Multi-representation based Data Processing Architecture for IoT Applications

Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, Amr El Abbadi
Department of Computer Science, University of California, Santa Barbara
{vaibhavarora, nawab, agrawal, amr}@cs.ucsb.edu

Abstract—Internet of Things (IoT) applications like smart cars, smart cities and wearables are becoming widespread and are the future of the Internet. One of the major challenges for IoT applications is efficiently processing, storing and analyzing the continuous stream of incoming data from a large number of connected sensors. We propose a multi-representation based data processing architecture for IoT applications. The data is stored in multiple representations, like rows, columns, graphs which provides support for diverse application demands. A unifying update mechanism based on deterministic scheduling is used to update the data representations, which completely removes the need for data transfer pipelines like ETL (Extract, Transform and Load). The combination of multiple representations, and the deterministic update mechanism, provides the ability to support real-time analytics and caters to IoT applications by minimizing the latency of operations like computing pre-defined aggregates.

I. INTRODUCTION

The number of internet connected physical devices, also known as Internet of things (IoT), have grown rapidly over the last few years, and are predicted to grow even faster. Gartner has predicted [6] that there would be 26 billion IoT devices by the end of 2020. Smart cars, smart cities, weather monitors, smart farms and fitness and health tracking wearable devices are some examples of internet connected devices, which continuously transmit data points from connected sensors. IoT applications continuously collect and process the data received from these devices. Some of IoT applications might receive data from a diverse set of sensors. For example, an app might collect both current traffic and weather data to route smart cars. One of the major challenges for IoT applications is to efficiently store and process this data.

Data management systems supporting IoT applications pose many challenges. IoT applications need to support both high frequency data ingestion as well as real-time analytics. Monitoring the weather in disaster prone areas, routing smart cars, tracking health data like heart rate are some scenarios were insights into the data are needed in real-time. In all the examples mentioned, real-time analytics must be considered while ingesting a high incoming rate of data.

The data ingestion and the analytical processing demands of IoT applications are also diverse in nature. Analytical queries might be online queries (aggregates over a certain attribute), graph-processing requests (calculating activity in a connected community of a user in a motion tracking app) or offline batch processing queries (models for predicting the long-term likelihood of droughts, cyclones etc, executing over months/years worth of weather data). Data ingestion might be in the form of continuous independent events or data values from multiple related sensors which should be ingested atomically. The data management architecture used for supporting IoT applications should be able to efficiently handle this heterogeneity of data processing demands.

One of the important challenges for modern data-intensive applications has been to tackle the Variety of big data. Online Transaction Processing (OLTP), Online Analytical Processing (OLAP), graph processing and stream processing are some examples of diverse data processing requirements of applications in the big data era. The variety of data processing needs of the applications have led to the fall of the “one size fits all paradigm” [26]. Applications now deploy a multitude of data processing engines, rather than just using a single relational or a NoSQL datastore. OLTP engines like MySQL [8], key-value stores like Cassandra [1], stream processing engines like Storm [4], column store like Vertica [11] and graph stores such as Neo4j [9] are some examples of such varied engines. This can be referred to as multi-engine architecture.

The multi-engine architecture lends itself well to IoT applications because of the variety of data processing needs of these applications. The multi-engine architecture benefits from the different characteristics of diverse engines. Transactional processing can be performed at OLTP engines like MySQL and VoltDB; continuous events can be processed using stream processing engines like Storm; analytics can be performed on engines optimized for analytics, like Vertica and Druid; and graph processing can be performed using data processing engines like Neo4j and Giraph. Additionally, executing different workloads on the different data processing engines helps in improving performance by isolating workloads. However, the multi-engine architecture has some drawbacks. Continuous ETL (Extract, transform and load) or CDC (Change Data Capture) pipelines [16], [2] need to be used for transferring data between these systems. These data pipelines result in a lag between the time when data is ingested and when it is reflected in the analytical query results. This time lag is a bottleneck to achieving real-time analytics. Furthermore, micro-ETL (ETL pipelines scheduled at highly frequent intervals) and CDC data pipelines need to be continuously maintained and techniques need to be developed to handle failures. Additionally, data transfer pipelines also consume a high network bandwidth.
while continuously transferring the data. IoT applications will further exacerbate the shortcomings of this architecture. IoT applications have to support write-heavy workloads, since IoT devices send the readings from connected sensors at a high frequency. A high frequency of data ingestion would lead to a higher load on the data transfer pipelines to support real-time analytics.

Another important characteristic of IoT applications is the need for periodically calculating pre-defined aggregate values. For example, values from a heart-rate or blood sugar-level monitor, might be used to calculate health indexes over a pre-defined past interval and generate alerts if needed. Traditionally data management systems use materialized views to store such pre-defined aggregate values. The views can either be updated asynchronously or synchronously with the update transactions. If the view is updated asynchronously, the view computation has to be delayed for the update to be applied to the base data to decide on the order of updates. If the view is updated synchronously, the footprint of the update transaction increases, which adversely impacts both system throughput and latency. However, the data management architecture for handling IoT applications should be able to efficiently support requests accessing pre-defined aggregate values.

To support IoT applications, we need an architecture which can still benefit from the advantages of using different data processing engines, while removing the bottleneck of continuous data transfer, and providing support for specialized IoT analytics like pre-defined aggregates. Different data processing engines use different representations of data to optimize for specific use-cases. For example, relational databases like MySQL use row-based representations, analytical engines like Vertica store data in column representations, whereas graph databases like Neo4j store networked data in graph representations.

We propose a multi-representation based data processing architecture, where copies of the data are stored in multiple representations. Storing data in different representations will help perform the various update and analytics operations on the representations most suited for them. All representations of the data are updated in a unified manner and hence, completely remove the need to maintain micro-ETL and CDC data pipelines. The different representations can be configured with different schemas to aid in specialized analytical operations like pre-defined aggregates, employed by IoT applications.

One of the reasons for using micro-ETL and CDC data transfer pipelines is that most systems providing transactional or single-item update support are non-deterministic in nature. Hence, the insert, update and delete requests are first sent to an update processing engine, where the operations are processed in a particular order. The applied updates are then periodically transferred and ingested at other datastores, which support read-only analytical queries. This ensures that the order of updates is the same at all the datastores, and hence analytical queries would return valid results. Non-determinism aids in achieving more concurrency as operations can be processed in any order as long as the transactional or update semantics (serializability, snapshot isolation, atomic updates etc) offered by the given data management system are satisfied. But this leads to the bottleneck of using data transfer pipelines. One possible solution is to use synchronous replication techniques to update the data residing in multi engines. However, this will adversely affect system throughout, as some of the engines might not be optimized for updates. To overcome these bottlenecks, we employ a deterministic ordering scheme to update the different data representations. The order of updates will be pre-decided by a sequencer layer and the requests and their corresponding processing order will be then sent to all the data representations. Using a deterministic ordering scheme, we can completely eschew micro-ETL and CDC pipelines. As the update order is pre-defined, each representation can update at its own rate. This allows the decoupling of updates at multiple representations, while removing the need for data transfers between the representations.

A deterministic ordering scheme is suited to IoT applications due to the nature of the data. Many IoT applications either do not need any ordering guarantees, or the ordering is naturally enforced by the timestamp of the values. However, some IoT applications, might receive data values (corresponding to a physical world state) from multiple sensors, and resolve the final state by a last-writer like approach. In such cases, there is a need for all the representations of the data to have the same final state. The deterministic approach will guarantee that there is one global order enforced at all representations and help in optimizing for latency by removing the data transfer pipelines.

Deterministic ordering has been previously proposed in partitioned \[28\] and main memory databases \[19\], \[29\], \[27\] to increase performance by reducing aborts using a pre-determined transaction order. We adapt the deterministic scheme to the multi-representation architecture, and use the deterministic order to update the different representations of data, thus completely removing the need of data transfer pipelines.

To support pre-defined aggregates, different representations may support different schemas. One of the representations can store the entire copy of the data, whereas another representation can be used to store schemas with attributes having aggregate values of base data. Because of the properties of IoT data, and due to the deterministic update mechanism providing a pre-determined update order, the aggregate computation does not have to wait for updates to be applied to base data and can be updated separately from the base data. The decoupling of the aggregate computation from the base data provides the ability to reduce the latency of computation of pre-defined aggregates.

The rest of the paper is organized as follows. We discuss the system architecture in Section II and the deterministic update mechanism in Section III. Some application scenarios are presented in Section IV and then we discuss the deployment of the data processing architecture in Section V. Section VI discusses the related work and Section VII concludes the paper.
II. ARCHITECTURAL OVERVIEW

The system architecture is illustrated in Figure 1. The data processing architecture comprises of two main components: the Execution Engine (EE) and the Data Storage Layer (DSL). Values from IoT devices are aggregated by sensor aggregators and sent to the application. The IoT application client sends the data management requests (updating the collected data values as well as client read / analytical requests) to the execution engine (EE), which processes the requests and sends them to the data storage layer (DSL). The EE is also responsible for collecting the results of the reads and returning them to the client.

The IoT application client operations are classified into two categories: write transactions and read-only queries. Providing a transaction interface gives the IoT applications the capability to express dependencies, such as values from multiple sensors which should be ingested atomically. Independent events can be expressed as transactions comprising a single operation. A characteristic of IoT applications is that the incoming values from the sensors represent new values and do not depend on existing values as in traditional databases. To model this aspect, each write transaction is comprised of blind writes. EE is responsible for ordering the write transactions. EE sends the deterministic order and the corresponding write transactions to the data storage layer.

A read-only query is specified to be executed at a particular representation. This information is included in the query sent to the EE by the IoT application client. The read-only query is expressed in the query language supported by the representation.

The data storage layer (DSL) stores the data in multiple representations, like row, column, graph and streams. Each representation receives the update transactions as well as read-only queries from the EE. Each representation is responsible for applying the update transactions in the order specified by the EE. As the update order is pre-defined, each representation can update at their own rate.

Each representation can be integrated with a different existing data storage engines. For example, MySQL can be used as one of the row representation and H-Store can be used as another row representation. Neo4j can be used for graph-based representation.

Each representation can also have a different schema. This characteristic is essential for taking advantage of different representations for efficiently calculating pre-defined aggregates. One representation can store the entire base data, while another representation might store attributes which are aggregates of base data values. As write transactions contain blind writes, pre-defined aggregates and incremental view computation can be computed directly on the aggregated data. The combination of different representations with different schemas and a deterministic update mechanism provides an efficient mechanism to calculate pre-defined aggregates with low latency.

III. DETERMINISTIC UPDATE MECHANISM

Deterministic scheduling is employed to update the data representations. EE receives the transactional write requests from application clients and pre-determines the order of transaction execution. Each representation then applies the transaction writes in the order-specified by the EE. A limitation for the deterministic mechanism is that entire read and write set of a transaction has to be known in advance. However, this aspect does not affect IoT applications, since write transactions are blind writes and the entire transaction (values to write) is known in advance. The deterministic update mechanism comprises 2 stages: sequencing at the EE and applying the ordered updates at each representation.

A. Deterministic Ordering at the EE

Write transactions are sent to the EE. EE comprises a sequencer component, which determines the order of execution of the write transactions. Each write transaction is assigned a monotonically increasing logical timestamp. The timestamp order assigned by the sequencer is the transaction execution order. The pre-determined execution order is then sent to each representation in the DSL. The sequencer uses a batching mechanism to efficiently process the received write transactions, rather than sending each transaction individually to the representations. In particular, the sequencer batches the received transactions, orders them and then sends the batch to each of the representations.

To ensure that the sequencer is not a single point of failure and performance bottleneck, multiple machines are combined to create a highly available sequencer component. As in [28], each machine shares the sequencing in a round-robin fashion, and is responsible for sequencing in the defined epoch period. Using multiple machines for the sequencer helps in the efficiently handling a large volume of write requests.

B. Applying the pre-determined order

Each representation receives a batch of update transactions from the EE. Each representation is responsible for applying the update transactions in the order specified by the EE. The sequencer sends batches of transactions to each representation. The size of the transaction batch send by the EE is configured.
for the representations, which are optimized for updates. Some of the data representations are optimized for analytics, and might be overwhelmed by the continuous batches of write transactions sent to them. To overcome this bottleneck, each representation has an additional level of batching to make the writes to the representation more efficient. The batch size at each representation is separately configured, based on the characteristics of the representation.

A challenging aspect of the deterministic scheduling is how to execute the sequence of transactions in the given order efficiently. A naive technique is to use a single thread to process the transaction batch. But this technique limits the throughput achievable by the scheme on modern multi-core processors. Existing deterministic schemes use different mechanisms to apply the deterministic order in a multi-core setting. H-store [27] partitions the data among the cores and each core executes the transactions accessing the data under its control, in a single threaded execution. However, the performance is highly dependent on partitioning scheme, as any transaction accessing data across its partition, will lead to stalling the execution to ensure the pre-determined order. Calvin [28] uses a locking mechanism and a locking thread guarantees that access to the data is provided according to the pre-determined transaction order. Bohm [19] is designed for a main memory multi-version environment and divides the concurrency control and execution steps. The concurrency control threads divide the data among them and pre-process the transactions accessing the data under their control and create place holder versions for the transactions to access. The execution threads also divide the data among them and execute the transactions, blocking when the place holder versions needed have not been created yet. LADS [29] also pre-processes the transactions and examines the transactions for their dependencies, and creates a dependency graph, which captures the dependencies between all the operations (within a transaction and across transactions). It then divides the dependency graph into different sub-graphs while minimizing the edge-cuts among the sub-dependency graphs. The sub-graphs are then executed by different worker threads.

The technique employed in the proposed multi-representation architecture has some similarities to the one employed in LADS, but catered to workloads supported by the IoT applications. The update component of each representation comprises a pre-processing thread and multiple executor threads. To efficiently execute the transactions, the deterministic schedule is applied in a multi-threaded environment in two phases. In the first phase, known as the pre-process phase, a transaction batch with a pre-determined schedule is pre-processed to construct the dependency graph of transactions within the batch. The dependency graph expresses the ordering dependencies between the transactions present in a given transaction batch. The dependency graph is constructed by the pre-processing thread at the granularity of the transaction batch. Transactions are the nodes in a dependency graph. If any two transactions access a common data item, then they are termed as dependent and the graph has a directed edge from the transaction earlier in the schedule to the one later in the schedule. Each edge is also labelled with the data item(s) causing the dependency. The dependency graph is then uniformly partitioned (each partition gets the same number of vertices) into components known as batch-partitions. Each batch-partition is then assigned to a different executor thread. The number of batch-partitions can be configured. Since each partition is assigned to a different thread, a possible configuration is for the number of partitions to be set to the number of cores in the machine.

In the second phase, known as the execution phase, multiple executor threads are invoked to execute the different batch partitions. Before executing a transaction in the assigned batch partition, the executing thread ensures that all the transactions corresponding to the incoming edges of the transaction have been executed. Each thread executes the batch partition assigned to it. After all the executor threads finish processing, the next transaction batch can be processed.

The proposed technique is catered towards low conflict workloads like write transactions among IoT applications. As opposed to LADS [29], we do not partition the dependency graph for minimum edge-cuts. This would lead to stalls in a high conflict setting, but in a low conflict setting, it reduces the time taken to partition the graph. As we partition the graph dynamically for each batch, the proposed technique would better utilize the CPU, as compared to static partitioning of the data among the threads, employed in H-Store [27].

Representations supporting long running read queries, employ multi-versioning to update the data in parallel. Ingestion of each transaction batch leads to the creation of a new version. The entire transaction batch will be made atomically visible with the creation of a new version. As each transaction comprises blind writes and the transaction batch is made visible atomically, the execution phase for applying the deterministic order at such representations is modified to only ingest the last value corresponding to a data item. For executing a transaction in the assigned batch partition, the executor thread does not have to wait for incoming dependencies of a transaction to execute. During the execution of a transaction present in a batch partition, the executor thread first inspects the outgoing edges of the transaction. If there is an outgoing edge corresponding to write of a data item, then that particular write of the data item is ignored. This ensures that only the last write corresponding to each data item is executed. As the entire transaction batch is made atomically visible, the values after the entire execution of the batch will be consistent.

IV. APPLICATION SCENARIOS

We now discuss some application scenarios for IoT applications to illustrate the benefits of the system architecture presented.

Weather Monitoring: Consider a weather monitoring application which receives data values corresponding to air pressure, temperature, humidity and wind speed from a diverse set of sensors. The application can be used by clients to execute online queries co-relating different data values, run prediction
models and to view the current temperature displayed on the application dashboard.

The proposed system architecture is well suited to such an application. One representation can be a row-based representation backed by a relational engine like MySQL. This representation will store the entire data (with timestamp as the primary key) and will allow clients to write queries co-relating different weather statistics like wind, humidity, temperature etc. The second representation will be used to maintain a pre-defined aggregate like average temperature in a timestamp range (for example a range can be equivalent to 10 readings in base data). The second representation can be updated directly without waiting for the updates to be applied to the first representation and then calculating the average temperature over a range. The average temperature over the most recent range will then be used to update the application dashboard. Since, the second representation needs a simpler data model, we can employ a main memory store like Redis. A third representation can be used to store the entire data and will be used to execute prediction models. Since, this representation is used to execute offline jobs, it can be backed by a data processing engine like Spark.

**Smart City:** Consider a future city with self driven cars. These cars are routed based on the weather and traffic conditions, which are monitored using multiple sensors placed between each set of intersections. There are multiple sensor devices on each such road stretch, each sending traffic information and the current weather condition, along with other statistics. The values are sent to the data management system for storage and processing.

The multi-representation architecture can be used for data management of such an application. One of the representation can be a row-based representation (a key-value store like Cassandra can be used) which keeps the current traffic and weather reading for each stretch of the road. This information is used by the application to route traffic. Since, there are application components which query the data and use it to route the traffic, the weather and traffic information is atomically ingested to make sure every decision is made based on a consistent state. A second representation can be used to store the average traffic values in a time-range. Since, the second representation needs a simpler data model, we can use a main-memory store like Redis to store the aggregate. A third representation stores the traffic information over last month and provides the ability to run queries co-relating values at different times (a relational columnar engine can be used). Since, the different sensors send information related to the same stretch of the road, a last writer like approach can be used to resolve conflicting values. The deterministic ordering mechanism ensures that all representations write the data in the same order, so every representation will return consistent results. The transactional engine allows the weather and traffic information to be ingested atomically, and using the different representations allows the data management system to handle diverse analytics and ingestion demands.

**V. Deployment**

An important component for supporting low-latency operations for IoT applications is deployment. Modern-day data intensive applications are deployed in datacenters around the world to support scale and low-latency for accesses from around the global. However, some of IoT applications, need very low latency support. To enable such operations, the edge computing model has been predicted to play a vital role [14], [5]. In the edge computing model, data is stored in cloudlets or edge datacenters, which are closer to the end user, in addition to storing the same data in the cloud.

The proposed multi-representation architecture is suited to be deployed in such an environment. The data can be partitioned, with each partition corresponding to a spatial region of the IoT application. A representation of the data corresponding to a data partition, will be stored in a edge-datacenter. Any representation supporting operations requiring real-time response (like some pre-defined aggregates), will be deployed in the edge as well. The representations supporting queries which have lower latency latency requirements or have to support offline jobs, for data from all the partitions, will be deployed in the cloud. This way any queries which require a holistic view of the data across regions can be supported at the cloud datacenters. The deterministic update mechanism allows the write operations to be ordered and then apply the incoming data in the same order at both the edge datacenters and the cloud datacenters. This removes any extra synchronization between the edge datacenters and cloud datacenters, and also supports low-latency because we do not have to wait for the data to be applied to all the representations in the edge, before sending the data to the cloud datacenters.

**VI. Related Work**

Several academic and industrial solutions have been proposed in the past to tackle heterogeneous data processing demands, some of which are akin to current IoT application needs. We compare and contrast the proposed multi-representation data processing architecture with these approaches below.

Relational databases have been around for the last four decades [13] and were the default data management solution for many years. Data warehousing techniques [15] were then introduced to support analytical processing. One of the major reason behind using data warehousing techniques was to separate performance characteristics of transactions and analytics. This classification into two major systems was used to satisfy major data management needs for a long time. A plethora of systems have emerged in the last decade to support disparate data management demands. Key-Value stores [1], stream processing engines [4], graph databases [9], column stores [11], [25] and batch processing engines [3], [17] have been developed for supporting variety in data processing. However, none of these systems individually meet the variety of demands of IoT applications.

Federated database engines like Multibase [24] and BigDAWG [18] integrate heterogeneous databases and provide
a query processing layer over them to seamlessly query over multiple sources. However, we propose a more tightly coupled approach by integrating the representations of the data and updating of such representations, instead of focusing on the query processing mechanism.

Main memory database approaches have been proposed to satisfy both OLTP and OLAP demands. SAP Hana [21] employs a main memory columnar design approach to support both OLTP and OLAP workloads. Hyper [20] is a system which handles both transactional and analytical workloads in either a column or a row oriented main memory database, using a copy-on-write technique. The multi-representation architecture, on the other hand, is a solution for diverse operations such as stream processing, graph-based operations, pre-defined aggregates, online queries, and provides the ability to separate the processing of such diverse operations, while removing the need for transferring the data between the systems.

Snappy Data [23] integrates a distributed in-memory data-store (GemFire) engine with Apache Spark’s runtime and provides support for OLTP, OLAP and streaming. Data can be stored in either row or columnar form. OLTP operations are supported using the in-memory engine and OLAP operations are supported using Spark’s executors. Such an architecture removes the overhead of maintaining multiple data representations. However, it cannot provide separation of transactions and analytics on different servers or different representations. Furthermore, since such an architecture uses a single data representation, it has to choose one representations for both transactional and analytical workloads, which will result in sub-optimal performance for diverse operations of IoT application.

There have been several past attempts at using more than one representations for specific needs [10], [12], [22]. Most of the systems target hybrid OLTP-OLAP workloads, and either need a data transfer mechanism or synchronously update the data.

The recently proposed Lambda Architecture[7] stores data in two layers, a batching layer (like HDFS) optimized for batch processing and a speed layer, like Apache Storm [4], which processes data streams. Each ad-hoc query is sent to both representations, and the results are integrated and sent back to the client. However, such an architecture, only deals with simple events and does not support updating multiple values atomically, and does not provide any ordering mechanism. Furthermore, it deals with stream and batch processing operations, which are only a subset of diverse requirements of IoT applications.

VII. CONCLUSION

We propose multi-representation based data processing for supporting IoT applications. Data is stored in multiple representations, and a deterministic ordering mechanism is used to update the data. The proposed data processing architecture allows the ability to perform real-time analytics, tackles diverse requirements of IoT applications, completely removes data transfer pipelines, and reduces the latency of computation of pre-defined aggregates.

VIII. ACKNOWLEDGMENTS

This work is partially funded by a gift grant from Oracle.

REFERENCES