of interest by a data consumer (e.g., analytics applications).

Feature extraction. For each input data source, domain experts specify the method of extracting (or "decapsulating") and abstracting features from raw data, which is implemented as a component called *Extractor*. Each data source is associated with one or more extractors for fast processing. The extracted features from each data source are directly forwarded to the next phase, or de-duplicated and cached in local and transient feature stores. The transient feature stores constitute the locally derived knowledge from the associated input data sources.

Feature aggregation. The next phase is to aggregate the local knowledge at different extractors to form a global view. It is implemented by a component which we refer to as *Collector*. Each collector takes as an input the features extracted by multiple extractors, and aggregates the features while de-duplicating any redundancy. At each collector, a local feature store keeps the derived and de-duplicated knowledge from all the inputs fed into the collector. In the same manner as an extractor, a collector can optionally forward new values to one or more other collectors, constructing a hierarchical structure (e.g., a tree or a DAG) for the purpose of scalability, load balancing, and redundancy.

Feature storage. At the top of the hierarchy of collectors, one or more collectors are designated as *Root Collectors*. Root collectors provide the permanent storage for the collected features, as well as the *Query Service* (QS) for accessing the information.

Feature retrieval. FCCE provides a rich query interface by supporting 3 different ways of accessing the derived knowledge. First, one can use the *registration service* (RS) to find the root collector that stores the features of interest, and follow the *query protocol* (QP) to send queries to the corresponding feature store using specific feature types and keys as query predicates. Second, one can also subscribe to specific extractors/collectors (as routed by the registration service) about feature types of interest using the *subscription protocol* (SP). Once new key-value pair of the subscribed feature types are inserted to the (local) feature stores, they are also forwarded to the subscribers. Finally, one can use the interface of *feature correlation* (FC) and set up customized correlation functions to acquire knowledge from different types of features.

5. IMPLEMENTATION

In this section, we address the detailed issues of implementing FCCE. Figure 4 illustrates the core components of FCCE (referred to as its *core modules*), which are implemented using about 28,000 lines of C code. Based on their functionalities, the core modules are grouped into four layers: (a) the ingestion and extraction layer that receives incoming data and normalizes them into features, (b) the feature collection layer that aggregates and de-duplicates the features, (c) the feature store management layer that archives and manages the aggregated features, and (d) the feature access service layer that provides query services for retrieving or correlating features. Below we detail their implementation.

5.1 Ingestion and Extraction

FCCE supports a range of methods to ingest live data into the system. Presently, raw TCP and UDP socket readers, a file reader, and API wrappers (available in C, Python, and Perl) are implemented.

A *feature extractor* decapsulates features from the associated input data, and encodes the features into a pre-defined format. More precisely, the ingested data is first decoded by a data type-specific component (e.g., tcpdump to decode pcap data), which can be either loaded from a collection of modules or a custom-built module. The desired information is extracted from the data and mapped into key-value pairs based on the configuration. In addition, a system timestamp (T) is automatically attached to each key-value pair to indicate its insertion time; moreover, the identifier of the feature of interest (featureID) is also attached to the key, which

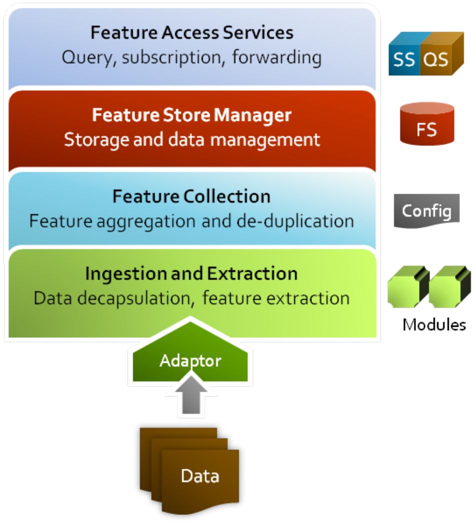

Feature Access Services
Query, subscription, forwarding
SS
QS
Feature Store Manager
Storage and data management
FS
Feature Collection
Feature aggregation and de-duplication
Config
Ingestion and Extraction
Data decapsulation, feature extraction
Modules
Adaptor
Data

Figure 4: Illustration of the FCCE core modules, handling feature extraction, collection, storage, and access services.

will direct FCCE to appropriately store the feature. Each key-value pair is then encoded into a format defined by the configuration, with the binary key K and value V.

Example 4. *We show how a "Domain Name-to-IP Address" feature,* IPByNameDate*, is created from DNS responses. We configure the feature extractor with the appropriate data adaptor (e.g., Wireshark) to handle all incoming DNS responses and map them into the featureID of* IPByNameDate*. Subsequently, for every DNS response, a key-value pair is extracted by the adaptor of the format* ⟨(Domain Name, Date), IP Address⟩*. For instance, if a DNS lookup returns the following resource record on* 2013/04/01*:*

www.example.com. IN A 192.168.1.1

*The adaptor generates an entry with key (*www.example.com., 20130401*) and value* 192.168.1.1*.*

5.2 Feature Collection

At the feature aggregation layer, a set of *collectors* aggregates the features fed by multiple extractors (or peer collectors) and de-duplicates redundant information in the input. Specifically, for a new key-value entry K-V (associated with timestamp T), a collector implements the de-duplication operation as sketched in Algorithm 1. Note that every tuple $\langle \mathsf{K}, (\mathsf{T}, \mathsf{V}) \rangle$ written to the local feature store is also forwarded to the designated upper-level collector(s) in the hierarchy, which are configurable in the registry service (details in §5.4).

5.3 Feature Storage Manager

At the core of our implementation of FCCE is the *feature store manager* (FSM), which manages an underlying custom-built key-value store. Although the use of off-the-shelf key-value stores is also possible, our implementation leverages several key properties of FCCE, including (i) idempotence, (ii) write-a-lot-read-a-little, and (iii) append-only update, thereby achieving superior storage and retrieval performance over conventional key-value stores (see §6.1).

Algorithm 1: De-duplicating and storing new feature values.

```
for newly arriving key-value pair $K$-$V$ with timestamp $T$ do
  $R \leftarrow$ lookup of local feature store using key $K$
  if $R = \emptyset$ then
    store tuple $\langle K, (T, V)\rangle$ {new record is created}
  else
    if $V$ exists in $R$ then
      $T' \leftarrow$ timestamp associated with $V$ in $R$
      if $T < T'$ then
        replace $T'$ with $T$ in $R$
        write $R$ to feature store {update timestamp}
      end if
    else
      append $(T, V)$ to $R$
      write $R$ to feature store {append new value}
    end if
  end if
end for
```

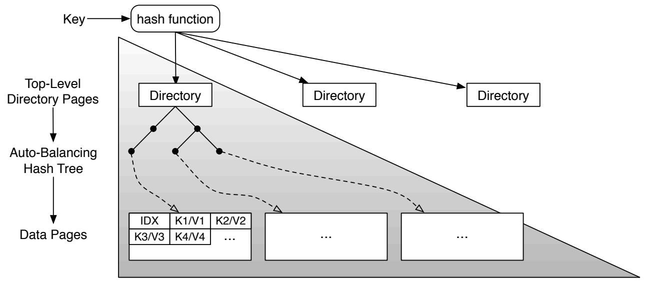


Figure 5: Implementation of FCCE feature store manager

Figure 5 illustrates the implementation of FSM. Overall the key-value store is organized as a set of fixed size pages. There are three types of pages: free map pages, data pages, and directory pages. Free map pages maintain a record for each page with the amount of free space on that page. The data pages contain one or more key-value data entries stored in the form of key length, value length, key bytes, and value bytes. When an entry could not fit in one page, it is expanded to another page. The directory pages store the directory records. Overall these records form a dynamically expanded set of hash tables, which map the hashes of the keys to respective data pages. Further, the 16-bit checksum values are also used to reduce false hits. Note that since no information about the location of data entries on the data page is maintained, the data can be moved around within each page without the need to perform directory updates. Finally, as shown in Figure 5, all the directory pages are organized as a set of auto-balancing hash trees, thereby evenly distributing the storage load over different pages.

5.4 Feature Access Services

FCCE provides multiple services for feature data access. We highlight three most important services, and describe their roles.

Registration Service: A central component of FCCE is the *registration service* (RS). The RS is used as a map to locate other services. Because of the potential geographic distribution of the system, where communication may be disrupted, a completely centralized service registration system was ruled out. Our implementation has an RS on every node in the system. The local RSes contain only information about services that are only available on the local node. This enables locally contained operations to run even when the node becomes temporarily isolated or disconnected.

Global information is forwarded to a set of global registration servers. The actual forwarding is offloaded to the local registration servers, which are responsible for ensuring that the information eventually reaches the global servers.

Registration information consists of a set of key/value pairs. The following example indicates the presence of a tap service (with identifier tap1) for the data type DNS in the zone rcx, whose service interface listens on 10.10.0.5:55000.

```
'class=tap,type=dns,zone=rcx,tapid=tap1,
address=10.10.0.5,port=55000'
```

A query to the registration server provides some subset of the key/values and all entries that match will be returned. Thus, a query for

```
'class=tap,type=dns,zone=rcx,tapid=tap1'
```

would match the above and return all the values. This functionality is also used to locate where features may reside.

Query Service: The query service provides the most direct way to access the features in the store, which allows data consumers to look up the feature store using feature types (featureID) and query key(s) as query predicates. In particular, the query service implements the procedure as sketched in Algorithm 2.

Algorithm 2: Looking up features of interest using key(s).

```
combine and encode query key(s) with feature type to form key set $\{K\}$
for each key $k \in \{K\}$ do
  $R \leftarrow$ lookup of feature store using $k$
  if $R \neq \emptyset$ then
    for every feature value $(T, V) \in R$ do
      emit tuple $\langle k, (T, V)\rangle$
    end for
  else
    report failure
  end if
end for
```

The query service, co-located at root collector nodes, maintains registrations that provide higher level keying information about what data is in their feature stores. For example, a query service might be registered as a feature store class (fs), offering the feature IPByNameDate for the date range between 2013/04/01 and 2013/04/02, where two different query services endpoints (hosted on nodes 10.10.0.6 and 10.10.0.7) offer features for the same date.

```
'class=fs,feature=IPByNameDate,date=20130401,
  address=10.10.0.6,port=12345'
'class=fs,feature=IPByNameDate,date=20130401,
  address=10.10.0.7,port=12345'
'class=fs,feature=IPByNameDate,date=20130402,
  address=10.10.0.7,port=12345'
```

A query interface can locate all the query services offering features with the name IPByNameDate by requesting 'class=fs,feature=IPByNameDate'

at the registration service. This query would return the two registered query service endpoints above. If only information from the date 2013/04/02 was desired, the query interface would request

'class=fs,feature=IPByNameDate,date=20130402'

The query interface then sends the query directly to the resulting set of query service endpoints, identified by their IP address and port number. While the registration service provides a very high performance service, in practice, the query service would cache lookups and not have to perform the registration lookup for every query. In the same way, stores from different tenants can be separated in a multi-tenant setup.

The subscription service provides access to a near real-time stream of events ingested into the system. By means of the get API, a client subscribes to specific collectors that offer a subscription service. As part of the subscription operation, a client may subscribe by a feature name (featureID) and a key. Moreover, during subscription, the client may select value fields and define specific filters as with a regular query (see §3.3). A subscription server routes the request to the appropriate collectors (after consulting the registry), where a forwarding channel to the query processor is established. If a new event meets the filter criteria, then the desired fields are selected and forwarded. A dedicated network connection between the client and the subscription service is kept alive for as long as the client subscribes to the events. New events are passed as messages from the service to the client by means of an underlying subscription protocol.

5.5 Feature Correlation

Correlation is currently provided at the effective edge of the system. The results of two queries can be joined based on data fields within the records retrieved. In order to allow subscription based joins, periodic markers can be emitted to allow either aggregation, or expiration of older data. Future work will expand the correlation capabilities, moving it from the edge closer to the data.

6. EVALUATION

Over the last few years, we have gained significant experience when designing and optimizing FCCE, and deploying it in various production environments. In this section, we report the results of a comparative performance evaluation and a deployment of FCCE in a production environment. Moreover, we provide descriptions, results, and experiences of how we apply FCCE in cyber security analytics with two concrete use cases in one of our deployments and with a data set publicly available.

6.1 Storage and Retrieval Performance

We first compare the performance of FCCE and a conventional key-value store in terms of data storage and retrieval efficiency. We compared FCCE with Redis [12], a representative key-value storage system featuring state-of-the-art performance [28, 36]. To generate realistic workloads, we used Yahoo Cloud Serving Benchmark (YCSB) [25], a widely adopted framework for performance evaluation of key-value stores. YCSB offers common set of workloads consisting of CRUD operations, e.g., create, read, update, and delete.

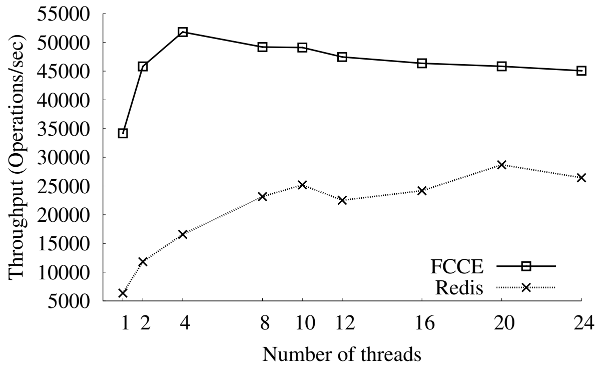


Figure 6: Throughput Comparisons

Setup: We performed our evaluation on a machine with two 2.53 GHz Intel Xeon quad core CPUs and 96 GB of RAM running CentOS 5.8. We used the latest stable version 2.8.6 of Redis with snapshotting disabled. We configured the workloads of YCSB for 60% reads, 30% updates, and 10% inserts, which were similar to the workloads of our DNS analytics. In order to avoid contention for resources with YCSB, we ran YCSB on a separate machine with two 2.67 GHz Intel Xeon hexa core CPUs and 96 GB of RAM running CentOS 5.8. Our data set consisted of 8 million 1 KB records, and overall 10 million operations were performed. The resulting size of the feature store in FCCE was 9.6 GB, and Redis consumed 20 GB of memory. We tested with varying number of client threads, e.g., 1, 2, 4, 8, 10, 12, 16, 20, and 24, to measure scalability with increasing amount of load.

Results: Figure 6 shows the throughput of FCCE and Redis. The highest throughput of FCCE was 51,807 operations/sec in 4 thread configuration, and the highest throughput of Redis was 28,693 operations/sec in the 20 thread configuration. Overall, FCCE achieved 1.6–5.4x higher throughput than Redis.

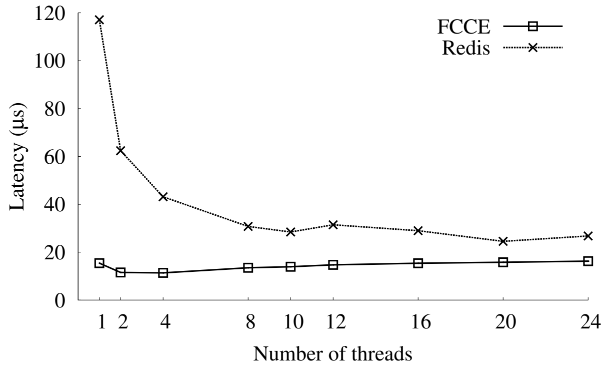
Figure 7 depicts read, update, and insert latency of FCCE and Redis. Latency was the average response time of operations, and is measured in microseconds. For all three kinds of operations, FCCE had much lower latency than Redis. For read and update operations, the latency of FCCE decreased as the number of threads increased up to 4. With 8 or more client threads, the latency of FCCE slightly increased and became stable, which explains the decrease in throughput. For insert operations, the latency of FCCE gradually decreased as we increased the number of client threads. For all three kinds of operations, the latency of Redis showed similar patterns, e.g., the latency steadily decreased with an exception of the increase at 12 thread configuration.

6.2 Real-world Deployment

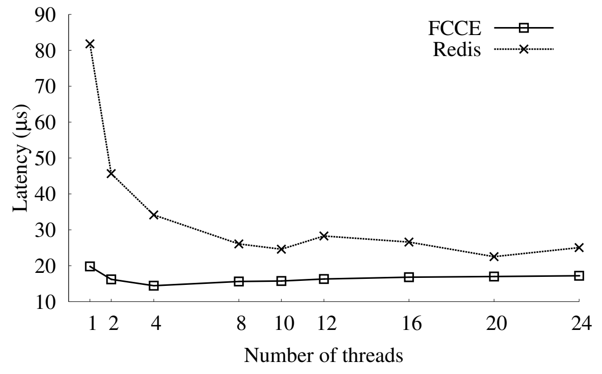
In one of our deployments of FCCE in a large production network with over 150,000 workstations, we ingest a variety of network monitoring related data. The nodes deployed are geographically distributed so that processing can be placed close to the data sources. Eight nodes are used for ingesting live raw data. The features are forwarded to a single central node which de-duplicates the merged stream of features. This is then forwarded to 5 nodes for final persistent storage. The persistent stores provide the historical query

Table 1: Currently deployed data sources and data rates.

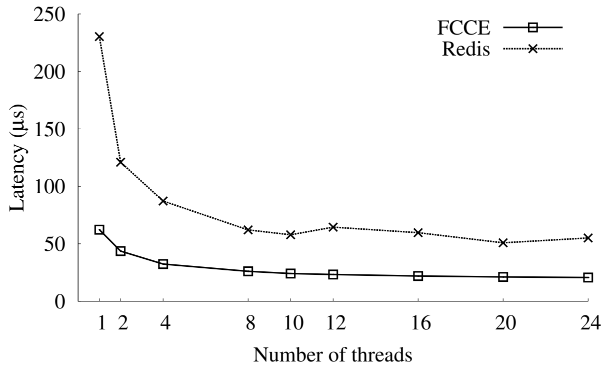
Data Type	**Number of sources**	**Events/day**	**Data/day**	
			gzipped	uncompressed
DNS messages	6	1250 M	64 GB	250 GB
Firewall logs	1	860 M	30 GB	200 GB
IPS alerts	2	26 M	4 GB	34 GB
HTTP headers	2	2 M	200 MB	1 GB
Proxy logs	1	200 M	30 GB	140 GB
VPN	1	1 M	40 MB	400 MB
Total		**2.3 B**	**128 GB**	**625 GB**



(a) Read Latency



(b) Update Latency



(c) Insert Latency

Figure 7: Latency Comparisons

service. The hardware resources of these nodes vary but all have at least 12 cores and 96GB of RAM. The persistent stores have fiber attached storage with multiple terabytes of disk space.

In Table 1, we report the different data types along with the number of sources (taps), the average number of inserted events per day, and the average amount of raw data processed per day. In addition, we also monitor a number of lower volume data feeds, such as DHCP and ARP traffic, as well as periodic batch data sets, such as web proxy logs, wireless network authentication logs, and workstation asset management databases. Incorporating all of these disparate data types into a single FCCE deployment as normalized features allows us to easily correlate across multiple features.

In the remainder of this section, we report performance metrics we measured in our deployment:

Insertion rate: In our real-world deployment, where we ingest DNS requests and replies, we measured insertion rates with a mean of over 113,000 events per second. In addition, in this environment, we observe periodic burst rates as high as 200,000 events per second, to which our deployment scales well.

Data reduction: To demonstrate the amount of reduction that can occur in our deployment, one month of DNS of data collected from one of our DNS taps was processed. The raw data consisted of more than 2 billion records, requiring as much as 240 GB of unstructured storage. The resulting feature store, including meta data to facilitate searching, used 6.6 GB and contained 300 million unique features.

Data retrieval and query response times: In contrast to a conventional key-value store, FCCE offers a set of unique functionalities, including feature collection and correlation. We thus further divide the empirical evaluation into two parts: one part that compares FCCE against alternative systems in their common functionalities, e.g., data storage and retrieval, in §6.1, and another part that evaluates the functionalities unique to FCCE, e.g., feature correlation in cyber security analytics, in §6.3.

6.3 FCCE in Cyber Security Analytics

As an example of how we can utilize our deployment of FCCE, we present two use cases. The first use case is a simple analytic for detecting fluxing domain names [33] over some time period indicating potential botnet activity. In the second use case, we demonstrate how we identify devices in our network looking up such fluxing domain names by correlating DNS lookups with DHCP and asset database information.

Detection of fluxing domain names: We implemented an application analytic on top of FCCE, that looks for DNS

Table 2: DNS data set for APT infection discovery

	# Queries	**# Responses**	**Size (gz)**
Feb. 2013	2,267,510,219	2,329,277,927	69.5 GB
Mar. 2013	1,539,510,477	1,555,669,803	48.3 GB
Total	3,807,020,696	3,884,947,730	117.8 GB

names that have multiple IP addresses across multiple countries. We acknowledge, that this analytic is too simplistic for exact matches in practice, but will return a small subset of candidate names for further in-depth analysis.

To achieve this extraction, our analytic first pulls all of the country codes from a date range. This query is covered by the DNS feature CCByDate. With the results from this query at hand, the analytic creates a filter CC, Date and queries for all the names in each country code on each date in the feature NameByCCDate. The analytic then reorganizes the result by domain names, and reduces the list down to those names that have more than some threshold set of countries associated with them. At this point, we have already reduced the set of candidate domain names to a much smaller set. Applying additional rules on IP addresses returned and corresponding autonomous system (AS) numbers, through similar queries, reduces the set further to the set of potential fluxing domain names.

We measured the query response times for the two main queries described above on a single HS22 blade in our evaluation testbed. The first query (CCByDate), over a range of 5 days in February 2012, took less than 1 second (on average) and returned 565 records. The second query (NameByCCDate) completed in 17 seconds (on average) and returned more than 500,000 records.

Identification of devices querying fluxing names: Having a set of fluxing domain names, it is critical to identify which devices in the monitored environment were querying them. Because most environments often utilize DHCP for assigning IP addresses, using only IP addresses to identify devices in historical data becomes ineffective. This is why feature correlation in FCCE is a crucial functionality in analytics. Using FCCE, we are able to correlate across various data sources (including DHCP messages), and create an identity trail for every DHCP-configured device.

For this purpose, we can ask for all the dates that a fluxing domain name was queried using DateByName feature. Using the results of this, we can ask for all of the clients using ClientByNameDate. This returns a set of IP addresses for the clients. The next translation is to hardware MAC addresses using MACByIPDate. Finally, using features extracted from a work station asset management database, we can request DeviceIDyMACDate, and identify the owner of the device using OwnerByDeviceIDDate.

The sequence of queries can be processed very quickly, even for a popular domain name. For example, in less than 20 seconds, we could identify all the clients who looked up "www.google.com." during the entire month of January 2012, and map their IP addresses to their MAC addresses, and then to their device IDs.

6.4 Applications

To evaluate the applicability of FCCE for processing diverse types of data, we applied FCCE to the advanced persistent threat (APT) infection discovery challenge using DNS data[1] hosted by Los Alamos National Laboratory. In the challenge, we were given two months of real DNS logs collected at a large site. The logs were 117.8 GB of gzipped logs consisting of 7.7 billion DNS queries and responses. Detailed breakup of the DNS data set is described in Table 2. The logs also contained DNS requests from several simulated APT attacks. The simulated APT attacks were performed in three steps: (a) infecting a host with an initial vector, such as phishing email with a link to a malicious site, (b) downloading and executing malicious software (malware), and (c) establishing command and control (C&C) communications, such as periodic callbacks to a C&C server. Then, the goal of the challenge was to develop techniques to detect malicious external domain names used in the simulated attacks and compromised hosts using the DNS logs.

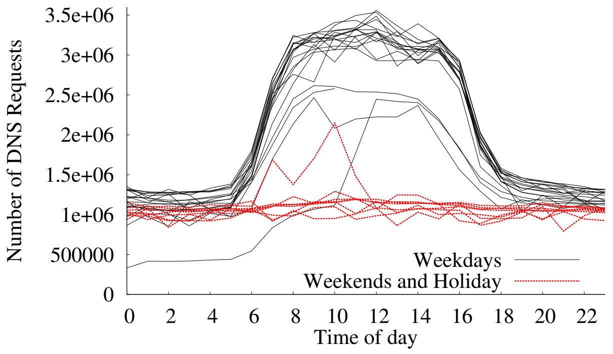


Figure 8: DNS requests during a day (February 2013 data was used, and each line denotes the number of requests for each day. NameByClient and DateByName features were correlated.)

We processed the "custom" DNS logs using a Perl script to extract features in a CSV format, and built a feature store using FCCE. The resulting feature store contained 9 features, including NameByClient and DateByName; the size of the feature store was 45 GB. The throughput of FCCE was 219,000 operations/sec when we built the feature store with eight extractors on a machine with two 2.40 GHz Intel Xeon deca core CPUs and 512 GB of RAM running CentOS 6.5.

In order to differentiate machine-generated DNS requests from human-generated DNS requests, we first investigated "business hours". By correlating NameByDate and ClientByName features, we plotted a graph showing the number of DNS requests during a day. As shown in Figure 8, the request pattern during weekdays was clearly distinguished from the request pattern during weekends, and business hours during weekdays were noticeable.

We then focused on detecting *periodic callbacks* generated by a machine at step (c) of the attacks. We observed that machine-generated periodic callbacks left DNS requests with regular intervals in DNS logs unlike human-generated DNS requests. We obtained DNS request history of a client using NameByClient feature while specifying a date using ++where query modifier. We then grouped domain names to acquire a list of timestamps of DNS queries for each domain name. It took 1.4 second to retrieve one day of query history of

[1]http://cps-vo.org/node/3453/og-panel/6

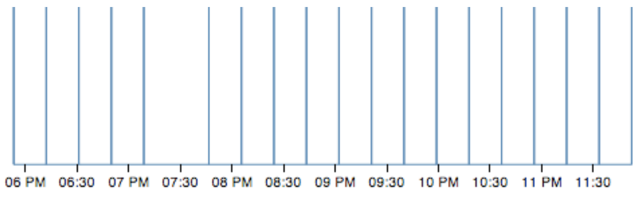


Figure 9: Periodic callbacks with a missing event (Beaconing pattern was detected using NameByClient feature.)

a client using FCCE (on average at a cold cache). Based on the list of timestamps, we calculated the number of DNS queries, the duration of DNS queries (e.g., how long the callbacks lasted), and the periodicity score of DNS queries for each domain name. When measuring a periodicity score, we adapted an algorithm from [35] to mitigate errors caused by oscillating and noisy (e.g., missing DNS request due to power outage) events. For example, Figure 9 shows periodic callbacks we detected where the first callback started at 5:53pm and lasted for 6 hours with about 19 minutes period. Note that one periodic callback was missing at 7:29pm. We flagged strictly periodic callbacks spanning over one hour.

In order to distinguish periodic callbacks by malicious programs from periodic requests by benign programs (e.g., cron jobs for software updates), we measured "reputation" of domain names. For this purpose, we used lifetime of domain names (DateByName), e.g., malicious domains might be used only for a short period of time to avoid blacklisting, and popularity (ClientByName), e.g., domain names queried by majority of hosts may be benign. One caveat is that our approach for measuring reputation was limited because the given DNS logs were not raw DNS data, instead they were pre-processed by dns_parse[2] and redacted to conceal the original domain names, IPs, and DNS server information. Therefore, we could not leverage other valuable information, such as external URL blacklists (e.g., spamhaus), country and Autonomous Systems distribution, Time To Live values (e.g., lower TTL for resilience against blacklisting), and string characteristics (e.g., random URL strings often generated by domain name generators of bots).

Finally, FCCE took 6 minutes using 40 processes to detect the simulated APT attacks from one day of DNS logs, where we checked 62,867 distinct clients against 177,190 domain names.

7. RELATED WORK

In the big data era, analytics systems are facing significant challenges of collecting, storing and analyzing enormous amount of historical and real-time data. In the cyber security domain, for instance, timely and in-depth response to security incidences often demands data storage systems to support both (near) real-time access to large volume of incoming data flow and fast retrieval of stored data spanning over large time windows. Existing work are often designed only for one of these requirements.

Traditional relational database management systems (e.g., MySQL, PostgreSQL, DB2, and Oracle) often do not scale well and suffer from significant storage overhead when performing large scope operations and transactions. Early work that attempted addressed the limitations focused on improving distributed query processing [34] and distributed transactions [32]. More recent work built highly scalable and available database clusters by partitioning and replicating tables over multiple servers, as exemplified by development of MySQL Cluster, VoltDB [39], Clustrix [6], ScaleDB [15] and ScaleBase [14]. Although these systems provide supports for rich schemas, SQL-like interfaces, and an ACID (atomicity, consistency, isolation, and durability) transaction support, they are not very efficient when executing large scope operations, e.g., join and transactions spanning several nodes, due to the communication and two-phase commit overhead.

Changes in data access patterns and needs for good scalability in simple read/write operations over large number of nodes motivated the paradigm shift towards so-called "NoSQL" data stores. NoSQL data stores feature a simple call-level protocol (as opposed to a SQL binding) and capabilities of efficiently using a distributed index for data storage [20]. NoSQL systems achieve much higher scalability and performance by relaxing ACID guarantees. For example, many systems [3, 29] provide only eventually consistentcy where updates are eventually propagated to all nodes; however, out-of-date read is possible during the propagation.

A widely used model among NoSQL systems is a *key-value store* where a key is associated with a value for all the data. Key-value stores deliver a high read/write speed with low latency. One of the popular key-value stores is Redis [12], an open source key-value *in-memory* database. Redis can achieve extremely high performance when the durability of data is not necessary (Redis offers snapshotting for data persistence). Similarly, Memcached [31], Membrain [9] and Couchbase [7] also feature in-memory indexing systems with different methods for persistence and replication. The main limitation of these in-memory key value store is that the capacity of the database is limited by the available physical memory. There are also many key value stores that persist data on hard drives. For instance, Tokyo Cabinet [16] is a high performance key-value store with various plugins to support reliable data persistence on different media. Project Voldemort [11] is an open source key-value store written in Java. It supports multi-version concurrency control for updates and transparent recovery of failure nodes. HyperDex [30] is a novel key-value store that exploits hyperspace hashing to provide a unique search primitive enabling not only retrieval based on primary key but also efficient queries on secondary attributes .

Basic key-value stores are extended by allowing storage of more complex or nested values. For example, Riak [13] is an open-source distributed key value store similar to Voldermort with support for JSON-based object and indices on primary keys. SimpleDB [2], Amazon's cloud DB service, groups data into domains, and multiple indexes are built on the attributes to support select operations. Amazon also offers a similar NoSQL database service called DynamoDB [1] with better scalability and low-latency responses. CouchDB [4] and MongoDB [10] are open source document stores which support "document collection" (similar to SimpleDB's domain) with richer data models. PNUTs [24] is a parallel and geographically data storage developed in Yahoo!. It achieves massive scalability by relaxing consistency guarantees and providing simpler relational model to users. The success of Google BigTable [22] motivated development of several similar systems, including HBase [19], HyperTable [8], and

[2]https://github.com/pflarr/dns_parse

Cassandra [3]. They share similar data models (rows and columns) and achieve high scalability by splitting both rows and columns over multiple nodes. They differ mainly in the their concurrency mechanisms. Specifically, Cassandra provides weak multi-version concurrency while HBase and HyperTable enable strong concurrency via locks and logging. Recently, there is a growing trend of building layer on top of existing NoSQL data stores to bring them functionally closer to traditional relational databases in terms of consistency and transaction support. For instance, Megastore [18] and Spanner [26] were two systems built by Google on top of BigTable and provide synchronous replication, strong consistency across datacenters and support for fully serializable ACID. In addition, Spanner also provides a SQL-based query language to facilitate the application development. G-Store [27] was built on top of HBase and provided multi-key consistency support (as oppose to the single-key atomic access provided in previous systems such as Bigtable, PNUTS, etc). CloudTPS [41] is a scalable transaction manager built on HBase and SimpleDB to allow cloud database services to execute ACID transactions of web applications. Spinnaker [37] utilizes Paxos-based replication protocol to provide either strong or time-line consistency on reads. Comparing with Cassandra which guarantees only weak eventually consistency, Spinnaker is shown to achieve similar read speed with minimum write overhead.

Both traditional DBMSs and recent NoSQL data stores are designed for storing relatively static records and processing queries and transactions. However, many applications e.g., security threat detection, market analysis often require support for near real-time processing of rapidly incoming data streams. To address these new challenges, a number of data stream management systems have been developed. For example, STREAM [40] was a stream data management system providing a SQL-like declarative query language. It also featured intelligent resource management and dynamic approximation of results based on current load. TelegraphCQ [21] implemented a data flow engine on top of the PostgreSQL code base, and supported continuous query processing over data stream with the use of highly adaptive query optimization techniques for evaluating multiple queries over data stream. NiagaraCQ [23] was a continuous query system designed for monitoring dynamic web contents. Its queries were expressed based on XML and XML-QL rather than SQL. Motivated by these research prototypes, some commercial streaming systems have also been developed. Notably, IBM InfoSphere Streams [38], Microsoft StreamInsight [17], and Twitter Storm [5] are all highly scalable platforms to support continuous analysis of massive data volumes and heterogeneous data types.

FCCE differs from previous systems in that its unique architecture design provides a unified interface for both historical and real-time analytics. At the core of FCCE, a distributed key-value store allows highly scalable and low-latency access to large volume of historical data. At the upper layer, the rich query and subscription interfaces provide versatile supports for a wide range of real-time analytic applications.

8. FUTURE WORK & CONCLUSIONS

We plan to extend the presented FCCE with several interesting features as we continue to push into the limits of its current capabilities:

- **Raw data archive and retrieval:** Once correlated features are retrieved, often-times it is desirable to inspect the related raw data. It is an open topic how to effectively reference and retrieve raw data that relates to query results. We currently able to create features that allow us to quickly identify the regions of files which contain the records we are interested in. However, our current compression methods for the raw data prevents rapid seeking to the regions. When a large number of raw data files must be accessed, the amount of time required becomes quite large.

- **Feature balancing:** We are including functions to monitor and automatically balance features across distributed feature stores to distribute load and further reduce latency of complex queries.

- **In-store correlation:** We are working to include in-store correlation capabilities into the feature store. This will further reduce latency and allow for highly-complex queries satisfying minimal latency.

In this paper, we presented the design, implementation, and evaluation of FCCE, a highly scalable, low-latency, and distributed feature collection and correlation engine. FCCE is built on top of a customized key-value data store and offers a comprehensive framework for efficient feature extraction, aggregation, storage, and retrieval. FCCE also provides a set of versatile APIs that allows efficient feature correlation across different data sets. FCCE has been deployed in our production network for over 2 years ingesting more than 2 billion network events per day and providing low-latency responses to queries for a variety of analytics jobs. In particular, we demonstrated, with real-world use cases, that FCCE yielded superior performance over existing approaches and excelled in the very challenging cyber security domain by helping to pull out security intelligence bits hiding in the noise of big data.

References

[1] Amazon DynamoDB. http://aws.amazon.com/dynamodb/. Page checked: 2/28/2014.

[2] Amazon SimpleDB. http://aws.amazon.com/simpledb/. Page checked: 2/28/2014.

[3] Apache Cassandra Project. http://cassandra.apache.org/. Page checked: 2/28/2014.

[4] Apache CouchDB. http://couchdb.apache.org/. Page checked: 2/28/2014.

[5] Apache Storm. http://storm.incubator.apache.org/. Page checked: 2/28/2014.

[6] ClustrixDB. http://www.clustrix.com/. Page checked: 2/28/2014.

[7] Couchbase. http://www.couchbase.com/. Page checked: 2/28/2014.

[8] Hypertable. http://hypertable.org/. Page checked: 2/28/2014.

[9] Membrain. http://www.sandisk.com/products/enterprise-software/membrain/. Page checked: 2/28/2014.

[10] mongoDB. http://www.mongodb.org/. Page checked: 2/28/2014.

[11] Project Voldemort: A distributed database. http://www.project-voldemort.com/. Page checked: 2/28/2014.

[12] Redis. http://redis.io/. Page checked: 2/28/2014.

[13] riak. http://basho.com/riak/. Page checked: 2/28/2014.

[14] ScaleBase. http://www.scalebase.com/. Page checked: 2/28/2014.

[15] ScaleDB. http://www.scaledb.com/. Page checked: 2/28/2014.

[16] TokyoCabinet. http://fallabs.com/tokyocabinet/. Page checked: 2/28/2014.

[17] M. Ali, B. Chandramouli, J. Goldstein, and R. Schindlauer. The extensibility framework in microsoft streaminsight. In *IEEE International Conference on Data Engineering*, pages 1242–1253, 2011.

[18] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Conference on Innovative Data system Research*, pages 223–234, 2011.

[19] D. Carstoiu, E. Lepadatu, and M. Gaspar. Hbase - non sql database, performances evaluation. *International Journal of Advancements in Computing Technology*, 2(5):42–52, 2010.

[20] R. Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2010.

[21] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *ACM SIGMOD International Conference on Management of Data*, pages 668–668, 2003.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, 2006.

[23] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for internet databases. In *ACM SIGMOD International Conference on Management of Data*, pages 379–390, 2000.

[24] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: yahoo!'s hosted data serving platform. *Very Large Data Base Endowment*, 1(2):1277–1288, 2008.

[25] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *USENIX Conference on Operating Systems Design and Implementation*, pages 251–264, 2012.

[27] S. Das, D. Agrawal, and A. El Abbadi. G-Store: a scalable data store for transactional multi key access in the cloud. In *ACM Symposium on Cloud Computing*, pages 163–174, 2010.

[28] DataStax Corporation. Benchmarking Top NoSQL Databases White Paper. http://www.datastax.com/resources/whitepapers /benchmarking-top-nosql-databases, 2013.

[29] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Symposium on Operating Systems Principles*, 41(6):205–220, 2007.

[30] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A distributed, searchable key-value store. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 25–36, 2012.

[31] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, 2004.

[32] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., 1st edition, 1992.

[33] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling. Measuring and Detecting Fast-Flux Service Networks. In *Network and Distributed System Security Symposium*, 2008.

[34] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[35] Z. Li, J. Wang, and J. Han. Mining event periodicity from incomplete observations. In *ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 444–452, 2012.

[36] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving big data challenges for enterprise application performance management. *Very Large Data Base Endowment*, 5(12):1724–1735, 2012.

[37] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *Very Large Data Base Endowment*, 4(4):243–254, 2011.

[38] R. Rea. IBM InfoSphere Streams: redefining real time analytic processing. http://public.dhe.ibm.com/software/data/sw-library/ii/whitepaper/InfoSphereStreamsWhitePaper.pdf. Page checked: 2/28/2014.

[39] M. Stonebraker and A. Weisberg. The VoltDB main memory dbms. *IEEE Data Engineering Bulletin*, 36(2):21–27, 2013.

[40] The STREAM Group. STREAM: the stanford stream data manager. Technical Report 2003-21, 2003.

[41] Z. Wei, G. Pierre, and C.-H. Chi. CloudTPS: Scalable transactions for Web applications in the cloud. *IEEE Transactions on Services Computing*, 5(4):525–539, 2012.