Distributed Sampling Storage for Statistical Analysis of Massive Sensor Data

Hiroshi Sato, Hisashi Kurasawa, Takeru Inoue, Motonori Nakamura, Hajime Matsumura, Keiichi Koyanagi

Distributed Sampling Storage for Statistical Analysis of Massive Sensor Data

Hiroshi Sato[1], Hisashi Kurasawa[1], Takeru Inoue[1],
Motonori Nakamura[1], Hajime Matsumura[1], and Keiichi Koyanagi[2]

[1] NTT Network Innovation Laboratories, NTT Corporation
3-9-11, Midori-cho, Musashino, Tokyo, Japan
[2] Faculty of Science and Engineering, Waseda University
2-7 Hibikino, Wakamatsu-ku, Kitakyushu, Fukuoka, Japan

Abstract. Cyber-physical systems interconnect the cyber world with the physical world in which sensors are massively networked to monitor the physical world. Various services are expected to be able to use sensor data reflecting the physical world with information technology. Given this expectation, it is important to simultaneously provide timely access to massive data and reduce storage costs. We propose a data storage scheme for storing and querying massive sensor data. This scheme is scalable by adopting a distributed architecture, fault-tolerant even without costly data replication, and enables users to efficiently select multi-scale random data samples for statistical analysis. We implemented a prototype system based on our scheme and evaluated its sampling performance. The results show that the prototype system exhibits lower latency than a conventional distributed storage system.

Keywords: data accuracy, random sampling, relaxed durability

1 Introduction

Thanks to advances in sensor devices and networking technologies, it is becoming possible to retrieve people's behavior, object states, and environmental conditions as sensor data in real-time. These sensor data are massive and continuously increasing since there are an infinite number of targets to sense in the physical world. Applications with significant socio-economic impact will be developed using such physical world data. Systems that interconnect the cyber world with the physical world are referred to as *cyber-physical systems* (CPSs)[1]. Applications of CPSs include, but not limited to, environmental monitoring, transportation management, agriculture management, pandemic prevention, disaster recovery, and electric grid management.

There are many technical challenges with CPSs. One key challenge is providing real-time sensor data to applications. Gathered sensor data are attractive for various applications. CPSs have to simultaneously deal with huge amounts of sensor data for applications. Because the physical world is ever-changing, CPS applications must timely and continuously adapt to these changes, predict what

will occur, and perform the appropriate actions. Of course, not every CPS application is required to do these in real time. CPS applications can be divided into two types; real-time, as mentioned above, which place a high priority on timely data acquisition, and batch processing, which places a high priority on accurate results and attempts to use data exhaustively. For batch processing applications, software frameworks, such as MapReduce[2], have been proposed and already applied to some domains. They tend to be extremely large; only limited organizations have environments equipped to process such massive data exhaustively. These applications are a minority. Thus, our target applications are real-time ones. We adopt a kind of approximate query processing technique to speed up the data providing; it is a sampling. The sampling reduces the size of data, and reduce the latency of processes.

Another key challenge is reducing storage cost for sensor data. Since sensor data is massive and continuously increasing, storage cost is a serious issue. Although fault-tolerance is generally mandatory, it is not realistic in terms of cost to keep doubling, tripling, etc.. data. We need another scheme for ensuring fault-tolerance. It sounds impossible to ensure fault-tolerance without data replication; however, we realize this by a novel approach that emphasizes statistical properties of the data instead of individual data values. This approach enabled us to translate the data durability into the data accuracy. Therefore, we can relax the data durability unless applications always request the maximum accuracy.

The rest of this paper is organized as follows. Section 2 discusses requirements for storage systems that process massive amounts of sensor data. Section 3 describes our approach and proposes a storage scheme that satisfies these requirements. Section 4 evaluates the performance of a prototype system by comparing it with a conventional storage system. Section 5 describes related work and compares them with ours. Section 6 concludes the paper and describes future work.

2 Requirements

To clarify the requirements for storage systems, we first describe the characteristics of data generated by sensors in a large-scale sensor network. Next, we describe the characteristics of applications using such data. Then, we discuss the requirements for storage systems that store and query the data.

2.1 Characteristics of Sensor data

Individual sensor data records are tiny but are collectively massive. A single sensor data record generally consists of a sensing value/values and its/their metadata such as sensor, temporal, and spatial attributes. On the other hand, there is a massive amount of sensors in a network, which continuously generate and transmit data. Although each record is small, they collectively become an enormous data stream through a large sensor-network. Consequently, sensor data are massive and continuously increasing.

A sensor data record is just a sample of a physical condition, e.g., temperature. Sensor data records may have a margin of error in their sensing values and are sparse. These are inherently defective data; therefore, analyzers must lump them together at a suitable granularity to enhance the quality of each cluster. Statistical analysis is thus a quite natural method for understanding sensor data.

In summary, sensor data are massive and defective but each record is unimportant; therefore, they need to be lumped together and statistically analyzed.

2.2 Application Characteristics

When processing a massive amount of data, most applications, especially real-time applications, tend to quickly obtain an overview of the data rather than inquire about each piece of data since analyzing all data is too detailed and costly. In this case, "overview" involves statistics such as averages, trends, and histograms. A key factor of statistical analysis is its accuracy.

Generally, latency of analyzing correlates with the size of the data to process, and accuracy also correlates with the size of the data; consequently, there is a trade-off between speed and accuracy. As mentioned above, our target applications place a high priority on quick analysis. On the other hand, each application has its own accuracy requirement. In addition, the level of accuracy depends on the context of the application. The accuracy requirement dynamically changes. Therefore, the most appropriate data set for an application is the minimal set satisfying its dynamic accuracy requirement.

In summary, applications prioritize quick response. They require the minimal set of data that satisfies their dynamic accuracy requirement.

2.3 Storage Requirements

So far, we described the characteristics of sensor data and applications. Now, we discuss the requirements for storage systems.

The first requirement is *scalability* because sensor data are massive and continuously increasing. Although there are other ways of mitigating the increasing amounts of data, such as compression and disposition, they have the following disadvantages. Compression requires encoding and decoding, which increases latency of processing. Disposition is effective if all potential applications using the data can share a common policy to dispose of data; however, this is prohibitively difficult. Thus, storage must be scalable, which requires adopting a distributed architecture.

The second requirement is *fault-tolerance*. Data durability is mandatory for common storage systems. Slight data loss is acceptable in the case of sensor data, provided that the loss is not biased. Since sensor data should be interpreted by a statistical process, non-biased data loss is not critical. It does not detract from the availability of the system; therefore, fault-tolerance is established. In other words, storage must maintain the statistical properties of data even if it loses partial data. We call this statistical stability.

The last requirement is *quick sampling*, which is a real-time requirement of applications. Reducing the data size to process naturally reduces latency. In statistical analysis, the required size of a sample depends on the accuracy requirement. Therefore, storage must arbitrarily provide the size of a sample according to the application's accuracy requirement.

In the next section, we propose a storage scheme satisfying these three requirements: scalability, fault-tolerance, and quick sampling.

3 Distributed Sampling Storage Scheme

Our scheme is designed for efficient statistical processing and inherently offers a sampling function. The main idea is to distribute random samples among servers to achieve high scalability, fault-tolerance, and efficient sampling.

Our scheme consists of one manager, data servers, and clients. The manager monitors the data servers and provides their live list to clients. The data servers receive requests of inserting, reading, and deleting records from clients and manage them. While the architecture is almost the same as other distributed storage scheme, the main point of our scheme is the method of clients inserting records.

We first explain our approach and procedure, then describe quick sampling, load balancing, and fault-tolerance techniques in detail.

3.1 Approach

We believe that the fundamental cost reduction technique for analyzing data is sampling, and a data storage system should natively provide a sampling function. Although many data mining techniques have been proposed, they require huge computational cost when analyzing all original records. We often extract a random sample from the original and apply data mining techniques to the sample for fast analysis. Randomness of sampling is important for preventing information drop. If sampling is executed regularly, some high-frequency data components may be dropped. Therefore, we developed our scheme on the basis of random sampling processing.

The following conditions are necessary to acquire a fair random sample:

- the data records in the sample are extracted randomly from all original records, and
- the data records in the sample are not biased toward the extraction process.

We assumed that a distributed storage scheme enables efficient sampling if it stores a fair random sample on each node. For the fairness, we adopt a simple method, which is to randomly select a data server to insert each record. If every data server stores a fair random sample, the sampled records are not the same among the data servers but have the same statistical feature. In other words, each server has statistical redundancy with the rest of the servers. Our scheme involves random sampling at inserting records by clients. We discuss load balancing and fault-tolerance from the viewpoint of sampling.

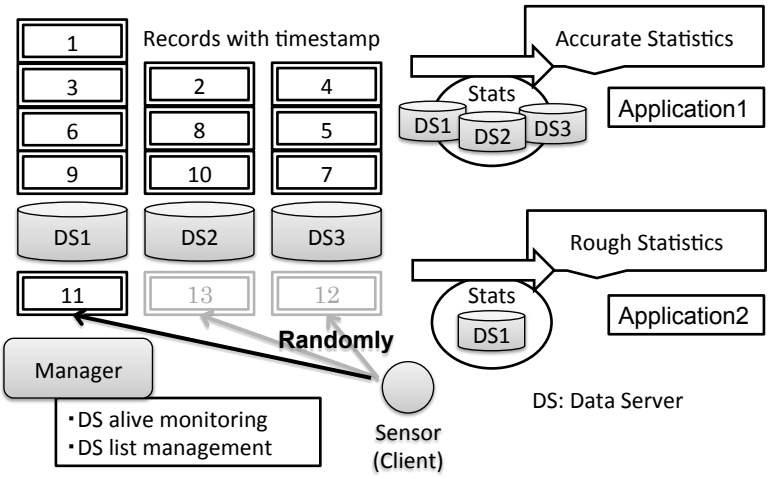

Fig. 1. System overview of distributed sampling storage.

3.2 Procedure

We describe the management, insertion, read, and deletion processes as follows. The system overview is shown in Figure 1. The effectiveness of each process is described in Sections 3.3 to 3.5.

Management The manager maintains live data servers at all times for fault-tolerance and fair sampling. The manager monitors the data servers and maintains a stable scheme. Furthermore, the total number of live data servers determines the sampling rate of the records in a data server. We use the live data server list for not only managing data server status but also for maintaining accurate statistics.

1. The manager accesses the data servers at regular intervals and maintains the live data server list.
2. The manager stores the times when and IDs of data servers join or leave the storage system.

Insertion A client inserts a record into the randomly selected data server for fair sampling and load balancing. The client randomly selects a data server at each insertion so that data servers store fair random samples. This fair random selection also makes loads of data servers be balanced.

1. A client receives the present data server list from the manager.
2. The client randomly selects one of the data servers in the list for each record.
3. The client directly sends the record insertion request to the data server.
4. If the data server leaves unexpectedly, the client should not reselect another data server. Instead, the client disposes the request to maintain fairness and sampling rate.

Read A client changes the number of accessed data servers on the basis of the sampling rate. Each sample in the data servers does not overlap and has the same statistical feature. Thus, the client can modify the sampling rate based on the number of accessed data servers.

1. A client sends the read request of records in the insertion time range, and the sampling rate to the manager.
2. The manager calculates enough data servers to satisfy the sampling rate by referring to the data server list in the insertion time range, as we discuss in Section 3.3.
3. The manager notifies the client of the data server list regarding the request.
4. The client sends the record read request of the time range to the data servers in the list.

Deletion The manager does not manage which data server stores a record. The manager only detects the live data servers for each insertion time. Thus, a client should send the deletion request to all the data servers to which the client may insert records.

1. A client receives the live data server list at the time of inserting the record from the manager.
2. The client sends the record deletion request to all the data servers in the list.

3.3 Quick Sampling Technique

The sampling rate of our scheme can be modified on the basis of the number of accessed data servers.

Each data server has randomly sampled records. If there are N records and n data servers, every data server stores nearly N/n sampled records. If a client needs only the N/n sampled records, the client can quickly obtain it by reading all the records in any one of the data servers. However, clients usually require sampled records with various sampling rates. We should prepare a sampling technique for different sampling rates.

A client can modify the sampling rate by changing the number of accessed data servers because the data servers have non-overlapping and well balanced records. As a result, our scheme returns sampled records quickly regardless of the sampling rate. Let us consider the relation between the number of data servers and the sampling rate α $(0 < \alpha \leq 1)$. We classify this relation into the following three cases.

Case 1 If $\alpha < 1/n$ holds, the request size is smaller than the size of the sampled records in a data server. Then, a client reads all the records in any one of the data servers and randomly extracts its sampled records by setting the sampling rate to $n \cdot \alpha$.

Case 2 If there is a natural number m $(m \leq n)$ such that $\alpha = m/n$ holds, a client may access any m of the data servers and read all the records.

Case 3 Otherwise, it is a combination of cases 1 and 2. A client reads all the records in any $\lceil n \cdot \alpha \rceil$ of the data servers and randomly extracts from one of them at a sampling rate of $n \cdot \alpha - \lfloor n \cdot \alpha \rfloor$.

3.4 Load Balancing Technique

Many data servers are controlled by the manager in our scheme. However, we avoid concentrating heavy loads on the manager and balance the data server loads.

Although the manager monitors data servers and shares their status with clients, the manager does not read or write a record. That is, the manager only updates and provides the data server list. As a result, the traffic of the manager is comparatively small.

The data server loads are due to almost exclusively the requests from clients and are well balanced. There is no synchronization and no status update access among data servers. Since a client randomly selects a data server from the data server list, the insertion requests are not concentrated on a specific data server. Moreover, the reading loads of data servers can be distributed. This is because the manager can determine the load of data servers and select less loaded ones to notify the client.

Our scheme scales out as follows. The new data server does not receive a record until the manager includes it into the data server list. After appearing in the list, a client, who receives the updated list, starts inserting records into the new data server. Over time, the load among the data servers will balance. Note that the data servers in our scheme do not re-allocate records to each other. We cut the record modification process among the data servers to simplify the scheme.

3.5 Fault-Tolerance Technique

Acquiring just a sample is sufficient for most clients. If a client requires a random sample whose sampling rate is α and the records are in n data servers, the client receives the records from $\lceil n \cdot \alpha \rceil$ data servers. Because the data servers have a non-overlapping and well balanced set of records, the client can access any $\lceil n \cdot \alpha \rceil$ data server from the live data servers. That is, the client can obtain the same statistical records if $n - \lceil n \cdot \alpha \rceil$ data servers are left.

The above observation suggests that our scheme does not require costly replication. We believe that statistical stability is important, and our scheme satisfies stability without replication. Our scheme involves fault-tolerance at fair random sampling.

4 Evaluations

In this section, we compare the sampling performances in a prototype distributed sampling storage system based on our proposed scheme with that of a conven-

tional system. We show that the prototype system outperforms the conventional one in sampling and is scalable.

We implemented our simple prototype system using PostgreSQL servers[3] as back-end databases, and conducted two experiments to evaluate its sampling latency and scalability. We selected pgpool-II[4] as a target for comparison, which is an open source middleware working between PostgreSQL servers and a client.

4.1 Experimental Setup

Environments. The experimental environment consists of five PCs (Intel Xeon quad-core X3450 2.55 GHz quad-core, 8-GB memory, 3.5-inch 250-GB 7,200-rpm SATA HDD $\times 4$, CentOS5.6 64 bit) for storage and one PC (Intel Core2 quad-core Q9950 2.83 GHz, 8-GB memory, 3.5-inch 160-GB 7,200-rpm SATA HDD, CentOS5.6 64 bit) for clients. They are connected over Gigabit Ethernet.

Since each storage PC is quad-core and has four physical disks, each one can be regarded as an independent server. Then, on each storage PC, a maximum of four PostgreSQL 8.1 server processes run. Each process links to a database on each disk of the PC. On the client PC, several client processes can run simultaneously.

A record is a fixed length of 56 bytes. Each record consists of fields of record ID, insertion time, sensing time, client ID, client IP address, and value.

Distributed Sampling Storage. To compare the sampling performance derived from the difference in data distribution, we omitted the data server management processes from the prototype system. Therefore, the system only consists of PostgreSQL servers as data servers and clients implemented in C++. The data servers run on four storage PCs, and the clients run on one client PC. Clients detect the live server list from the beginning, randomly select data servers to insert records, and select data servers in turn to read records for load-balancing.

Pgpool-II. We use pgpool-II 3.0 in parallel query mode in which data can be split among multiple data servers, so that a query can be executed on all the servers concurrently. The system consists of PostgreSQL servers as data servers, a pgpool server as a proxy server between a client and data servers, and clients implemented in C++. The data servers run on four storage PCs, the proxy server runs on one storage PC, and the clients run on one client PC. Clients always access the proxy server both to insert and to read. Then the proxy server inserts or reads instead of the clients. In the last read process, clients also extract a sample from the acquired records at the required rate.

4.2 Evaluation 1: Latency of Sampling

We evaluated the sampling response with respect to the sampling rate α, where $0.02 \leq \alpha \leq 1.0$. In preparation, we stored $10,000/\alpha$ records into each storage with 10 data servers. We then simultaneously measured the latency for sampling

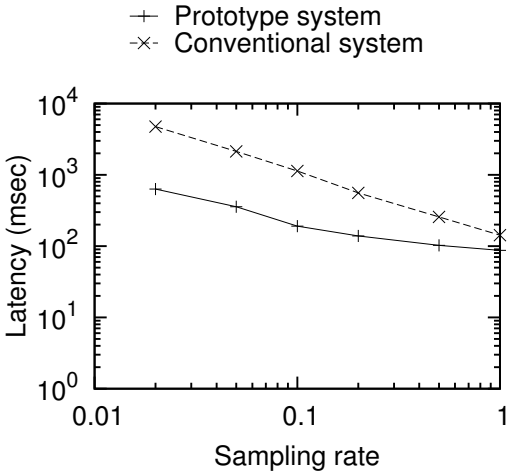

Fig. 2. Latency with respect to sampling rates.

from the storage at a rate of α by 5 clients. A total of (probably overlapping) approximately $50,000$ sampled records were acquired.

Figure 2 shows the results. The horizontal axis is the sampling rate and the vertical axis is the latency. The prototype system outperformed the conventional system in sampling rate α, and was about ten times faster, where $\alpha \leq 0.1$. This gap is nearly equal to the data size to access. With α smaller than 0.1, the prototype system accessed records in just one data server, whereas the conventional system accessed those in all ten data servers.

4.3 Evaluation 2: Scalability

Next, we evaluated the sampling response with respect to the number of data servers n, where $1 \leq n \leq 10$. In preparation, we stored $100,000$ records into each storage with n data servers. We then simultaneously measured the latency for sampling from the storage at a rate of 0.1 by 5 clients. A total of (probably overlapping) approximately $50,000$ sampled records were acquired.

Figure 3 shows the results. The horizontal axis is the number of data servers and the vertical axis is the latency. The prototype system outperformed the conventional system for any number of data servers n, and became faster as n increased, whereas the conventional system did not. Thus, the prototype system is scalable.

5 Related Work

Stream database management systems are being extensively studied. Representative examples are STREAM[5] and Aurora[6]. These systems are mainly aimed

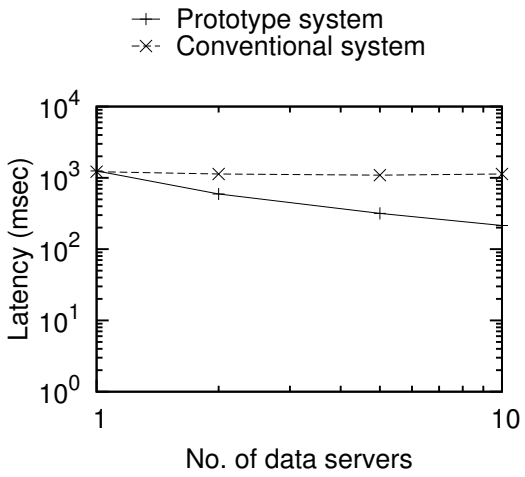

Fig. 3. Latency with respect to number of data servers.

at support for continuous query over a data stream and do not support sample extraction from stored data in the repository; thus, they cannot enable efficient sampling.

Sampling-based approximate query processing is also being studied. The aim is quick sampling or answering. As a classical example, Olken et al. studied random sampling on relational databases. They proposed several efficient algorithms[7][8]. Babcock et al.[9] proposed a method of providing fast approximate answers in decision support systems. This method combines samples selected from a family of non-uniform samples to enhance approximation accuracy. Pol et al.[10] proposed online algorithms for maintaining very large on-disk samples of streaming data. These algorithms reduce disk access in updating samples on disk by storing them based on their proposed geometric data model. These studies enable efficient sampling; however, they have to keep a copy of the data in extra storage for fault-tolerance. Therefore, they are costly.

Reeves et al.[11] proposed a framework to archive, sample, and analyze massive time series streams, especially for data center management. This framework is aimed not only for speed but for reducing the size of archives by compression and summarization. This enables simultaneous quick answering and cost reduction; however, it is difficult to share the stored data among various applications since compression parameters and summarization need to be optimized for each application. In addition, the system still needs to duplicate data for fault-tolerance.

6 Conclusion

We proposed a distributed sampling storage scheme to efficiently sample from a massive amount of sensor data. Clients insert a record into the randomly selected data server, then each data server independently stores a fair random sample; consequently, this storage is scalable and fault-tolerant even without costly data replication. Experimental results showed that the prototype system exhibited lower latency of the sampling process than the conventional system, and it is scalable by increasing the number of data servers. We believe that our scheme make a big contribution to the realization of CPSs.

In the future, we plan to add more flexibility to sampling for a wide range of applications.

References

1. Lee, E.A.: Cyber Physical Systems: Design Challenges. Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on, pp. 363–369, (2008)
2. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51, 1, pp. 107–113 (2008)
3. PostgreSQL, `http://www.postgresql.org/`
4. Pgpool Wiki, `http://www.pgpool.net/`
5. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: STREAM: The Stanford Data Stream Management System. Technical Report, Stanford InfoLab. (2004)
6. Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Erwin, C., Galvez, E., Hatoun, M., Hwang, J. H., Maskey, A., Rasin, A., Singer, A., Stonebraker, M., Tatbul, N., Xing, Y., Yan, R., Zdonik, S.: Aurora: A Data Stream Management System (Demonstration). In proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '03) (2003)
7. Olken, F., Rotem, D., Xu, P.: Random sampling from hash files. Proc. SIGMOD '90, pp.375–386 (1989)
8. Olken, F., Rotem. D.,: Random sampling from B+ trees. Proc. VLDB '89, pp.269–277 (1989)
9. Babcock, B., Chaudhuri, S., Das, G.: Dynamic sample selection for approximate query processing. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD '03). ACM, 539-550. (2003)
10. Pol, A., Jermaine, C., Arumugam, S.: Maintaining very large random samples using the geometric file. The VLDB Journal 17, 5, pp. 997–1018 (2008)
11. Reeves, G., Nath, J. L. S., Zhao, F.: Managing massive time series streams with multi-scale compressed trickles. Proc. VLDB Endow. 2, 1, pp. 97–108 (2009)