CHAPTER

5 BIG DATA MANAGEMENT ON WIRELESS SENSOR NETWORKS

Chih-Chieh Hung*, Chu-Cheng Hsieh†

Tamkng University, New Taipei City, Taiwan * *Slice Technologies Inc., San Mateo, CA, United States*†

ACRONYMS

GFS	Google file systems
HDFS	Hadoop distributed file system
RDD	resilient distributed dataset
SQL	structured query language
WSN	wireless sensor network

1 INTRODUCTION

Recent development of various areas of information and communication technologies has contributed to an explosive growth in the volume of data. These data can be analyzed for insights that lead to better decisions and strategic business moves. Amazon Inc. shows us an example that uses data insights to improve business intelligence. The company gained a patent for what it calls anticipatory shipping, a method to start delivering packages even before customers click "buy." This method takes previous orders, product searches, wish lists, shopping-cart contents, returns, and even how long an Internet user's cursor hovers over an item. Analyzing such large-scale, rapid-generated, and various types of data helps Amazon Inc. shorten the delivery time from their hubs to customers, thereby earning great customer satisfactory.

The data sets used in the above example are so-called big data. This term refers to data sets that so large or complex that traditional data processing applications are inadequate. Big data usually has three V characteristics: volume (the quantity of generated and stored data may not be easily handled by conventional databases), velocity (the speed at which the data is generated and processed to meet the demands and challenges that lie in the path of growth and development), and variety (data generated are from multiple sources and in multiple formats). These characteristics of big data bring not only huge opportunities but also huge challenges, including analysis, capture, data curation, search, sharing, storage, transfer, visualization, querying, updating, and so forth.

Thanks to advanced development of big data techniques, these techniques include file systems for big data [e.g., Hadoop distributed file system (HDFS)], noSQL databases (e.g., HBase), data processing models for big data (e.g., MapReduce), streaming techniques for big data (e.g., Storm), query engines (e.g., Impala), big data architecture (e.g., lambda architecture), and so forth. More details will be given in Section 3. Powered by these techniques, including tools and architectures, it is possible for people to collect, store, and analyze big data from diverse sources efficiently. Therefore, people are willing to develop/deploy novel methodologies/devices for collecting data. A wireless sensor network (WSN) is one of the important sources, which can collect data in several kinds of environments easily.

A WSN is composed of small, low-cost, and self-organized sensor nodes. Fig. 1 gives an illustrative example of a WSN. Sensor nodes are able to sense the readings from the environment they deployed, and to communicate with each other within their communication rages in an ad-hoc manner. The readings collected by sensor nodes will relay to "sinks" which can send data outside sensor networks and receive commands from clients and applications. Once the sensor nodes are deployed, sensor nodes could send the sensing readings from the environments to sinks periodically. These characteristics make WSNs a promising solution to collect data in variety of fields, even in areas that people cannot reach easily—for example, in forest areas, battlefields, and volcano areas.

The major challenging issue on the data management in WSNs is energy preservation. Energy preservation means that every computation should reduce energy consumption as much as possible. The energy of sensor nodes is easily exhausted since the sensor nodes are powered by batteries. Once the energy of sensor nodes is exhausted, the whole network is likely to be partitioned into disjoint subnetworks so that some readings cannot be sent to any sink. Decentralization is one of the promising ways to achieve energy preservation, which refers to distributing computation tasks into sensor nodes

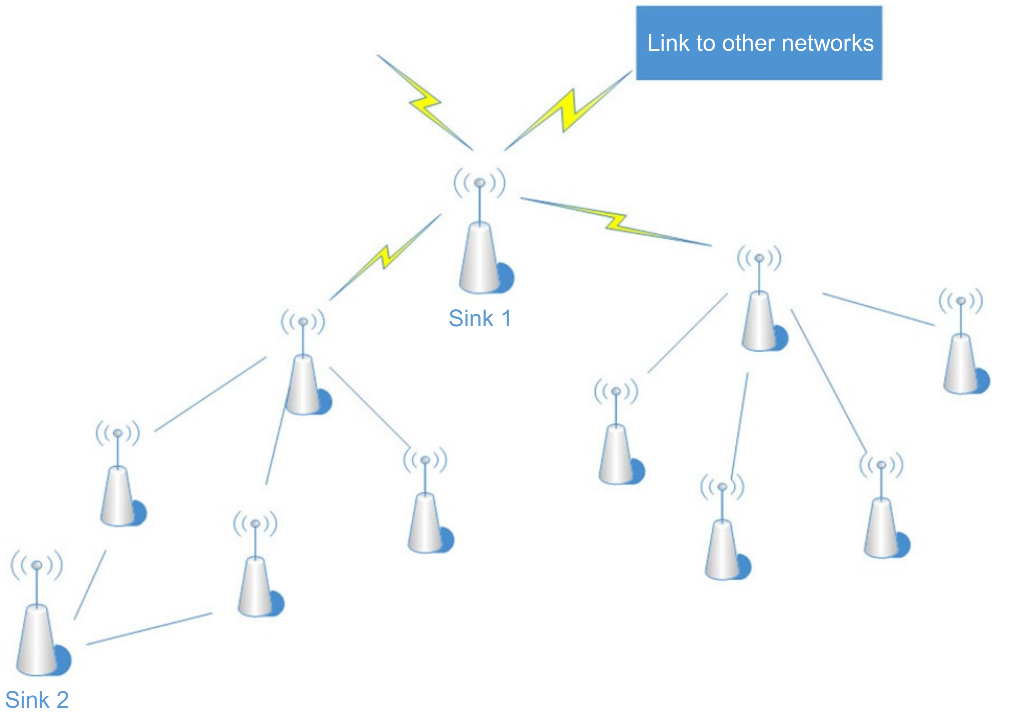

FIG. 1

An illustrative example of wireless sensor networks.

rather than executing them in a centralized way. In a large-scale sensor network, sensor nodes usually relay messages for the others so that the batteries of sensor nodes are easily drained. Communicating with nearby sensor nodes could save more energy of sensor nodes. Therefore, many existing works used this concept to develop their data management approaches.

Decentralization is helpful for energy saving. However, compared to the global approaches, only local information could be obtained in each sensor node in decentralized approaches. Therefore, the design of decentralized algorithms would be more sophisticated, the performance of decentralized algorithms would be limited, and the computation of decentralized algorithms would not be so complicated. Thus, designing data management systems on WSNs in a decentralized way is energy efficient but the complexity of the data management system will be increased significantly. With such constraints and the power of big data technologies, centralization returns and becomes an option when designing the data management system of WSNs. Using big data systems as data management systems for WSNs is reasonable as the readings in WSNs could be viewed as big data since they satisfy the three characteristics: volume (a large amount of sensor nodes may be deployed and keep reporting readings to sinks), velocity (readings are reported in high speed), and variety (heterogeneous sensor nodes may exist in a sensor network). Obviously, there is a trade-off between centralization and decentralization approaches. This chapter will give an overview of both approaches to provide some guidelines to readers when they build a data management system of WSNs.

The rest of this chapter is organized as follows. Section 2 gives an overview of data management on WSNs, including sensors as a database, query processing mechanisms, and data collection approaches. Section 3 introduces state-of-the-art big data tools and frameworks, and concepts. Section 4 demonstrates some successful examples that build high-performance data management systems for WSNs. Section 5 proposes some future directions for whomever may be involved in the relevant research fields of this chapter, i.e., exploiting big data techniques on WSNs. Section 6 concludes the chapter.

2 DATA MANAGEMENT ON WSNs

The purpose of data management in sensor networks is to separate the logical view (name, access, operation) from the physical view of the data. Users and applications need not be concerned about the details of sensor networks, but about the logical structures of queries. From a data management point of view, the data management system of a sensor network can be seen as a distributed database system, but it is different from traditional ones. The data management system of a sensor network organizes and manages perceptible information from the inspected area and answers queries from users or applications. This chapter discusses the methods and techniques of data management in sensor networks, including the difference between data management systems in sensor networks and in traditional distributed database systems, the architecture of a data management system in a sensor network, the data model and the query language, the storing and indexing techniques of sensor data, the operating algorithms, the query processing techniques, and two examples of data management systems in sensor networks: TinyDB and Cougar.

This section introduces three essential components of a data management system: storage, query processing, and data collection. Section 2.1 introduces how the readings are stored among sensor nodes. Section 2.2 gives an overview about query processing mechanisms in WSNs. Section 2.3 describes how data collection could be achieved.

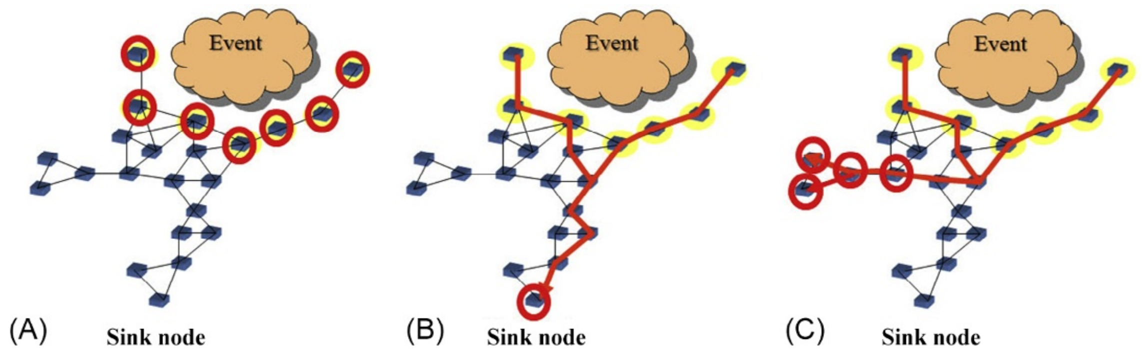

FIG. 2

Data storage approaches. (A) Local storage, (B) external storage, and (C) data-centric storage.

2.1 STORAGE

The first step for data management on WSNs is storage. A naïve approach to store the readings from sensor nodes is to send all the data readings to sinks once a sensor node gets the readings. It is practical in small-scale sensor networks. However, in large-scale WSNs, the large amount of data are generated and relayed by sensor nodes so that the network lifetime will be reduced. The centralized approaches are not practical. To manage such a large amount of data in an energy-efficient way, the decentralized sensor database approaches are usually used for WSNs as one of the most energy-efficient forms of data storage.

The sensor database model views the whole network as a database where each sensor node is the basic unit for storage. The sensor database may be used to store sensor data (the obtained data need to be stored in some way before processing it), to hold the runtime information (e.g., routing tables), and to maintain a history of performance statistics (for performance tuning or debugging).

The widely storage approaches could be categorized into three classes: local storage, external storage, and data-centric storage. Fig. 2 shows three different storage approaches where the cloud-like shape denotes the event, the arrows are the way sensor nodes relaying data, and the circles are the place storing data. *Local storage* refers to the sensor node which measures the physical phenomenon that stores the data. Then, some protocol should be defined in order to allow potential consumers of those data to find and access the nodes where they are stored. The main drawback is the limited memory space of sensor nodes and the local view of storing data. *External storage* refers to the sensor nodes that send back all the readings to sinks, and manage all the data there. The major drawback is mainly from the concern of energy preservation. Data-centric storage mixes the two approaches above together, which defines an event-driven function to decide the sensor nodes where the readings should be sent [1]. That is, relevant data are categorized and named according to their meanings and all data with the same general name will be stored at the same sensor node. Then, when users query the data with a particular name, it can be sent directly to the sensor node that stores those named data.

2.2 QUERY PROCESSING

Query processing in WSNs could be done collaboratively by the server-side and sensor-side. In the server-side, the operation system for sensor networks is executed in the base station, which is responsible for parsing queries, injecting queries into the network, and collecting results as they stream out of the network. Users could submit their queries at the base station. In the sensor-side, the operation system is executed on the sensor nodes, which is responsible for receiving queries, processing queries, communicating, sensing, and sampling.

To provide a user-friendly interface to execute queries, the operation systems for sensor networks usually supports structured query language (SQL)-like query languages. Such languages could describe how users would like data from sensor nodes to be collected, transformed, and aggregated. Note that the query languages usually differ from most significantly from traditional SQL in that its queries are continuous and periodic. The following example gives a declarative query to obtain the sum of light readings from a set of sensors (S1, S2, S3, S4, and S5) every 2 s (i.e., the sampling period is 2 s).

SELECT SUM(light)
FROM S1, S2, S3, S4, S5
EPOCH DURATION PERIOD 2 s

Upon receiving these queries, the sink injects them into the WSN. Routing trees are the most common method of propagating queries and collecting query results from sensor networks, and many routing protocols in various sensor operating systems adopt this approach: the DHV protocol in TinyOS [2], ContikiRPL protocol in Contiki [3], and LiteOS [4]. A routing tree can be viewed as a query tree where the nodes are sensor nodes that participate in the query processing and the edges between nodes represent the routing paths determined by existing routing protocols. Building energy-efficient routing trees with respect to queries, i.e., building routing trees that can minimize the total number of messages of relaying data from sensor nodes to sinks, is an important issue since data transmission are the most costly operation in WSNs.

One of the most popular solutions is to aggregate many messages into one message on their way to sinks. This solution works for some widely used queries, such as MAX, MIN, or SUM,[1] since they can be evaluated inside sensor networks. Fig. 3 shows an illustrative example that the MAX value is going

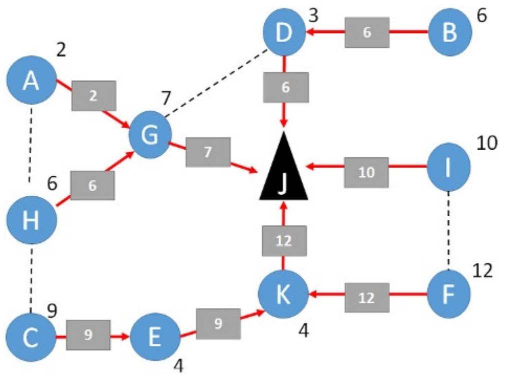

FIG. 3

Executing an aggregation operation MAX.

[1]Readers should be aware of that this technique cannot be used in all cases. For some complicated operations, it takes more sophisticated skills to make it work.

to be extracted from this network. The rectangle node denotes the sink in the network. Sensor nodes are in the circle and the number associated with the circle is the reading of this sensor node. The sensor node and the value that a sensor node passes to are represented by the arrowed line and the number associated with the line, respectively. For example, the reading of A is 2 so that A passes 2 to G. G receives the results 2 and 6 from A and H, respectively, and gets its reading 7. In this case, only 7 has to be passed to the next sensor node since the maximum of 2, 6, and 7 is 7. Therefore, G passes 7 to the next node J. This shows the example of using aggregation techniques on MAX operations. Aggregating MAX operations could reduce the total number of messages needed to 10, instead of 17 without aggregation.

In-network aggregate query processing, in which sensor nodes use aggregate operators to reduce the number of messages, thereby conserves energy [5,6]. Similar to the routing tree, a multicast tree is proposed to minimize the transmission cost from a given source to a set of receivers [7–9]. Generally, the solution of finding multicast trees is based on finding Steiner trees, shortest path tree, and so on. Most studies of multicast trees concentrate on how to construct a multicast tree with the minimum communication cost or minimum data-overhead. The authors in Ref. [10] proposed an approach to share the intermediate results among multiple query trees. When multiple aggregate queries are submitted to WSNs, it is possible to generate the intermediate results of these queries. Sharing these intermediate results of queries can further reduce the number of messages involved for these queries. Fig. 4 gives an illustrative example. There are two query trees where Q1 and Q2 are represented as a solid and dashed line, respectively. The grey nodes are data source. Here, the aggregate operation SUM is executed. The number associated with a sensor is the intermediate result at that sensor. Fig. 4A does not share the intermediate results so that the total number of messages for Q1 and Q2 are 5 and 4, respectively, thereby the total number of messages is 9. Fig. 4B shows an example that the intermediate result of S4 can be shared. Therefore, sensor node S7 of Q2 could directly obtain the intermediate result of S4 without accessing readings at S2 and S3. The total number of messages in Fig. 4B can be derived as 7. Compared to the number of messages incurred in Fig. 4A (i.e., 9),

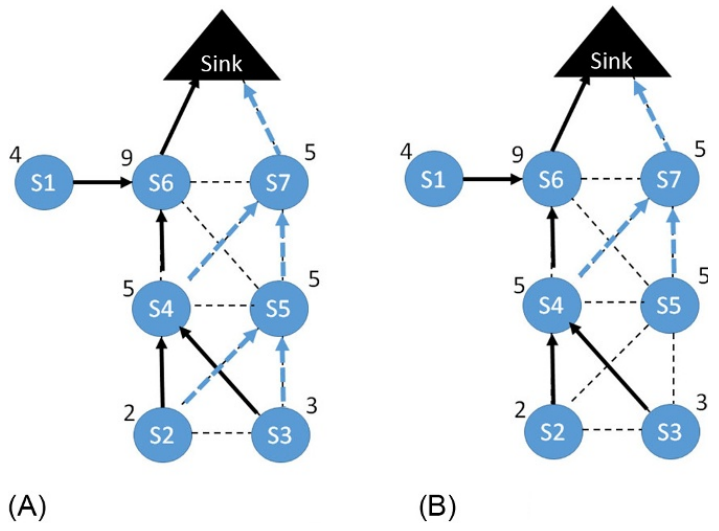

FIG. 4

Aggregation technique: sharing routing trees. (A) Without sharing. (B) With sharing.

the reduced number of messages is 2 (i.e., $9-7=2$). Given a set of query trees, it is possible to investigate further how to share intermediate results among queries. In the example in Fig. 4B, Q1 is a backbone and Q2 is a non-backbone. This figure shows that the backbone is performed as usual and non-backbones must adjust their query trees to access the intermediate results of backbones. Clearly, sharing intermediate results among query trees reduces the number of messages involved in multiple queries.

2.3 DATA COLLECTION

Data collection is a fundamental work in WSNs, which could be viewed as a special query to ask sensors to send the information sensed in monitored areas to the sink. The naïve approach for data collection is asking all the sensor nodes reporting the readings to the sink in a given period. Once the number of sensor nodes are huge, this approach may make sensor nodes drain out their batteries easily. To develop energy-saving strategies, some errors from readings collected are usually allowed, since the sensor nodes themselves may have some errors in their measurements. In general, approximate data collection could be roughly realized by probability-based approaches and clustering-based approaches by exploiting the spatiotemporal locality nature of sensor readings.

Probability-based approaches are used to realize approximate data collection with building probabilistic models of sensing readings collected from WSNs [11,12]. The authors in Ref. [12] explored a model-driven architecture, and a centralized probabilistic model was used to estimate the readings of the sensor nodes. Furthermore, the authors in Ref. [11] employed spatial correlation for approximate data collection, where a replicated dynamic probabilistic model is built to predict sensor readings. If the readings are predicted accurately, sensor nodes will not send their readings to the sink, thereby reducing the communication cost. In these works, probabilistic models need to be built and carefully maintained. It is not easy to build appropriate probability models that capture sensed readings fully, because the reading distribution may vary.

Clustering-based approaches are to find some representative nodes which represent a group of sensors so that only representative nodes need to report their readings to the sink. The authors in Ref. [13] derived an extension of a declarative query, termed a snapshot query, for WSNs. Snapshot queries can be answered via a data-driven approach using a linear regression model to predict readings of 1-hop neighbors. The authors in Ref. [14] formulated data gathering into a connected correlation-dominating set problem to select representative nodes. These representative nodes should form a connected subgraph in order to relay sensed data. Thus, the number of selected nodes should be sufficiently large to form a connected correlation-dominating set. The authors in Ref. [15] proposed a centralized algorithm, named EEDC, which partitions sensor nodes based on spatial correlation into disjointed cliques such that sensor nodes in the same clique have similar readings. Furthermore, a round-robin schedule is employed to share the workload of the data collection in each clique. The authors in Ref. [16] proposed a more sophisticated approach to reduce the number of representative nodes based on the fact that one sensor node may represent sensor nodes at a farther distance, rather than only one-hop. Moreover, extending network lifetime should also take remaining energy of sensor nodes into account. According to the concept above, the authors in Ref. [16] modeled selecting representative nodes into a set-cover problem, developed both centralized and distributed algorithms, and proposed the corresponding maintenance mechanisms to dynamically select new representative nodes.

3 BIG DATA TOOLS

In 1969, Edgar F. Codd proposed the famous *relational model* to provide a declarative method for specifying data and queries. Since then, IBM started to build *System R*—a prototype of managing data and allowed users to send a query to retrieve desired information using a standardized query language "SQL." The database management system (DBMS) became a standard of data storage and processing until the 2000s.

Along with the introduction of personal computers and the invention of the Internet, the volume of data are growing so rapidly such that traditional solutions like DBMS are incompetent to satisfy business needs. The term "big data" was then introduced and corresponding ecosystems of big data are developed, contending as well as collaborating with each other to become the new standard.

Most big data systems can be viewed as workflows: an application is represented by a directed graph that chains one job after another. Each job consumes an input data set and output data set. Most of the time, both input and output data sets are immutable. For example, in the Hadoop distributed system, as we shall discuss in Section 3.2.1, jobs are chained by directory names and data are stored to disk.

Despite the fact that expensive I/O operations are expected in such design, the workflow notion is very popular because it has many advantages. (1) Many clients can consume one dataset without affecting each other. (2) Jobs are loosely coupled so that they can be developed in different language and are set to different schedules/priorities. (3) Failure recovery is straightforward—each transition output acts as checking points. (4) Finally, the flow of data processing are easily traceable.

This section will discuss some popular technologies that were developed for the age of big data. We shall start with the data storage, followed by batch data processing, streaming data processing, and end with a discussion on popular architecture design.

3.1 FILE SYSTEM

The foundation of every big data architecture is to find an efficient and reliable way to store a large volume of data for later processing. The two main strategies of dealing with large data are partitioning and replicating: partitioning means cutting sizable data into pieces so they can be stored in different machines; replication means duplicating the data so that a system can achieve fault recovery and augment the throughput. We shall also discuss the idea of caching, which speeds up the data processing through reducing the expensive I/O operations and data transferring across networks.

Apache Hadoop, an open source big data framework, was initiated in 2006, aiming to be a framework for the analysis and transformation for very large data sets. Hadoop consists of seven components where the most obvious ones are a processing component based on the MapReduce paradigm (see Section 3.2.1), and a distributed file system to store data.

Its corresponding storage system, Hadoop distributed file system (HDFS) became a separate project in 2009. HDFS contain two types of nodes: NameNode and DataNode. NameNode record attributes for files and directories like permissions, modification, namespace, and disk quotas, and content are split into large blocks, say 128 MB, into DataNode. Naturally, a client talks to the NameNode to look up its corresponding DataNode locations. Similar to Google file systems (GFS), to achieve fault tolerance, each block has its replica stored in other DataNodes.

Although HDFS and GFS share a lot of commonalities, one of the key difference between GFS and HDFS is the notion of a lease. In HDFS, when a client is permitted to write to the file, no other client can do so. The writing client needs to renew a lease periodically by notifying the NameNode. This design (a lease) allows Hadoop to schedule tasks easily.

The second generation of Hadoop was introduced in late 2012. One of its new key features was YARN (Yet Another Resource Negotiator), a cluster management technology. The fundamental idea is to separate resource management and job scheduling/monitoring. With the help of YARN, Hadoop can easily maintain a multi-tenant environment, with better security controls and better availability.

3.2 BATCH PROCESSING

3.2.1 MapReduce in Hadoop

MapReduce is a programming model for processing and generating large data sets [17]. It contains two main processes: (1) map(k, v) -><k', v'> and (2) reduce(k', <v'>*) -><k', v'>. The map takes input as key/value pair and produces another intermediate key/value pair. On the other hand, MapReduce is used to aggregate/summarize data—for example, to count the number of words appearing in a document. The map operation breaks content into words:

```
map(String key, String value):
  // key: document identifier
  // value: full text in the document
  for each word w in value:
        Emit Intermediate(w, “1”);
```

A reduce operation adds up counts for each word w:

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
        result += ParseInt(v);
      Emit(AsString(result));
```

A hidden step in between map and reduce is a shuffle step—redistributing/grouping < k′, v′ > * by every word w such that the reduce operation above then sums up the total and emits.

MapReduce programs are not guaranteed to be fast or a panacea for every problem. Authors in Ref. [18] concluded that relational databases still have advantages for several scenarios. However, with the release of Apache Hadoop project, one of the most popular frameworks to support the MapReduce paradigm, MapReduce has been extensively adapted to deal with the big data challenge.

3.2.2 RDD in Spark

Probably one of the most promising breakthroughs on data processing would be the project Apache Spark. It was initiated by Matei Zaharia at UC Berkeley’s AMPLab in 2009, and later became an open source project in 2010.

One of the main shortcomings of MapReduce is that its data flow overlooks reusability. For most data-centric tasks, to complete a task one must perform many linear data flow: reading data from disk, shredding data into pieces, distributing them among network to conduct a map function to process data, summarizing result in the reduce step, and eventually storing results on a disk. Reading from and then writing to disk are expensive operations that should be avoided when possible, they are even more costly if we have to distribute data into machines across a network.

A new data structure, resilient distributed dataset (RDD), was introduced to greatly "speed up" the process through encouraging data caching and operations grouping. An RDD is a read-only distributed data collection that can be rebuilt if needed. According to research in Ref. [19], Spark can perform 10 times faster than the classical map-reduce system because of the usage of RDDs.

Under the hood, operations are categorized into two types: transformations and actions. Transformations refer to operations that can be further optimized by grouping together. Operations like *map*, *filter*, *sample*, *union*, *intersection*, *groupBy*, etc. fall into this category. For example, assuming a long list of integers are provided, our goal is to divide each number by two, and next to report its square. Two operations are in essence mergeable. A similar concept can be extended to merge operations that require no global knowledge—operations that can be completed in one machine without the need of re-shredding and re-distributing. On the other hand, some operations, such as finding the top *N* integers, require a global sorting and thus later operations cannot proceed unless the current operation has been summarized. Operations like *reduce*, *collect*, *count*, *first*, *take*, etc. fall into this category.

In Feb. 2014, Spark became a top-level Apache project. In 2015, Spark project has more than 1000 contributors and is now one of the most active open-source projects. Many libraries have been developed on top of Spark, including:

- Spark SQL: allow SQL queries be written against data;
- Spark Streaming: allow user to write streaming jobs the same way you write batch jobs;
- MLlib: implementation of many machine learning algorithms, such as classification and clustering; and
- GraphX: implementation of many graph algorithms, including PageRank, SVD++, etc.

3.3 STREAMING DATA PROCESSING

The classic big data technology, such as Hadoop MapReduce, achieves high throughput of data processing at a cost of having high latency. When data are batch processed, making data available for (near) real-time analysis becomes a challenge. Recalling the steps of MapReduce, shuffling will only start when all map tasks have completed. The wait for shuffling, the scheduling, and the data transferring across nodes all commit to the high latency.

Certain attempts allow the system to reduce its latency. In this section, we are discussing the advances toward real-time analysis, a.k.a. streaming processing. Several real-time/streaming processing systems for big data have been proposed in recent years [20]. The most famous projects include Storm (Twitter), Spark-streaming (Databricks), and Samza (LinkedIn).

3.3.1 Continuous operator model

One of the pioneers of streaming processing is the project Storm. Its initial release was on Sep. 17, 2011, and it became a top-level Apache open-source project in 2014. Storm, MapReduce Online [21], and many streaming databases are often considered to be based on a *continuous operator model* [22]. In the continuous operator model, long-lived, stateful operators process each record and continuously update their internal states, and then send new records out.

Naturally, some operators might malfunction, and to achieve scalability, two common strategies are often used: (1) to have *replication*—two copies of each node are used for processing the same records, and (2) to use upstream backup—rebuild the failed node's state by sending data to the corresponding operator again. The former approach costs two times the hardware, and requires some synchronization protocols like Flux to coordinate; the latter one requires the system to wait when rebuilding the failed node's state through re-running [23].

3.3.2 Discretized stream model

Another stream processing model, *discretized streams* (D-Streams), was introduced in Spark-streaming. The term "D-Streams" comes from the notion of having a never-ending sequence of RDDs. According to the report from Ref. [24], Spark-streaming is at least two times faster than Storm at their study.

Simply put, D-Streams treats streaming as a series of deterministic batch operations, for example, *map*, *reduce*, and *groupBy*, of fixed duration like 1 s or 100 ms. Computations are considered as a set of short, stateless, deterministic tasks, and intermediate states are stored as RDDs (see Section 4.1.3). Since the input data has been chopped into finer granularity, with the help of RDDs, which try to keep data in memory, a sub-second end-to-end latency is attainable.

D-Stream models are stateless and thus recovery can be achieved by *parallel recovery*—when a node fails, other nodes in the same cluster can work in parallel to recompute the missing RDD in that no state information is stored in the failing node. The system may also periodically create state RDDs, say, by replicating every 10th RDD, as checkpoints to speed up the recovery; however, since the lost partitions can be recomputed in any other nodes in parallel, its recovery is often fast. On the contrary, an upstream backup is slow because it depends on a single idle machine to perform the recovery.

4 PUT IT TOGETHER: BIG DATA MANAGEMENT ARCHITECTURE

Until now we have focused on the functions, libraries, and tools that are used for storing and processing data on both WSNs and big data systems. This section will focus on architecture: how to combine pieces into a solution that satisfy user needs.

To achieve the aforementioned properties, *lambda architecture* was proposed in 2013 [25]. One of the authors in Ref. [25], Mr. Marz, is the creator of the Apache Storm project, and during his time working on Twitter, he developed a concept to build Big Data system as a series of layers, consisting of a batch layer, a serving layer, and a speed layer, as shown in Fig. 5.

4.1 BATCH LAYER

Imagine that a user needs to send a query to be answered by the system. One may consider the operation as an equation:

```
query = function (all data)
```

When the size of data are big, to speed up the process, an obvious way is to have some precomputed intermediate stages (views). That is,

```
batch view = function_1 (all data)
query = function_2 (batch view)
```

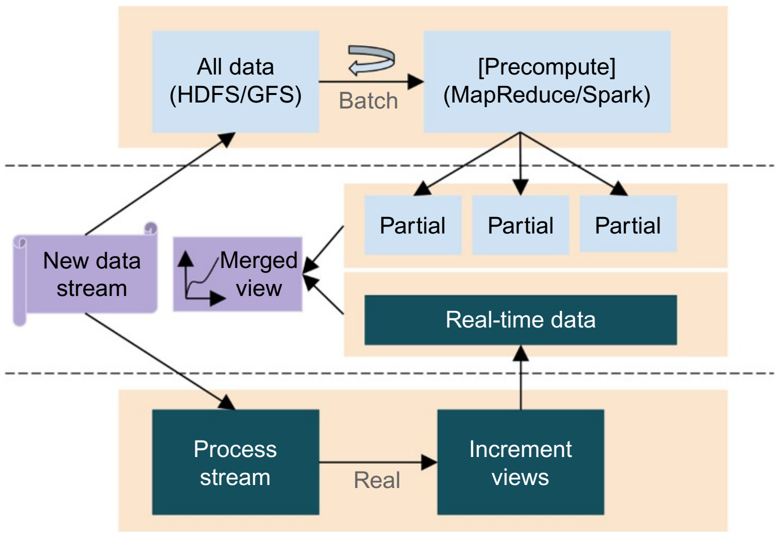

FIG. 5

Lambda architecture.

The batch layer regularly precompute raw data and store output into an immutable, constantly master data set. Batch computations are often simple and straightforward, i.e., acting as a single-thread program, and parallelism comes naturally by consuming only partial input data once at a time—scaling to however many nodes you have available. Eventually, the batch layer produced batch views and these views are then used by the serving layer. In practice, both Hadoop MapReduce and Apache Spark are ideal frameworks to serve as the batch layer.

4.2 SERVING LAYER

The goal of serving layer is to have a framework/application that answers user queries-based views. Since it is usually take a few hours for raw data to be computed, serving queries based on batch views generated from batch layer will be out of data by a few hours. The latency issue can be addressed later by incorporating real-time views generated from speed layer. Cloudera Impala, Elephant DB, and Apache HBase are popular choices for batch-layer output.

4.3 SPEED LAYER

The speed layer aims to reduce the latency by allow arbitrary functions to be computed on the fly based on the incremental data. Instead of accessing the raw data, speed layer updates the real-time views immediately when it receives new data. In practice, stream-processing technologies are often used in this layer, such as Apache Storm, SQLstream, and Apache Spark-streaming.

Every time a batch layer operation is complete, we should discard corresponding pieces in real-time view because they are no longer needed. We can summarize the lambda architecture into the following three formula:

```
batch view = function_1 (all data)
real-time view = function_2(real-time view, new data)
// incremental update
query = function_3 (real-time view, batch view)
```

Note that speed layer require random writes, and thus it is often more vulnerable and complicated, in terms of implementation and operation. If anything goes wrong, one may simply discard the real-time view and in a few hours, the batch layer will help to recover the system into normal state. In practice, not every algorithm can be computed incrementally, and an approximation algorithm is then introduced to get a close answer. For example, HyperLogLog set can be used to compute unique counts [26].

5 BIG DATA MANAGEMENT ON WSNs

5.1 IN-NETWORK AGGREGATION TECHNIQUES AND DATA INTEGRATION COMPONENTS

In-network aggregation is an important concept in both big data management and WSNs. In WSNs, as discussed in Section 2.2, many in-network aggregation approaches are well developed. The main goal is to save more energy for WSNs. The similar functionality exists in the big data systems: data integration components, such as Flume and Kafka, Flume and Kafka are designed to manage data flow, to avoid data explosion, and to unify the data representation. Data integration components usually play a key role in constructing a large-scale system since the incoming data may vary from sources and they may feed in the different destinations. Such systems usually provide data-driven decision and user-defined functions to manipulate data. Therefore, by combining two concepts, it is possible to integrate in-network aggregation techniques in WSNs and data integration components in big data systems.

Authors in Ref. [27] classified the data integration into three categories: complementary, redundant, and cooperative data integration. Fig. 6 shows illustrative examples of these three categories, which

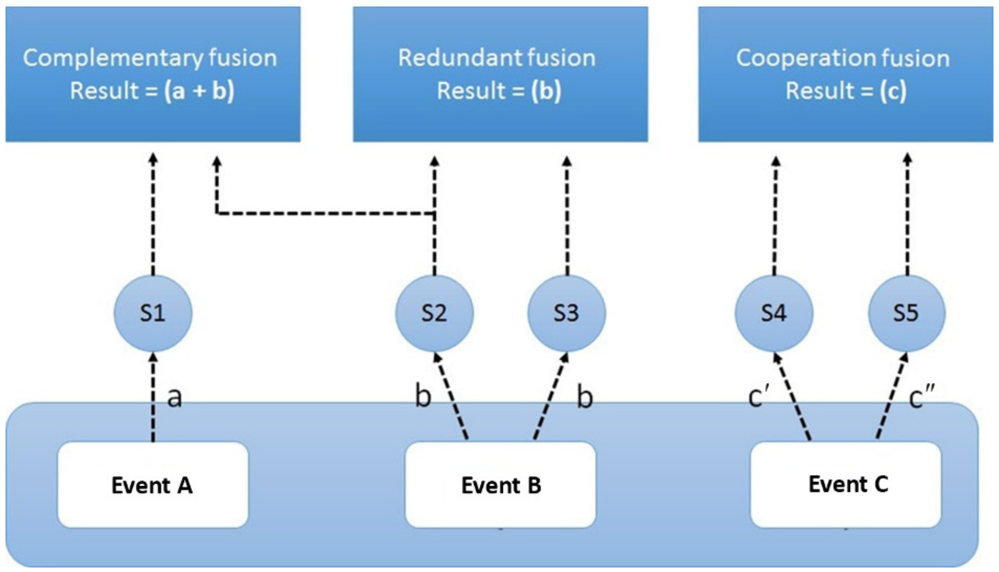

FIG. 6

Three categories of data integration.

are borrowed from Ref. [27]. Complementary data integration is performed when source nodes obtain different pieces of data that need to be fused in order to complete the scene. In the redundant fusion approach, if two source nodes share the same piece of data, the data is first ranked and fused into a high-quality single piece of data. This approach therefore offers trustiness and reliability of sensed data. Cooperative data integration is used when independent sources are fused their data to produce a new piece of data. Cooperative data integration is suitable for the body sensor networks. The killer applications for body sensor networks are human-centric applications like health and sports monitoring. To choose where the data integration modules should be put, one should consider the complexity of the computation needed and the scale of a sensor network. For example, regarding cooperative data integration, the computational power of sensor nodes cannot afford complicated operations but the meaning of readings from different sensor nodes should be checked mutually to make sure the event described by these readings. Moreover, the body sensor networks are usually on a small scale. Therefore, this data integration would be better to put in the big data system (i.e., data integration should be done in a centralized manner).

To conclude, as main sources of big data, exploiting in-network aggregation could prolong the lifetime and contribute to reduction of data volume of the big data, thus accelerating of the values discovery process from this big data. However, the decision where the data integration component is put should be taken the scale of wireless networks and the computational complexity of operations into account.

5.2 EXPLOITING BIG DATA SYSTEMS AS DATA CENTERS

A very popular way to combining WSNs and big data systems is to deploy WSNs as data sources for big data systems and to construct the components of big data systems as a center of data storage and analytics. Interestingly, in early 2000, almost all the centralized approaches were evolved into the corresponding distributed approaches. However, with the rise of power of big data systems, the trend of data management comes back to the centralized approaches. This section gives two cases, which follow this concept to integrate big data systems and WSNs.

5.2.1 Case 1: Fire security system

Here, a fire security system will be discussed [28] which is called the Human Agent Robot Machine Sensor (CFS^{2}H). CFS^{2}H is a multi-agent system that consists of five subsystems. For ease of understanding, the five subsystems are described by the objects in the corresponding subsystems: Sensor, BigData, Human, Agent, and Robot. The interaction of this system is as follows:

1. Sensors collects readings of the environment, to send readings to the central big data system for further analysis. While detecting possible fire threats, Sensors will send the notification to Human, and send commands to Agent to detect the fire location.
2. Agent finds the locations of fire accident and sends the precise coordinate to Human and Agent.
3. Human and Agent move to the location.
4. Agent sends the notification and live video to Human.
5. Human controls Agent to direct people close to the fire accident to the nearest exit.

Here, the BigData subsystem in this work was designed to be a centralized place where all the data from all other subsystems could be processed. While every subsystem has to communicate with two or three other subsystems, BigData subsystem has to communicate with all of them. Its main role is to be a central data analyzing and processing station. It should be able to receive and send data to any subsystem. In case of any danger, BigData subsystem has to send warnings and commands to proper subsystems to acknowledge them about it. More specifically, WSN and BigData subsystems establish communication with other subsystems and exchange information without any obstacles. The BigData subsystem stored all the incoming data; received readings from Sensor, query from Human, location and temperature from Agent, location and current state from Agent, and sent current state to Human, and threshold (used to detect fire accidents) to Sensor.

5.2.2 Case 2: Environment monitoring system

An air quality monitoring system is discussed here [29]. Authors in Ref. [29] built an environment monitoring system which consists of WSNs to collect the readings of air pollutant, like SO_2, CO, NO_2, and so on, and a big data pipeline to import, process, and store data from sensor nodes.

The whole system is shown in Fig. 7, which is borrowed from Ref. [29]. This system consists of three parts:

1. Data Acquisition Module (DAM): This module includes the sensing-related devices, such as sensor nodes and sinks.
2. Message Oriented Middleware (MOM): This module is responsible for the communication between DAM and DPM. This component transmits the information obtained from the base stations to the processing and storage system, providing independence of operation for both modules. A MOM is a

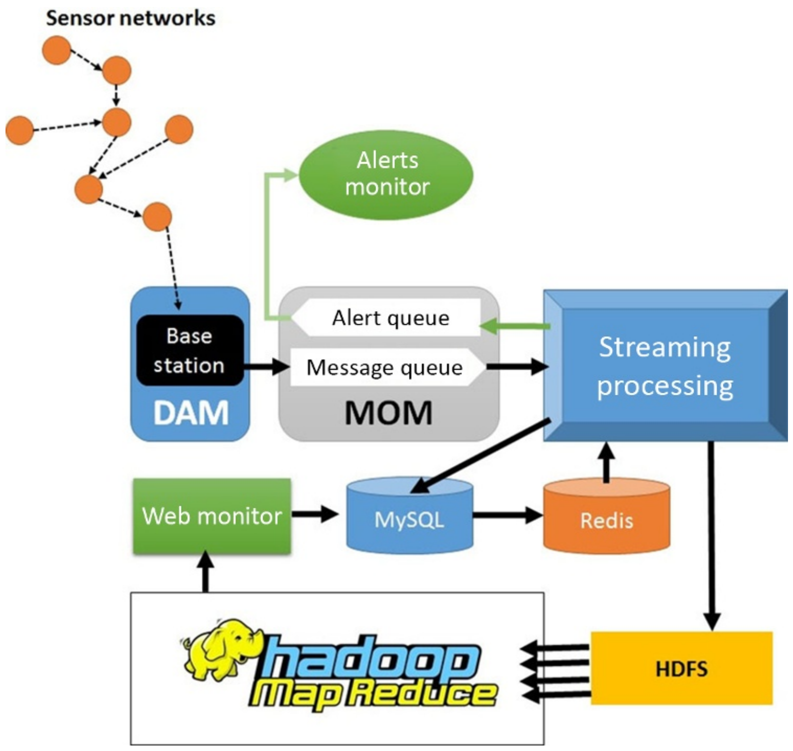

FIG. 7

The architecture of the environment monitoring system.

distributed component that provides asynchronous messaging between applications as well as reliable information delivery mechanisms, guaranteeing the independence of their architectures. This module could be built by message queue systems, such as RabbitMQ or Kafka.

3. Data Processing Module (DPM): This module operates in both streaming processing system and batch processing manners. Stream processing for detecting and sending real-time alarms if necessary, and for computing some statistics, and MapReduce for further data analysis. As mentioned in Section 4, this work also uses Lambda architecture.

Batch processing is executed every 24 h. The data on HDFS are computed by MapReduce to obtain some statistics and some widely used query results, such as mean temperature of each base station per day, min/MAX/avg levels about pollutants, and so on. For stream processing, Storm is used to execute several operations in real-time where each operation could be represented as a spout or a bolt and the relationship among them could be represented as a Storm topology. For example, statistics were computed in three levels of aggregation: sensor, Gateway, and Base station, which are done by SensorStadisticsBolt, GatewayStatisticsBolt, and BaseStatisticsBolt, respectively. Interested readers could refer to Ref. [29] for more detail. In addition to the components for batch processing and stream processing, an in-memory database Redis is used as a cache to store the information of sensor nodes, including identifier, locations, type of sensors, and so on.

To conclude, this work not only consider the big data system as a centralized storage and processing center, but also consider how to combine message queues, and to exploit lambda architecture in their implementation. However, the role of WSNs remains at a data source. These cases told us that a promising way to combine WSNs and big data systems is to move all the computation to the big data system side and keep operations of WSNs as easy as possible. It makes sense because of the computational power of sensor nodes are limited. However, the well-developed approaches in WSNs have not integrated with the big data systems so far. From Section 5.1, one could understand that it may be possible to put some of easy tasks in big data systems in WSNs by the well-developed algorithms. A future research direction may be to design a more sophisticated architecture to do so.

6 CONCLUSION

This chapter gives an overview of the data management issues and solutions in WSNs and big data systems. Specifically, three major issues for data management are introduced: storage, query processing, and data collection. Some big data storage, computation models, and the architecture are introduced. Finally, some case studies of exploiting big data systems for data management on WSNs are discussed. The future works could aim at taking a good balance between centralization (get computation back to big data systems) and decentralization (put computation down to sensor nodes).

REFERENCES

[1] Shenker S, Ratnasamy S, Karp B, Govindan R, Estrin D. Data-Centric storage in sensornets. SIGCOMM Comput Commun Rev 2003;33:137–42.

[2] Dang T, Bulusu N, Chi Feng W, Park S. DHV: A code consistency maintenance protocol for multi-hop wireless sensor networks. In: Proceedings of European conference on wireless sensor networks (EWSN); 2009. p. 327–42.
[3] Tsiftes N, Eriksson J, Dunkels A. Low-power wireless IPv6 routing with ContikiRPL. In: Proceedings of international conference on information processing in sensor networks (IPSN); 2010. p. 406–7.
[4] Cao Q, Abdelzaher TF, Stankovic JA, He T. The LiteOS operating system: Towards Unix-like abstractions for wireless sensor networks. In: Proceedings of international conference on information processing in sensor networks (IPSN); 2008. p. 233–44.
[5] Madden S, Franklin MJ, Hellerstein JM, Hong W. TAG: A tiny aggregation service for ad-hoc sensor networks. In: Proceedings of symposium on operating systems design and implementation; 2002.
[6] Madden S, Franklin MJ, Hellerstein JM, Hong W. TinyDB: An acquisitional query processing system for sensor networks. ACM Trans Database Syst 2005;30(1):122–73.
[7] Mala C, Selvakumar S. Construction of an optimal multicast tree for group communication in a cellular network using genetic algorithm. Comput Commun 2006;29(16):3306–12.
[8] Ruiz PM, Gómez-Skarmeta AF. Approximating optimal multicast trees in wireless multihop networks. In: Proceedings of IEEE symposium on computers and communications (ISCC); 2005. p. 686–91.
[9] Sheu P-R, Chen S-T. A fast and efficient heuristic algorithm for the delay- and delay variation-bounded multicast tree problem. Comput Commun 2002;25(8):825–33.
[10] Hung C-C, Peng W-C. Optimizing in-network aggregate queries in wireless sensor networks for energy saving. Data Knowl Eng 2011;70(7):617–41.
[11] Chu D, Deshpande A, Hellerstein JM, Hong W. Approximate data collection in sensor networks using probabilistic models, In: In Proc. of ICDE; 2006. p. 48–59.
[12] Deshpande A, Guestrin C, Madden S, Hellerstein JM, Hong W. Model-driven data acquisition in sensor networks. In: In Proc. of VLDB; 2004. p. 588–99.
[13] Kotidis Y. Snapshot queries: Towards data-centric sensor networks. In: In Proc. of ICDE; 2005. p. 131–42.
[14] Gupta H, Navda V, Das SR, Chowdhary V. Efficient gathering of correlated data in sensor networks. In: In Proc. of MobiHoc; 2005. p. 402–13.
[15] Liu C, Wu K, Pei J. An energy-efficient data collection framework for wireless sensor networks by exploiting spatiotemporal correlation. IEEE Trans Parallel Distrib Syst 2007;18(7):1010–23.
[16] Hung C-C, Peng W-C, Lee W-C. Energy-aware set-covering approaches for approximate data collection in wireless sensor networks. IEEE Trans Knowl Data Eng 2012;24(11):1993–2007.
[17] Dean J, Ghemawat S. MapReduce: A flexible data processing tool. Commun ACM 2010;53(1):72–7.
[18] Pavlo A, et al. A comparison of approaches to large-scale data analysis. In: Proceedings of the 2009 ACM SIGMOD international conference on management of data. Rhode Island, USA: ACM; 2009.
[19] Zaharia M, et al. Spark: Cluster computing with working sets. HotCloud 2010;10:10.
[20] Liu X, Iftikhar N, Xie X. Survey of real-time processing systems for big data, In: Proceedings of the 18th international database engineering & applications symposium. ACM; 2014.
[21] Condie T, et al. MapReduce online. NSDI 2010;10(4):313–28.
[22] Toshniwal A, et al. Storm@ twitter. In: Proceedings of the 2014 ACM SIGMOD international conference on management of data. Utah, USA: ACM; 2014.
[23] Shah MA, Hellerstein JM, Brewer E. Highly available, fault-tolerant, parallel dataflows. In: Proceedings of the 2004 ACM SIGMOD international conference on management of data. Paris, France: ACM; 2004.
[24] Zaharia M, et al. Discretized streams: Fault-tolerant streaming computation at scale. In: Proceedings of the twenty-fourth ACM symposium on operating systems principles. Pennsylvania, USA: ACM; 2013.
[25] Marz N, Warren J. Big Data: Principles and best practices of scalable realtime data systems. New York, USA: Manning Publications Co; 2015.
[26] Flajolet P, et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In: DMTCS proceedings 1; 2008.