Scaling Up IoT Stream Processing

Taegeon Um
Seoul National University

Gyewon Lee
Seoul National University

Sanha Lee
Seoul National University

Kyungtae Kim
Seoul National University

Byung-Gon Chun
Seoul National University

ABSTRACT

Users create large numbers of IoT stream queries with data streams generated from various IoT devices. Current stream processing systems such as Storm and Flink are unable to support such large numbers of IoT stream queries efficiently, as their execution models cause a flurry of cache misses while processing the events of the queries. To solve this problem, we present a new group-aware execution model, which processes the events of IoT stream queries in a way that exploits the locality of data and code references, to reduce cache misses and improve system performance. The group-aware execution model leverages the fact that users create the groups of queries according to their interests or location contexts and that queries in the same group can share the same data and codes. We realize the group-aware execution model on MIST—a new stream processing system tailored for processing many IoT stream queries efficiently—to scale up the number of IoT queries that can be processed in a machine. Our preliminary evaluation shows that our group-aware execution model increases the number of queries that can be processed within a single machine up to 3.18× compared to the Flink-based execution model.

ACM Reference format:
Taegeon Um, Gyewon Lee, Sanha Lee, Kyungtae Kim, and Byung-Gon Chun. 2017. Scaling Up IoT Stream Processing. In *Proceedings of APSys '17, Mumbai, India, September 2, 2017,* 7 pages.
https://doi.org/10.1145/3124680.3124746

1 INTRODUCTION

Internet of Things (IoT) devices generate various real-time data streams in diverse places, such as real-time temperature in homes [11, 15], fertility in farms [24], ball movement in stadiums [10], the number of cars in parking lots [13], and much more. From these data streams, users can create IoT stream queries—continuously processing data streams generated from their IoT devices—to obtain useful information or to control their IoT devices with low latency. These IoT stream queries handle a small amount of data streams related to users' interests. For instance, users can create queries that notify them of break-in at their home by inspecting the data streams generated from their home doors, or that adjust the fan speed of the air conditioner by analyzing the room humidity and temperature data streams.

As users have various query requests on diverse IoT devices deployed, they create many stream queries that are tailored to their concerns. If the number of users is 10 million, with each user creating 100 queries, then the number of queries becomes 1 billion. Thus, the number of IoT stream queries is huge. This workload—large numbers of small queries—is very different from the workloads that current stream processing systems target: small number of stream queries that process a large amount of data.

Current stream processing systems (SPSs) [8, 23, 27] such as Storm, Flink, and Spark Streaming do not efficiently support large numbers of IoT stream queries efficiently, because their execution models cause a flurry of cache misses for this workload. When processing stream queries, they create separate processes or threads per query. This design works well on the distributed execution of big queries. However, when running large numbers of IoT stream queries, this execution model causes frequent context switching among threads (or processes). Since each thread holds the data of each query, such as input data streams, internal query states and codes, the frequent context switching increases the working set size in a CPU, resulting in frequent cache misses, which in turn increases CPU use and hinders SPSs from processing many IoT stream queries.

To solve this problem, we take advantage of the fact that users can naturally create groups of IoT queries according to the users' intentions. For example, a user who intends to control her house can create a home group and add her home-control IoT queries to the group. Queries inside the same group can process common data streams and have common application codes (logic). For example, in the home group,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APSys '17, September 2, 2017, Mumbai, India
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5197-3/17/09…$15.00
https://doi.org/10.1145/3124680.3124746

queries that adjust the air conditioner or the heater according to the current home temperature process the same home temperature data stream. In addition, queries adjusting the temperature of different rooms inside the home can use the same code to modify the room temperatures.

In this paper, we design a new *group-aware* execution model that processes the events of queries in a way that exploits the locality of data and code references. The group-aware execution model enables a single thread to consecutively process all the events of queries within the same group. This execution model reduces CPU cache misses and improves system performance, since the data and code residing in the CPU cache will be reused. We realize the group-aware execution model on MIST—a new stream processing system aimed at supporting large numbers of IoT stream queries in a cluster of machines. The group-aware execution model scales up MIST to increase IoT queries that can be processed in a single machine; thus it helps MIST minimize the necessary number of machines to process billions of IoT queries. Our preliminary evaluation shows that the group-aware execution model improves the maximum number of stream queries that can be processed in a single machine up to 3.18×, compared to the Flink-based execution model, while maintaining the median latency below 10 ms.

2 BACKGROUND AND MOTIVATION

In this section, we describe the execution models of current stream processing systems (SPSs) by investigating two popular streaming frameworks: Storm [23] and Flink [8]. We show the limitations of their execution models in dealing with large numbers of IoT stream queries.

2.1 Directed Acyclic Graph

Modern SPSs [8, 23, 27] are designed for users to easily run big stream queries on distributed environments. They represent a stream query as a data-flow DAG (directed acyclic graph). In a DAG, a vertex (v) is either a source (s), an operator (o), or a sink (k). An edge ($v_x \rightarrow v_y$) represents the stream of data flowing from the upstream vertex (v_x) to the downstream vertex (v_y). We explain the details of the data stream and each type of vertex as follows:

- **Data Stream**: A data stream consists of continuous events, and each event is a pair of a value and a timestamp. The value holds real-world information, such as the current temperature or location, and the timestamp specifies the time of event generation.
- **Source**: A source is the root vertex of a DAG, and it fetches or receives a data stream from external systems, such as Kafka [14] or MQTT Broker [7]. It sends input events to downstream operators. This operation is I/O-bound because it receives data from the network.
- **Operator**: An operator is the intermediate vertex that has incoming and outgoing edges. Operator receives events, processes their values according to the defined operation (filter, map, windowing, aggregation, or user-defined function), and it emits the processed events as input events to downstream vertices. In general, operations, such as filters, maps, windowing, or aggregates, are CPU-bound computations.
- **Sink**: A sink is the leaf vertex of a DAG, and emits input events to external systems. It is I/O bound because it sends data through the network.

2.2 Execution Model and Limitations

We explain the execution models of two popular streaming systems—Storm and Flink—to show how they execute DAGs. After that, we investigate their limitations of processing large numbers of IoT stream queries.

2.2.1 Execution Model. Storm and Flink are good at processing a rather small number of big queries in a distributed manner, by creating separate processes (in Storm) and threads (in Flink) per DAG.

Storm [23] represents the DAG of a query as a topology consisting of two component types: spouts and bolts. A spout is mapped to a source, and a bolt is mapped to an operator or a sink. To run a topology, Storm creates one or more JVM processes, called Workers, and distributes spouts and bolts across these Workers. Each Worker maintains several threads, called Executors, and each Executor has an incoming and outgoing event queue for a component (spout or bolt). When events sent from the upstream vertex are enqueued to the incoming event queue, the Executor thread processes the events and sends them to the outgoing event queue. Spouts receive events from the external system and send them to the outgoing event queue. If there is no event to process in the incoming queue, the Executor thread sleeps until another event arrives.

Flink chains operators in a DAG and creates a DAG of Tasks, where each Task runs on the long-running runtime processes, called TaskManagers. A TaskManager can run multiple Tasks from different queries. In the TaskManager, each Task is executed by one thread, and the thread processes the events of the vertex. As with Storm, each Task thread has an incoming event queue, which sleeps when there is no event to process in the queue.

2.2.2 Limitations. Creating multiple processes and threads per query causes an excessive number of context switching among them, which increases the working set size and leads to frequent cache misses. Thus, the CPU becomes a bottleneck.

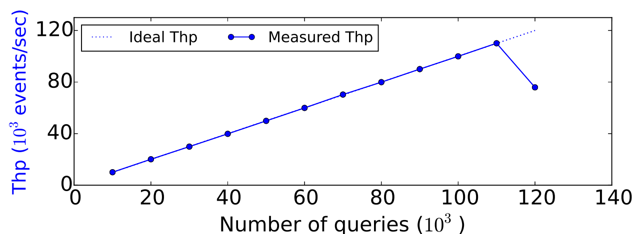

Figure 1: Performance evaluation of the Flink-based execution model

As the number of queries increases in SPSs, the number of threads or processes also increases. If the number of threads is large, the number of context switching also increases. In addition, since each IoT stream query processes a small amount of data streams, the threads sleep and wake up repeatedly, which also increases the frequency of context switching. Commonly, IoT devices generate events at a certain interval (e.g., 1 second). After a thread processes an event in a CPU, it waits until another event is generated. In the meanwhile, another thread can process the event of a different query in the same CPU. As each thread handles each query, which has its own data, such as data streams, codes and internal operator states, multiple threads that process the events of different queries in the same CPU will increase the working set size. Eventually, a large working set will not fit into a CPU cache and lead to a flurry of cache misses.

To understand this problem, we evaluate how Flink performance degrades as the number of queries (threads) increases in a single machine (a single TaskManager). Storm evidently supports a smaller number of queries than Flink does because Storm executes queries in a heavier way than Flink (processes per query vs threads per query); thus, we choose Flink as a baseline.

We implement two categories of streaming queries for the evaluation: Abnormal heart rate detection (AHR) and Point of Interest recommendation (PoI). AHR query consistently detects abnormal heart rates based on the user's activity, and PoI query recommends nearby points of interest according to the user's current GPS location. To emulate IoT data streams, we use two datasets: GeoLife [30] and PAMAP2 [21]. PAMAP2 is a dataset containing user's activity, motion, and heart rate data; and GeoLife is a collection of GPS trajectory logs from people in Beijing. We generate an event per second on average, following a Poisson process model. Each AHR and PoI query consumes a data stream generated from PAMAP2 and GeoLife, respectively. EMQ [7] is used as a message broker for the reliable large-scale delivery of these events.

We gradually increase the number of AHR and PoI queries and measure throughput (the number of processed events per second). The measured throughput should be proportional to the number of queries, since each query processes one event per second, on average. In all cases of the experiments, we use a 28-core NUMA machine (2× Intel Xeon E5-2680 2.4GHz, 35M Cache, 8× 16GB RDIMM) to run MIST. Flink TaskManager runs on the machine, and emulated sources, message brokers, and result loggers run on separate machines.

Flink was able to handle approximately 4K queries, with the main bottleneck being the cost to maintain thousands of network connections. Flink deals with each query separately, so a large number of network connections are created when users submit many queries. Maintaining each network connection is expensive; thus it degrades system performance.

To investigate the problems concerning the execution model (thread-per-query) clearly, we implement a Flink-based execution model that addresses the network bottleneck by sharing the network connections among queries while creating a new thread per operator, based on the Flink execution model. Figure 1 shows that the Flink-based execution model handles up to approximately 110K stream queries, before the measured throughput significantly degrades. This is because the Flink-based execution model has a CPU bottleneck, caused by the higher number of cache misses. We present the detailed numbers, such as the number of cache misses and CPU use compared to the MIST group-aware execution model, in § 3.5. In the following sections, we discuss how the MIST group-aware execution model reduces the number of cache misses and improves system performance.

3 GROUP-AWARE EXECUTION MODEL

In contrast to Flink, which creates new threads per query and makes the OS scheduler responsible for scheduling the event processing, we design a *group-aware* execution model that exploits the locality of data and code references while processing the events of queries in the same group. When users create groups of queries, queries in the same group can process the same data stream and have the same code. Our main idea is to group events of different queries belonging to a single group and process them consecutively in a single thread. This approach better exploits the locality of data and code references within the groups and improve the system performance, as we can reuse the same data and code residing in the CPU cache.

3.1 MIST Overview

Before illustrating the group-aware execution model, we give a brief overview of MIST, which is a new stream processing system designed to process large numbers of IoT stream queries in a cluster of machines. We realize the group-aware execution model on MIST and enable MIST to increase the number of queries that can be processed in a single machine.

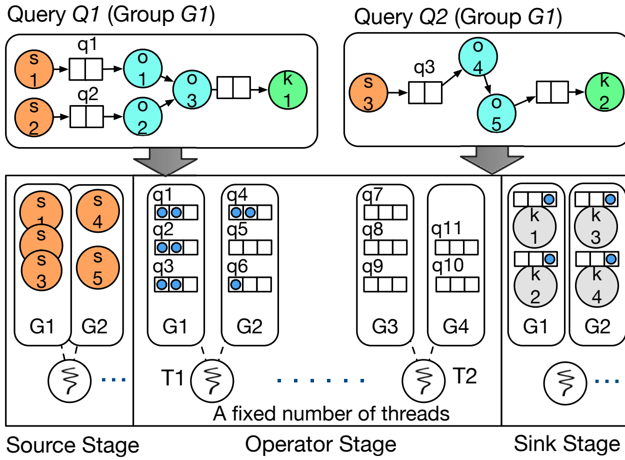

Figure 2: The architecture overview of the group-aware execution model in a stream engine. s_x, o_x, k_x, q_x, and T_x represents a source, operator, sink, event queue, and an operator thread.

Overall, MIST consists of three components: the front end, the driver, and the stream engines.

Front End. The MIST front end enables users to create and manage queries. It also provides an interface for users to group their queries by labeling them according to their purpose. Each query is converted to a DAG and submitted to the MIST driver.

Driver. When the DAG of a query is submitted from the MIST front end, the MIST driver receives the DAG and assigns it to a stream engine out of the engines available. Each stream engine is a process that runs on a single node and handles multiple stream queries. To process all of the queries in the same group by leveraging the group information, the driver assigns the queries in the same group to the same stream engine.

Stream Engines. A stream engine is a process that runs on a single node, and it is in charge of processing multiple IoT stream queries. It processes the events of queries according to the defined DAGs. We apply the group-aware execution engine in the MIST stream engine to increase the number of IoT stream queries processed in a machine. Next we present the details of the group-aware execution model.

3.2 Stage and Query Separation

Figure 2 shows the overview of the group-aware execution model in a MIST stream engine. A MIST stream engine consists of three separate stages: the source stage, operator stage, and sink stage. The threads of each stage are independent; hence, by separating the threads for I/O operations (sources and sinks) and CPU operations (operators), we can use the I/O and CPU operations effectively [26]. Each stage has a fixed number of threads (can be configured by system administrators) to reduce frequent context switching and cache misses that can occur in a large number of threads and to realize that a single thread processes the events of multiple queries in the same group.

To separate the operation of sources, operators, and sinks, MIST adds internal event queues to the DAG between a source and an operator and between an operator and a sink. In Figure 2, query $Q1$ has internal operator event queues between $s1$ and $o1$ and between $s2$ and $o2$. MIST also adds a sink event queue between $o3$ and $k1$. The source thread receives and sends events to the operator event queues, the operator thread processes the events by applying the functions of downstream operators in depth-first search order, and sends the processed events to the sink event queues; the sink thread then emits the events to external systems.

In following sections, we focus on the operator stage because processing the events of operators is the main bottleneck of the system. We explain how MIST assigns groups and queries to an operator thread and processes events in the operator stage.

3.3 Group and Query Assignment

To process the events of a group, MIST first assigns a new group to a thread, and whenever a new query for the group is created, MIST adds the operator event queue of the query to the group. As an example of the assignment of operator event queues, in Figure 2, MIST assigns group $G1$ to thread $T1$ and allocates the operator event queues of $Q1$ and $Q2$ ($q1$, $q2$, and $q3$) to $G1$, because queries $Q1$ and $Q2$ are included in group $G1$. Then, thread $T1$ will process the incoming events of all queries within group $G1$.

Our group assignment policy is to balance the number of operator event queues in the operator stage. For instance, in Figure 2, $T1$ has two assigned groups ($G1$ and $G2$) and six operator event queues; whereas $T2$ has two assigned groups ($G3$ and $G4$) and five operator event queues. When a new group and queries are created, MIST will assign the new group to $T2$ in order to balance the number of operator event queues. This assignment policy works well in a situation where the size of groups (the number of queries) does not frequently change, and the incoming event rate and required computations for each query are similar. In § 4, we will further discuss another assignment policy for other situations.

3.4 Group-Aware Event Processing

The remaining part is about how each operator thread processes events. Each operator thread has several assigned groups because the number of threads is less than the number of groups in general. At a high level, to schedule the events of assigned groups in a way that exploits the locality of

Algorithm 1: Event Processing Mechanism

```
1  Function PROCESSING(activeGroupQueue, pTimeout)
2      while not finished do
           // Sleep if the queue is empty
3          group ← activeGroupQueue.take();
4          opSchedQueue ← group.getOpSchedQueue();
5          startTime ← currentTime();
6          while group.hasEvent() and
7          elpasedTime(startTime) < pTimeout do
8              opEventQueue ← opSchedQueue.poll();
9              while opEventQueue.hasEvent() do
10                 event ← opEventQueue.poll();
11                 processEventInDFS(event);
```

data and code references, the operator thread picks an active group, which has at least one event to be processed, processes the events of the active group until there is no event to be processed, and repeats this process.

Algorithm 1 shows this event processing mechanism. A thread picks an active group from the active group queue (line: 3). For the active group queue, we use a blocking queue in order not to waste CPU cycles when there is no event to be processed. The thread sleeps if there is no active group. To wake up the thread, we create a group dispatcher thread that adds active groups to the active group queue by iterating the list of assigned groups to a thread. For example, in Figure 2, $T2$ sleeps because it has no active group. When an event is generated and enqueued to $q9$, $G3$ becomes active, and the group dispatcher will add $G3$ to the group scheduling queue of $T2$. Then, $T2$ will wake up and process the event. After selecting a group, the thread selects an active operator event queue that has at least one event (line: 8). After that, it processes all of the events from the operator event queue until it becomes empty. The operator event queue will be added to the operator scheduling queue again when an event for the operator event queue is created.

This event scheduling mechanism allows a thread to process the events of an operator successively, as well as the events of operators in the same group consecutively. As queries in the same group can have the same data stream and code, MIST can reuse recently referenced data and codes and reduce cache misses.

With a low probability, operator threads could be occupied by an active group where events are continuously generated, which would mean events in other active groups could not be processed by the threads for a long time. MIST prevents this situation by preempting the active group when the event processing time of the group is larger than the preemption timeout (`pTimeout`, line 7).

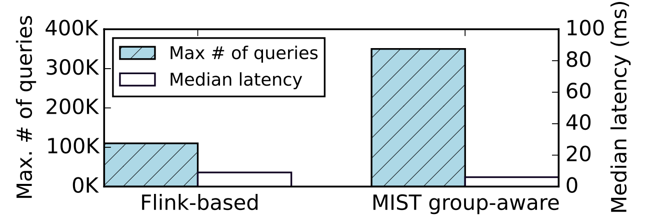

Figure 3: Performance of the Flink-based execution model and MIST group-aware execution model

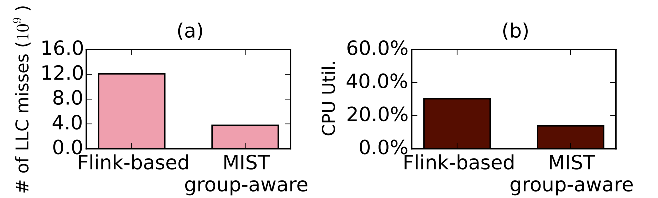

Figure 4: (a) shows the number of last-level cache (LLC) misses while processing 40K queries during 2 minutes and (b) shows the CPU use while processing 40k queries during 2 minutes in the Flink-based execution model and MIST group-aware execution model

3.5 Preliminary Evaluation

We evaluate the performance of the MIST group-aware execution model in the same environment used for Flink evaluation § 2.2.2. We set the number of threads in the source, operator, and sink stages to 100, 56 (2× the number of cores), and 100, respectively. To emulate query groups, we group 100 AHR and PoI queries, i.e., the number of groups was $\frac{n}{100}$, where n is the number of queries.

Figure 3 shows that the MIST group-aware execution model improves the number of IoT stream queries processed in a machine up to 350K with 6ms median latency; 3.18× larger number of queries compared to the Flink-based execution model. These results demonstrate that our group-aware execution model reduces cache misses and improves system performance. Figure 4(a) shows that the Flink-based execution model has a 3.19× higher number of last-level cache (LLC) misses compared to MIST in processing 40K queries in 2 minutes. The higher number of cache misses leads to inefficient CPU use in the Flink-based execution model, which is illustrated in Figure 4(b). At the same number of queries, the CPU use of the Flink-based execution model is 30.2%, whereas that of the MIST group-aware execution model is 13.8%.

4 DISCUSSION

We discuss interesting research directions below to more scale up IoT stream processing in MIST.

Reducing Duplicate Computations. In plenty of IoT stream queries, some queries can generate duplicate computations and this could be a cause of high CPU use. MIST can reduce these duplicate operations by merging queries that have the same computations. Sharing operations among multiple queries has been explored in the streaming database field [5, 9, 17, 20, 25, 28, 29], focusing on optimizing computations that have different parameters, such as different window sizes or different filter predicates. MIST can also use these techniques to merge queries. However, with billions of stream queries, finding the queries to merge could be time-consuming. To solve this challenge, we can also leverage the group information to identify mergeable queries. By limiting the search space within the same group, we can identify them quickly with negligible overhead.

Load Balancing in Group Assignment. Current group assignment policy supposes that an operator thread load is proportional to the number of queries. However, load imbalance can occur if the load is not proportional to the number of queries: the incoming event rate and amount of computation of each query are different. The load imbalance can increase the latency and degrade the system performance; thus, group assignment should consider the load of threads, which depends on the event rate, amount of computation, etc. As an example, queuing theory can be used to measure the load of threads [6]. Then, assigning new groups to the thread that has the lowest load balances the load among threads.

Group Reassignment. Even when we balance the load of threads while assigning groups, load imbalance could still occur over time as we pin a group to a certain thread. For instance, the number and the event incoming rate of queries in a group can change over time after the group is assigned to a thread. A high-level policy to mitigate the load imbalance among threads is to reassign the groups from overloaded threads to underloaded threads. To develop the reassignment policy, we should consider several details, including how to determine overloaded and underloaded threads, and which groups should be reassigned. These issues must be addressed while minimizing the number of groups to be reassigned, because processing events of groups in different threads is likely to increase the number of cache misses.

Reducing Memory Use. Some operators have internal states, such as windows or aggregates. When the size of persisting states for each query becomes large, memory can become a bottleneck. Unloading (removing) the states of inactive queries from memory to disk is a feasible solution to solve the memory bottleneck, because some IoT queries could become inactive for a long time. For instance, GPS queries that track bicycle location are only active when users are riding bicycles. However, with large numbers of queries, deciding which queries are inactive is challenging. We can again address this challenge using the group information. Since queries in the same group can share data streams, queries that process the same data streams are likely to become inactive at the same time, when the data streams become inactive. Hence, MIST could track whether a group is active or inactive and (un)load the whole queries in the same group. This approach can reduce the overhead of tracking each individual query status.

5 RELATED WORK

Stream Processing Systems. The limitations of current stream processing systems [8, 23, 27] are discussed in § 2.2.2.

Streaming Databases. Streaming databases [1–5] are designed to process data streams. They focus on streaming queries that process data streams in which the data format is relational data (e.g., uniform schema). Their target is different from MIST, which does not limit the data format generated from various IoT devices. In addition, they do not leverage the group information and exploit the locality of data and code references to scale up the number of queries processed in a machine.

Sensor Networks. Sensor networks aggregate data streams from geographically distributed sensors. Most sensor network communities focus on data aggregation in the network to reduce the communication overhead [16, 18]. In contrast to them, our group-aware execution model focuses on centralized processing of IoT stream queries in the back-end server.

IoT Platforms. There are commercial solutions providing an integrated stack for developing IoT platforms, such as Azure IoT Suite [19], AWS Internet of Things [22], or IFTTT [12]. As these platforms need to serve more and more stream queries, the MIST group-aware execution model can be used to scale up stream processing in these systems.

6 CONCLUSION

We propose MIST, a new stream processing system that is designed to efficiently handle large numbers of IoT stream queries. To scale up the number of queries that can be processed in a machine in MIST, we design a new group-aware execution model that exploits the locality of data and code references in the same group. Our preliminary evaluation shows that the MIST group-aware execution model improves the number of IoT stream queries up to 3.18× compared to the Flink-based execution model. We believe that MIST offers new, interesting opportunities for research to improve IoT stream processing.

ACKNOWLEDGEMENTS

We thank anonymous reviewers for their comments. We also thank Brian Cho, Zhenping Qian, Jooyeon Kim, Wonwook Song, Hyunmin Ha, and Jangho Seo for their feedback. This research was supported by Samsung Research Funding Center of Samsung Electronics under Project Number 0421-20150094.

REFERENCES

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The Design of the Borealis Stream Processing Engine. In *CIDR*.

[2] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *VLDB* 12, 2 (2003), 120–139.

[3] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. 2003. STREAM: the stanford stream data manager (demonstration description). In *ACM SIGMOD*.

[4] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. 2003. TelegraphCQ: continuous dataflow processing. In *ACM SIGMOD*.

[5] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD*.

[6] Robert Gallager Dimitri Bertsekas. 1992. *Data Networks* (2nd ed.). Prentice Hall.

[7] EMQ Enterprise. 2017. EMQ - Erlang MQTT Broker. http://emqtt.io/docs/v2/index.html. (2017).

[8] Apache Flink. 2017. Apache Flink: Scalable Stream and Batch Data Processing. https://flink.apache.org. (2017).

[9] Lukasz Golab, Kumar Gaurav Bijay, and M Tamer Özsu. 2006. Multi-query optimization of sliding window aggregates by schedule synchronization. In *CIKM*.

[10] Mahanth Gowda, Ashutosh Dhekne, Sheng Shen, Romit Roy Choudhury, Lei Yang, Suresh Golwalkar, and Alexander Essanian. 2017. Bringing IoT to Sports Analytics. In *NSDI*.

[11] Trinabh Gupta, Rayman Preet Singh, Amar Phanishayee, Jaeyeon Jung, and Ratul Mahajan. 2014. Bolt: Data Management for Connected Homes.. In *NSDI*.

[12] IFTTT. 2017. IFTTT. https://ifttt.com/about. (2017).

[13] A. Khanna and R. Anand. 2016. IoT based smart parking system. In *IOTA*.

[14] J. Kreps, N. Narkhede, and J. Rao. 2011. Kafka: A distributed messaging system for log processing. In *NetDB*.

[15] Nest Labs. 2017. Nest. https://nest.com/. (2017).

[16] Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2002. TAG: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*.

[17] Samuel Madden, Mehul Shah, Joseph M Hellerstein, and Vijayshankar Raman. 2002. Continuously adaptive continuous queries over streams. In *ACM SIGMOD*.

[18] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM TODS* 30, 1 (2005), 122–173.

[19] Microsoft. 2017. Azure IoT Suite. https://www.microsoft.com/en-us/cloud-platform/internet-of-things-azure-iot-suite. (2017).

[20] KVM Naidu, Rajeev Rastogi, Scott Satkin, and Anand Srinivasan. 2011. Memory-constrained aggregate computation over data streams. In *ICDE*.

[21] Attila Reiss and Didier Stricker. 2012. Creating and benchmarking a new dataset for physical activity monitoring. In *ACM PETRA*.

[22] Amazon Web Services. 2017. AWS Internet of Things. https://aws.amazon.com/iot/?nc1=h_ls. (2017).

[23] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *ACM SIGMOD*.

[24] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. 2017. FarmBeats: An IoT Platform for Data-Driven Agriculture. In *NSDI*.

[25] Song Wang, Elke Rundensteiner, Samrat Ganguly, and Sudeept Bhatnagar. 2006. State-slice: New paradigm of multi-query optimization of window-based stream queries. In *VLDB*.

[26] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: an architecture for well-conditioned, scalable internet services. In *SIGOPS*.

[27] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*.

[28] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. 2005. Multiple aggregations over data streams. In *ACM SIGMOD*.

[29] Rui Zhang, Nick Koudas, Beng Chin Ooi, Divesh Srivastava, and Pu Zhou. 2010. Streaming multiple aggregations using phantoms. *VLDB* 19, 4 (2010), 557–583.

[30] Yu Zheng, Xing Xie, and Wei-Ying Ma. 2010. GeoLife: A Collaborative Social Networking Service among User, Location and Trajectory. *IEEE Data Eng. Bull.* 33, 2 (2010), 32–39.