CSA: Streaming Engine for Internet of Things

Zhitao Shen[*], Vikram Kumaran[*], Michael J. Franklin[†], Sailesh Krishnamurthy[‡], Amit Bhat[*], Madhu Kumar[*], Robert Lerche[*] and Kim Macpherson[*]

[*]Cisco Systems, Inc
{zhitshen,vkumaran,amibhat,madhuku,rlerche,kimacphe}@cisco.com
[†]University of California, Berkeley
franklin@cs.berkeley.edu
[‡]Amazon Web Services, Inc.
sailesh@gmail.com

Abstract

The next generation Internet will contain a multitude of geographically distributed, connected devices continuously generating data streams, and will require new data processing architectures that can handle the challenges of heterogeneity, distribution, latency and bandwidth. Stream query processing is natural technology for use in IOT applications, and embedding such processing in the network enables processing to be placed closer to the sources of data in widely distributed environments. We propose such a distributed architecture for Internet of Things (IoT) applications based on Cisco's Connected Streaming Analytics platform (CSA). In this paper describe this architecture and explain in detail how the capabilities built in the platform address real world IoT analytics challenges.

1 Introduction

By some estimates the number of connected devices will approach 50 Billion by 2020 [1]. The *Internet of Things* (IoT), driven by the explosion in number of end points that will join the Internet, has become a popular movement in the industry today. Many recent papers have outlined the challenges introduced by the IoT (e.g.,, [2, 3, 4]). In this paper, we focus on the challenges related to data handling and processing in such an environment. The amount of data generated scales with the number of devices, leading to potentially huge data volumes. Current elastic cloud capabilities give us the ability to store and process large volumes of data, but given the rate, scale and distribution of data generated by IoT devices, processing all the data in the cloud might might not be feasible. Fortunately, however, not every sensor reading is equally important and by processing data near the point of generation, it is possible to make intelligent trade-offs among data fidelity, latency, bandwidth and resources.

For example, in offshore oil fields the volume of data generated typically exceeds the bandwidth available [5]. Intelligent data reduction near the source can solve this problem with minimal loss of information. In other industries such as manufacturing and transportation, where the devices connected to the network are in rapid motion through space, any data generated needs to be analyzed in context with minimum latency to be useful [6].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

In such situations there is very little leeway in where data is processed. Shipping data back and forth to a central cloud infrastructure over the wide area network is unacceptable to due the challenges of latency and unreliable communication links.

Another challenge posed by the IoT is data privacy, with the need for policy-based restrictions on what gets sent out from devices [2]. Finally, devices connected to the Internet display tremendous heterogeneity in communication protocols, formats and content. To deal with this variety in data sources, one needs intelligence near the data source to translate into a common representation for the system as a whole to be able to work together [2]. The challenges described above are present in most real-life IoT deployments and need to be considered for any successful solution.

Given that IoT devices typically generate streams of data, the ability to process those streams and to be able to correlate and join heterogeneous data streams as they are generated are critical capabilities. The devices generating data out in the field connect to a network gateway. A stream-processing engine, present at that edge gateway and embedded in the network in a high fan-in system [7], is an effective architecture for supporting these applications. Cisco's Connected Streaming Analytics (CSA) provides an embeddable platform capable of processing individual streams as well as stream-stream correlations and joins. It also supports tracking of numerous independent, concurrent, data sessions, making it an ideal platform for an IoT analytics architecture.

There are many use cases across multiple verticals that highlight how stream processing can address real-world IoT challenges.

- The oil and gas industry is increasingly being digitized, with sensors measuring the state of the entire operation around the clock. However operations are typically in remote areas that have poor connectivity, having limited bandwidth and relatively unreliable networks [8, 9]. The volume of data generated by an oil and gas operation runs into gigabytes per second, and it is a losing proposition to move the raw data into a traditional data store. Stream processing helps with intelligent data reduction at the network edge by picking salient features and sending only necessary data for central processing.

- Communications network operators are increasingly reliant on the analysis of real-time network telemetry for providing a disruption-free network. Traditional big data approaches focused scalability and are driven by data volume. However, in a network, the limiting factor is data movement, as using the network to move telemetry puts tremendous strain on its core function, namely, transmitting user data. A distributed analytics solution with in-stream analysis performed at the data sources, embedded on network devices, alleviates this problem [10].

- Another industry being transformed by IoT is manufacturing. Robots and machines, building products we use everyday, are increasingly being instrumented with sensors that continuously measure operation parameters. Distributed control is not a new concept in manufacturing, however current distributed control systems are proprietary boxes with many I/O control points [11]. In the new world of IoT, sensors and actuators are constantly being added and a truly distributed control system needs to be a platform that can incrementally grow in capability and capacity. A stream-processing platform that can run at the edge on gateway routers and switches connecting robots and machines can provide a low-latency open platform on which to build incremental analytics as new data sources and algorithms are developed.

In the industry today many of the architectures proposed to handle the challenges created by an Internet of Things are based on an assumption that all the data will reach and reside in the cloud [4, 12]. The elastic scalability of cloud infrastructure is an attractive solution to the challenge of unprecedented data volume from billions of sensors. We believe, however, that the world of IoT will evolve a very different architecture, primarily due to the challenges described in the previous paragraphs. IoT will not have the luxury of unlimited bandwidth, latency and connectivity in many real-life situations. Current proposed solutions treat software at the edges of the network as simple data accumulators with the primary purpose of shipping data to a central data center for off-line analysis and human consumption. While there are parts of use cases where that assumption might apply, we strongly believe that across various industry verticals data needs to be processed at the right level of context. In other words the intelligence needed to analyze and process data needs to exist throughout the network in a high fan-in system.

CSA is an advanced stream-processing engine based on the Truviso technology [13, 14] that has been extended with the capability to run embedded in network elements as well as in other parts of the network and the cloud. This approach enables an architecture where intelligence can be located across the network and placed as close to the data source as desired. This architecture is supported by several key features built into CSA:

- **Stream correlation using joins**. Streaming joins are very useful to correlate streams from both homogeneous and heterogeneous data sources. In many IoT cases, streaming joins can be performed at the network edge, as the data sources are mostly geographically correlated. One challenge for stream correlation is out-of-order data arrival due to the complexity of network environment in IoT and because devices may have different latencies for generating streams. The original Truviso system had limited join facilities but we further extend these to support two types of streaming joins (*best-effort* joins and *correlated joins*) to handle time-alignment differences between streams.
- **Session windows**. In CSA, we implement a new type of window operator to support session-based analysis. Sessionization is critical for IoT applications and can be used to correlate streams from homogeneous sources as well as to monitor complex events and on-going status over a single logical stream that is fed by multiple threads of data events coming from multiple sources.
- **Edge processing via containers**. In CSA, we create a low-footprint version of the stream-processing engine that has been ported to run in Cisco's routers and switches at the edge. CSA is built into available secondary compute resources [15] in a container, which enables analytics applications to run on routers and switches. Consequently, no additional hardware beyond the routers and switches is required to retrieve and process the streams from network-connected devices for edge analytics. One key benefit of edge analytics on network devices is that stream processing can scale with network size. The computational complexity for each edge node can be considered as bounded, as the number of devices connected in the sub-network is limited by capacity of the network gateway device.
- **Built-in time-series algorithms**. CSA provides additional machine learning algorithms that can operate over time-series streams for handling common IoT use cases. For example, we implemented an algorithm to discover periodic patterns over time-series data. Also, we can use an ARIMA (Autoregressive Integrated Moving Average) model for forecasting sensor values based on time-series streams.

In the remainder of this paper, we describe the overall architecture we have developed for edge analytics using CSA and focus on the new features listed above. Due to space constraints, however, we do not address time-series algorithms, which we plan to address in a later publication.

2 System Overview

2.1 Architecture

Figure 1 depicts A high-level overview of the architecture for distributed streaming intelligence in the network. The components of the architecture are deployed on a distributed infrastructure. At the edges of the network, the gateway routers and switches connect to sensors and devices that are the sources of data. The edge gateways have spare compute resources that can be used to run data processing applications in the containers. A little higher up in the stack are the fog nodes [16], having somewhat more compute and storage than the edge gateways. They

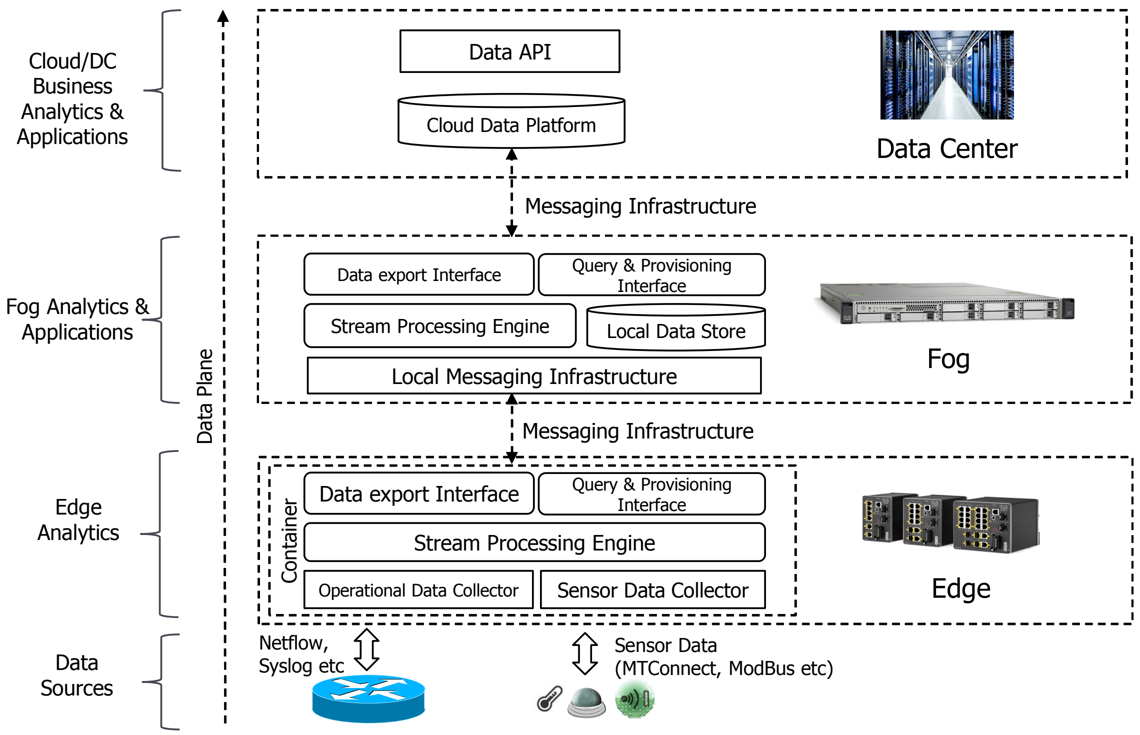

Figure 1: Edge Analytics Architecture

extend the cloud computing paradigm closer to the edge of the network. Further up the stack we get to the cloud or data center, where we have virtually unlimited scalability in terms of compute, storage and network. Cisco's CSA platform is software that can run on all the different levels in the hierarchy with appropriate resource-constrained capability. The main components of this architecture across the hierarchy are as follows:

Data collectors. The sensors and operational data are generated in a variety of protocols and content formats. There are no common standards for sensors and devices participating in the Internet of Things. This lacuna creates the need to have custom adapters to hide device heterogeneity and convert custom data streams into standard format. The *data collector* is a modular library of such adapters that will grow to handle the variations across the industry. The data collectors typically run on spare compute resources available in the edge gateways.

Stream processing engine. The data inputs transformed by the data collectors are processed by the *stream-processing engine*. When running at the edge, stream processing consists typically of simple aggregations, filtering, grouping, joins, local model scoring and prediction. As we move up the deployment hierarchy stream processing engines take on more complex computational tasks. Some of the key fundamental capabilities of the stream engine are discussed in detail in the later sections of this paper.

Query and provisioning interface. The interface remotely manages and monitors raw and derived streams in the engine. The platform is truly distributed and this interface provides a programming interface for remote administration.

Messaging infrastructure. Processing components of the architecture are sometimes distributed within a local area network and in many instances over a wide area network. In this architecture the stream processing engine described above exists in the data path. It consumes raw streams and emits processed data streams. The *messaging infrastructure* connects the various stream processing engines and provides an infrastructure to orchestrate the data flow.

Cloud data platform. In some applications the results of stream processing interact with devices and machines directly in a closed loop; for example, to change policy and influence actions. However, in most real world use cases, human intervention at a central location based on the visible state of the environment is still necessary. The *cloud data platform* provides the ability to combine real-time and historic data, operational data and business data for longer term visibility.

2.2 Streams and Queries

Connected Streaming Analytics (based on the Truviso engine) is designed to manage *streams* (i.e., unbounded, growing, data sets) in addition to relations (i.e., finite data sets) of the kind managed by a traditional RDBMS. CSA allows for these objects (i.e., streams and relations) to be created and queried in standard, full-featured SQL[1] of the sort supported by a typical RDBMS in an integrated fashion [13]. A SQL query that operates over one or more streams produces a continuous stream of results, and is therefore called a *continuous query*. The notion of continuous query is in contrast to a standard SQL query that operates exclusively over relations from a static view of a database and produces a finite data set (another relation) as its output. We refer to such a traditional query as a *static query*.

Streams. A continuous SQL query takes relations and streams as input, and produces streams as output. Unlike relations in a traditional database, a stream can be thought of as an unbounded bag of tuples, traveling though a network, where each tuple has a delineated timestamp attribute. A stream, like a table, is a database object that has an associated schema that defines the format of the data.

Raw and Derived Streams. In CSA, streams can be categorized as two different types depending how they are populated. *Raw streams* are populated by external data sources. A tuple in a raw stream can represent an event or state of the real world at a particular timestamp. *Derived streams* are defined using a continuous query on a raw stream or other derived streams, and populated by CSA.

Aggregation in CSA is computed in a shared fashion [17] and is therefore memory efficient. Additionally, CSA provides the capability of order-independent processing [14] and is useful for handling the out-of-order data appearing in real life IoT applications. In the reminder of this section, we briefly introduce the CSA query language.

Query Language. In CSA, queries can be posed exclusively on relations, exclusively on streams, or on a combination of streams and relations. Since a stream is unbounded, a streaming query that produces a stream never ends and, as stated above, is therefore called a *continuous query* (CQ). The only extension to the standard SQL syntax is a set of window (stream-to-relation) operators.

In order to process an unbounded stream of data, stream-processing engines apply windows that segment the stream into discrete finite data sets. CSA provides rich windowing semantics to support a variety of window definitions. For raw streams, windows may be either time-based (a specified interval of time, e.g. '1 minute') or row-based (a specified number of rows) depending on the need of the query. Derived streams, in addition to row and time windows, can define window-based windows, where the window size is specified as a number of windows in the underlying stream. Window based windows provide a level of abstraction, allowing the properties of a higher-level query to be specified in terms of the windows used by a lower-level query.

CSA offers a wide range of window (stream to relation) operators.

- **Chunking windows**: A *chunking window* is also known as a tumbling window. With chunking windows, the underlying stream is broken into successive, contiguous, and non-overlapping "chunks" of tuples.
- **Sliding windows**: A *sliding window* is expressed using an *advance* interval, and a *visible* interval. The former defines the periodic intervals (and thus the actual window edges) at which a new visible set is

[1]Note that only some of the non-monotonic SQL queries are supported in streaming fashion. For example, EXCEPT is not supported as a continuous query, while in most cases we can rewrite queries using a join operation if we only provide distinct tuples as results.

```
SELECT device_id, count(*) AS err_count
FROM message <SLICES '1 minute'>
WHERE type = 'ERROR'
GROUP BY device_id
ORDER BY err_count DESC
LIMIT 10
```

Figure 2: A Simple Continuous Query

constructed from the stream, while the latter defines the interval of tuples, relative to the periodic edges, that belong in each visible set. Note that both intervals can be either time-based or row-based intervals.

- **Landmark windows**: A *landmark window* is expressed using an advance interval, and a *reset* interval. The former defines the periodic interval ("advance" edges) at which a new visible set is constructed from the stream, while the latter defines a periodic interval that is used to compute a sequence of "reset" edges. Each visible set consists of all tuples that have arrived in the stream after the latest reset edge.
- **Session windows**: A *session window* correlates all tuples belonging to a given group whose time interval between consecutive tuples does not exceed a given timeout value. This window is useful to identify tuple sequences whose total duration is unknown in advance. The details of session windows and its application are discussed in Section 3.2.

Figure 2 shows a simple continuous query to find the top-10 devices with the most error messages in the past minute. <SLICES '1 minute'> defines a 1-minute chunking window and we conceptually transform all the messages in the past 1 minute into a relation via the window operation. Upon this resulting relation, the top-10 answers can be calculated by the standard SQL query using grouping and ordering.

2.3 Out-of-Order and Delayed Streams

Most stream-processing systems from both academia and industry assume that input streams arrive in order. This assumption is usually not true in real environments even for a single data source. For instance, data transportation with UDP packets may cause out-of-order delivery. In IoT environments, out of order data is the norm. One typical approach used to deal with this issue is to have the system rely on the physical order of streams. Tuples are timestamped using the clock time when they arrive. However, this approach is likely to produce incorrect results when trying to detect event sequences. Exact ordering is required for sequence matching.

Delayed streams are slightly different from out-of-order arrival. They can occur even if each independent data source is in order. The possible reasons are for delayed streams are: 1) the clocks of the local sources are not synchronized. 2) Network latencies are different from different sources to the engine. 3) The source device may encounter a delay while producing streaming data.

In CSA, we have two ways to handle correlating queries over out-of-order or delayed streams: 1) buffer-and-reorder mechanisms can reorder streams before feeding order-sensitive operators such as sessionization windows, for example, slack and drift [14] can be used to handle the streams with small degrees of out-of-orderness. 2) Coordinated joins can be used to correlate streams in a time-aligned fashion. We discuss these techniques in the following section.

3 Correlation, Sessionization and Joins

One of challenges for IoT analytics is the ability to correlate records from a single data source or multiple data sources. The typical streaming queries for correlating records are the join operators inherited from the relational database world, and pattern matching usually used for complex-event processing [18]. In this section, we introduce how correlating queries are performed in CSA: streaming joins and the sessionization window operator supporting pattern matching as well as complex-event processing.

3.1 Streaming Joins

Streaming join is a fundamental operation for relating information from different streams. Over the last decade, a much previous work has focused on *sliding-window joins* [19, 20]. As streams are potentially unbounded, an obvious issue of un-windowed streaming joins is that the join state grows continuously and will eventually outgrow memory. Therefore, windows are usually applied to the input streams to restrict the scope of the join. Continuous Query Language (CQL for short) [21] specifies the semantics of a sliding-window streaming join by treating it as a view of a relational join over the sliding windows.

Consider the challenge for time alignment in IoT applications. In CSA, joins can be performed in two ways: *best-effort* and *coordinated*.

Best-effort Joins. In best-effort fashion, the join is processed immediately once a window is emitted when the end of the window (specified for example, in time or as a number of records) is reached. The window is joined against the most recent windows of the other join inputs. The idea behind the best-effort joins is, where possible, to generate the join results with minimum latency. Basically, best-effort can accept slightly out-of-order data and is useful when the skewness of the multi-source streams is low.

In CSA, if any of the inputs to a join is a row-based or window-based window, best-effort joins are performed. As each window of the join operand is received, the window is joined against the most recent windows of other input streams. Hence, the results of best-effort joins can be non-deterministic and will depend on the order in which the input streams' windows arrive.

```
SELECT
  s.device_id, s.torque
FROM
  sensor s <VISIBLE 1000 rows
            ADVANCE 1 row>,
  message m <SLICES 1 row>
WHERE
  s.device_id = m.device_id AND
  s.torque > 100 AND
  m.type='ERROR'
```

Figure 3: Example for Best-effort Joins

```
SELECT
 s.device_id, s.torque
DEOM
  sensor s <VISIBLE '1 minute'
            ADVANCE '1 second'>,
  message m <SLICES '1 second'>
WHERE
  s.device_id = e.device_id AND
  s.torque > 100 AND
  m.type = 'ERROR'
```

Figure 4: Example for Coordinated Joins

An example of best-effort join is shown in Figure 3. This join specifies that a tuple from the message stream joins with the last 1000 tuples from the sensor stream. As it is a best-effort join, the join results will output whenever a new record arrives form either stream. In this example, the join outputs a result whenever an error message occurs after an abnormal sensor reading (e.g., torque too high) is observed for the same device.

Coordinated Joins. In many Internet of Things applications, timestamps from the stream are generated by the edge device collecting or generating the data. However the stream-processing engine might not receive the tuples in the order of timestamps due to (for among other reasons) latency in the network. In such a case, correlating streams in a time-aligned fashion is important. To this end, the join usually is processed in a synchronized order from multiple stream sources. Unlike best-effort joins, time plays a very important role for coordinated joins. To perform coordinated joins, the join operator will ensure that when a window of one of the input streams arrives, it is joined against the latest possible windows of the other streams according to their respective timestamps. In CSA, if all the inputs to the streaming join are time-based and their timestamps are in the same domain, coordinated joins are enabled.

An example of a coordinated join is shown in Figure 4. Similar to the best-effort join, this query shows how we join records from two streams. As both windows are time-based, the join is performed in a time-aligned fashion, that is, for a certain time *t*, the join will match the the records exactly between $t - 1$ *min* and *t* from the both streams based on the timestamps included in the data. Unlike best-effort joins, coordinated joins require buffering the tuples from the faster streams.

Coordinated joins are commonly used when one of the input streams is a derived stream with order-sensitive operators (e.g., aggregations), since the exact statistic or status for a certain time point is required.

3.2 Sessionization

The fixed-interval window operations (e.g., chunking, sliding, landmark) allow aggregates to be computed over data stream segments that are demarcated by a predetermined (user-specified) time interval or record count. While such windows enable many useful types of analytics, the rigidity of a given window size (time or row-based) can be too restrictive in situations where the segments of a data stream over which to run analytics are not known in advance. We developed techniques for *sessionization* to overcome such rigidity. Sessionization provides a way to operate on independent data event threads or sessions, each having its own independent window segments. We first describe the syntax of sessionization and then provide a simple example to introduce the main building blocks of CSA sessionization.

```
< SESSION session_key[, ...]
  TIMEOUT interval | NONE
 [EXPIRE WHEN conditions
             [RETAIN EDGE]]
 [ADVANCE interval
  OR
  ADVANCE WHEN conditions]
>
```

Figure 5: Syntax of Sessionization

```
SELECT
  device_id,
  FIRST(date_time) AS start_time,
  cq_close(*) AS check_time
FROM robot
  <SESSION device_id TIMEOUT NONE
   EXPIRE WHEN (FIRST(type) != 'START' OR
   LAST(type) != 'START')
   ADVANCE '1 second'>
GROUP BY device_id
HAVING
  cq_close(*)-first(date_time)>'10 minutes'
```

Figure 6: Example for Session Query

The details of the syntax of sessionization are shown in Figure 5. The session_key after the SESSION keyword specifies the keys for identifying the sessions. These keys should be same as those in the expressions for any GROUP BY clause in the stream queries and GROUP BY is mandatory when using session windows. Session windows are defined based on the semantic expressions rather than on fixed time intervals. Since sessions often do not have an explicit end record, the TIMEOUT clause specifies a timeout (expiring) interval for sessions that have no further associated tuples. If an EXPIRE WHEN condition is specified and is satisfied by the arrival of a tuple, then the session that the tuple belongs to is expired. If the optional RETAIN EDGE clause is specified, then after expiry, a new session is started with the current tuple as its first record. If an ADVANCE clause with a time interval is specified in a session definition, the session aggregation emits a result triggered by a time interval. For example ADVANCE '5 minutes' will cause the aggregation to emit a result every 5 minutes. If an ADVANCE WHEN condition is specified and is satisfied by the arrival of a tuple, then a result (projected in the SELECT clause of a stream definition) is emitted.

Consider the example of manufacturing in IoT. Suppose that we want to monitor a robot's status. When a robot is started, its START message will be sent out once. But sometimes the robot runs into a failure state and nothing is sent out after that. We need to detect such situation and reboot the device.

The session window definition in Figure 6 continuously computes sessions based on the individual device_id of a robot. The session starts only when we receive a START message and will be expired when we receive a non-START message[2]. According to the ADVANCE clause, the calculation occurs every second and only the sessions with duration longer than 10 minutes will be emitted, as specified in the HAVING clause.

There are several key features of CSA's sessionization:

- It enables precise metric computation over the sessions. It supports per-session expiry as well as result generation based on semantics that a user can specify as an aggregate (or even a complex combination of aggregates) as opposed to simply specifying rules based on the attribute(s) of a single tuple. Such semantics include not only CSA's built-in aggregates, but also any user-defined aggregates, including those that can do pattern-matching.
- Unlike the pattern-matching approach used in other stream-processing systems, the ADVANCE clause provides the ability to peek into ongoing activity for the sessions. For example, we can list all the ongoing sessions for each hour and compute an aggregation on top of it.
- Sessionization provides a TIMEOUT clause to expire the sessions which are not active for a certain period of time.
- Sessionization processing is memory-efficient, since we manage sessions within the shared aggregation infrastructure [17]. Also, we can avoid storing many of the tuples of a window in memory. Aggregate states are maintained for each session. The memory usage of sessionization depends on the number of concurrent sessions, not on their individual length.
- As each session maintains it own state, sessionization can easily scale out to multiple instances of the stream-processing engine for parallel computation. Key-based partitioning can be utilized to distribute data into multiple instances.

3.3 Applicability for the Internet of Things

In real world IoT analytics applications, we have to cope with challenges such as heterogeneous sources, differences in data formatting and temporal alignment of the streams. Joins and sessionization are useful for addressing these challenges posed by sensor data in an IoT deployment.

Integrating Homogeneous Data Sources Homogeneous data sources can be found in IoT deployments when similar devices and sensors are geographically collocated. Similar devices generate events with similar data schemas. In CSA, we suggest having single raw stream for homogeneous data schema from multiple sources. This greatly simplifies application of correlated queries, such as sessionization and joins.

Vertically Partitioned Data In many IoT protocol standards, the data is vertically partitioned. For example, a typical schema for the sensor stream includes {Timestamp, Type, Sub Type, Name, Id, Sequence, Value}. Self-joins on the stream are typically used to flatten attributes (correlate partitioned values) for the same device within a small time window.

Integrating Heterogeneous Data Sources Multiple data streams generated from a variety of devices and sensors are in many cases heterogeneous in their data schemas. A typical example is a machine that generates both sensor streams of physical measurements and event streams of state changes. For such situations, we can employ correlating queries on these streams, as we have shown in the preceding example. For heterogeneous data sources, separate raw streams are suggested. Streaming joins can be used to correlate the records from multiple streams.

[2]It is possible that only one START message is received. Usually heartbeats (punctuation) can be utilized to make the stream advance.

4 Edge Processing via Containers

Unlike traditional data-warehouse solutions where all data is collected and stored in a centralized place, the architecture we propose enables computation to be placed throughout the network, including at the edge. The CSA stream-processing engine is deployed on network-edge gateway routers and switches. Many of these edge gateways have spare compute and memory that can be exploited for non-network operations. The streaming engine is optimized for running in such constrained environments, and as majority of the processing is done in memory, there is very limited dependency on disk storage. In typical deployments, the CSA stream-processing engine runs inside a Linux container that is provided as a part of a Cisco edge gateway [15]. The container is hosted on a Cisco network device. Consequently, no additional hardware is required to retrieve and process the streams from network-connected devices for edge analytics.

There are several advantages to deploying the stream processing engine into a Linux container on a edge gateway: 1) The resources used by processes at the network edge are in a controlled space and the container reserves the essential resources for the network operation. 2) The application is isolated from the network OS which helps give security guarantees, 3) Linux containers are lightweight and fast for deployment. Running applications in the container is more efficient than running in a VM. Additionally, packaging CSA within a container image helps us to deploy applications into devices located in different layers of network (edge, fog and cloud) without much effort.

Another key benefit of edge processing on network devices is that streaming analytics can scale with network size. The number of devices connected in a sub-network can be considered bounded, since network devices normally have limited capacity and can only afford finite device connections. Therefore, we can consider the computational demand on each edge node to be bounded.

Besides efficiency in both network bandwidth and latency, edge processing is also very important for privacy. In many IoT applications such as Smart Cities, we are only allowed to bring processing to the streams and can expose only summaries or conclusions rather than raw data. Also, we can scrub and validate the data to be stored in data centers. For example, we can use CSA to anonymize sensitive personal information (information that can be used to identify a person, e.g., client MAC addresses) on the fly at the network edge, and expose only the data that is allowed to be stored in a data center according to local privacy laws.

5 Related Work

The Internet of Things and related applications have been widely studied [2, 3, 4]. However, streaming analytics in the context of Internet of Things is only starting to receive much attention. Aggarwal et al. [22] discuss how RDF streams can be handled with RDF queries and big-data facilities. Sheykh Esmaili [23] investigated event detection and FPGA implementation for embedded environments but the system described is not a fully functional stream-processing engine and is limited in its capabilities.

Earlier work on sliding-window joins[19, 20] does not consider time alignment and out-of-order events, which are widely observed in the real world. Li et al.[24] discuss out-of-order processing for stream joins. Recently, researchers from industry have been studying streaming joins in their respective contexts. Photon [25] from Google applies streaming joins to continuously combine a click event from a click log with its corresponding query event from a separate query log. Photon leverages distributed computing infrastructure from Google and joins are processed through different data centers. However, Photon is specifically designed for joining click and query streams and is not optimized for general streaming-join purposes. Also, sliding windows are not explicitly defined for joining. In CSA, we propose both coordinated joins and best-effort joins to cope with the challenges in correlating multiple data sources.

While sessionization was originally introduced for Web analysis [26], few implementations perform sessionization over streaming data. Akidau et al.[27] define session windows in a dataflow model. However, only timeouts are provided for grouping tuples into sessions. Also related to sessionization are event pattern-matching and Complex Event Processing (CEP) for event streams. SASE [18] and Cayuga [28] are examples of systems supporting CEP over event streams. These systems usually provide a NFA-based pattern matching implementation. A key difference between these systems and CSA is that they treat event processing as distinct from traditional Relational query processing. In comparison, CSA is an extension of a traditional database system so it can leverage existing feature sets (e.g. user defined functions and database extension) in the Relational world and can easily combine streaming and static data. Sessionization in CSA is also efficient as we reuse an existing aggregation framework [17]. NiagaraST [29] proposed T+D frames which are similar to session windows, but did not specify a full-featured query language.

6 Conclusions

Modern Internet of Things applications are pushing traditional database and data warehousing technologies beyond their limits due to the explosive increase in data volumes, distributed data creation and requirements for low latency. To address these issues, we advocate an architecture deploying the Connected Streaming Analytics (CSA) engine, inside throughout the network, on edge gateways, fog nodes and on data center machines. This architecture enables a variety of new IoT applications.

CSA provides a query language for continuous queries over streams that supports various window operators, efficient shared aggregations, the functionality of an integrated relational database, out-of-order stream processing and correlation queries such as streaming joins and sessionization. In this paper, we showed how streaming joins and sessionization support correlating heterogeneous data sources from the Internet of Things. The features provided by CSA can solve important challenges in real world applications such as temporal alignment for heterogeneous sources. For network edge processing, we deploy CSA in a Linux container on network devices. The marriage of networking capabilites with stream query processing is unique and, we believe, can change how we analyze data created by connected things in the emerging world of IoT.

References

[1] "Connections counter: The Internet of Everything in motion." http://newsroom.cisco.com/feature-content?type=webcontent&articleId=1208342, 2013.

[2] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[3] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the Internet of Things: A survey," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 1, pp. 414–454, 2014.

[4] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[5] M. P. Mills, "Shale 2.0: Technology and the coming big-data revolution in america's shale oil fields." http://www.manhattan-institute.org/html/eper_16.htm#.VgFwvnvShHK, May 2015.

[6] A. Pye, "Mining's drive for efficiency," *Engineering & Technology*, vol. 10, no. 5, pp. 80–83, 2015.

[7] M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong, "Design considerations for high fan-in systems: The HiFi approach," in *CIDR*, pp. 290–304, 2005.

[8] D. Bigos, "5 ways IoT technologies are enabling the oil and gas industry." http://www.ibmbigdatahub.com/blog/5-ways-iot-technologies-are-enabling-oil-and-gas-industry, June 2015.

[9] "Creating digital oil fields and connected refineries." http://www.cisco.com/web/strategy/energy/external_oil.html", 2015.

[10] A. Clemm, M. Chandramouli, and S. Krishnamurthy, "DNA: An SDN framework for distributed network analytics," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pp. 9–17, IEEE, 2015.

[11] D. Brandl, "Distributed controls in the Internet of Things create control engineering resources."

`http://www.controleng.com/single-article/distributed-controls-in-the-internet-of-things-create-control-engineering-resources/fb0eea6e0b9d0d8cd97aad4025e5c080.html`, June 2014.

[12] A. Alamri, W. S. Ansari, M. M. Hassan, M. S. Hossain, A. Alelaiwi, and M. A. Hossain, "A survey on sensor-cloud: architecture, applications, and approaches," *International Journal of Distributed Sensor Networks*, vol. 2013, 2013.

[13] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre, "Continuous analytics: Rethinking query processing in a network-effect world," in *CIDR*, www.cidrdb.org, 2009.

[14] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre, "Continuous analytics over discontinuous streams," in *SIGMOD Conference*, pp. 1081–1092, ACM, 2010.

[15] P. Jensen, "Cisco fog computing solutions: Unleash the power of the Internet of Things." `http://www.cisco.com/web/solutions/trends/iot/docs/computing-solutions.pdf`, May 2015.

[16] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for Internet of Things and analytics," in *Big Data and Internet of Things: A Roadmap for Smart Environments*, pp. 169–186, Springer, 2014.

[17] S. Krishnamurthy, C. Wu, and M. J. Franklin, "On-the-fly sharing for streamed aggregation," in *SIGMOD Conference* (S. Chaudhuri, V. Hristidis, and N. Polyzotis, eds.), pp. 623–634, ACM, 2006.

[18] D. Gyllstrom, E. W. 0002, H.-J. Chae, Y. Diao, P. Stahlberg, and G. Anderson, "SASE: Complex event processing over streams (demo)," in *CIDR*, pp. 407–411, www.cidrdb.org, 2007.

[19] L. Golab and M. T. Özsu, "Processing sliding window multi-joins in continuous queries over data streams," in *VLDB*, pp. 500–511, 2003.

[20] U. Srivastava and J. Widom, "Memory-limited execution of windowed stream joins," in *VLDB*, pp. 324–335, Morgan Kaufmann, 2004.

[21] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *VLDB J*, vol. 15, no. 2, pp. 121–142, 2006.

[22] C. C. Aggarwal, N. Ashish, and A. P. Sheth, "The Internet of Things: A survey from the data-centric perspective," in *Managing and Mining Sensor Data*, pp. 383–428, Springer, 2013.

[23] K. Sheykh Esmaili, *Data stream processing in complex applications*. PhD thesis, ETH Zürich, 2011.

[24] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: a new architecture for high-performance stream systems," *PVLDB*, vol. 1, no. 1, pp. 274–288, 2008.

[25] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman, "Photon: fault-tolerant and scalable joining of continuous data streams," in *SIGMOD Conference*, pp. 577–588, 2013.

[26] D. Gayo-Avello, "A survey on session detection methods in query logs and a proposal for future evaluation," *Information Sciences*, vol. 179, pp. 1822–1843, May 2009.

[27] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *PVLDB*, vol. 8, no. 12, pp. 1792–1803, 2015.

[28] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, "Cayuga: A general purpose event monitoring system," in *CIDR*, pp. 412–422, www.cidrdb.org, 2007.

[29] D. Maier, M. Grossniklaus, S. Moorthy, and K. Tufte, "Capturing episodes: may the frame be with you," in *DEBS*, pp. 1–11, ACM, 2012.