Fast Archiving and Querying of Heterogeneous Sensor Data Streams

Ying Zhu
University of Ontario Inst. of Tech.
ying.zhu@uoit.ca

Ken Q. Pu
University of Ontario Inst. of Tech.
ken.pu@uoit.ca

Abstract

We propose techniques based on bitmap-based indices to efficiently store heterogeneous (i.e., time-varying schema) streaming data, supporting fast archiving and query throughput while being space-efficient. We also present the architecture and performance evaluation of a system implementation.

1 Introduction

Many modern and emerging applications, in business and science, require the efficient management of high volumes of data streams. Managing these data streams poses challenges that are distinct from those addressed by traditional database management systems [2]. For instance, sensors streaming to the same base station could differ in the types of data they monitor and send, while the same sensor could emit different types of data at different times. In commercial applications where radio frequency identification (RFID) technology is used, the RFID tags are scanned by the RFID edge servers, which in turn stream the data to the database. Depending on the tagged item, not only the data itself but the *structure* of the data could be very different. These share a number of common elements: (1) Multiple sources are streaming data to one receiver. (2) Data is streaming at high rates. (3) A huge accumulation of readings must be stored in a short time. (4) The data are highly heterogeneous. A system that manages these infinite heterogeneous data streams must satisfy the following requirements: (a) Rate control mechanism to deal with fast streams; (b) Adaptive indexing for time-varying data structures; (c) Efficient data storage; (d) Everything must be append-only due to volume.

We developed a bitmap-based indexing method that is fast and minimizes storage space for heterogeneous data streams. We further developed a system implementation, *ArQSS* (archiving and querying sensor streams), that provides fast archiving and query processing (for filtering queries), with an asynchronous performance monitor, and a run-time adaptive controller. The performance of *ArQSS* is evaluated, through extensive experiments and comparisons to alternatives. We show that *ArQSS* is superior in overall efficiency in archiving, query response, and storage space.

The remainder of the paper is organized as follows. Related work is discussed in Sec. 2. We describe in Sec. 3 our indexing methodology and present in Sec. 4 our system implementation (*ArQSS*). Performance evaluation of *ArQSS* is given in Sec. 5.

2 Related Work

Query processing in data streams was proposed for online aggregation queries by Hellerstein [5]. Data streams were generated by the query processor inside the relational database engine. Formal data models and query semantics of data streams were presented by Babcock et al. [1]. Their data model is focused on homogeneous data streams. In this paper, we are concerned with heterogeneous streams. Several stream processing systems have been implemented (STREAM [6], Aurora [2]) based on the data model and query semantics in [1]. They focus on continuous queries on homogeneous data streams. Our implementation adapts to time-varying heterogeneity of the stream, and optimizes the storage layout and query processing accordingly. Berthold et al. [4] proposed a model to predict quality of service metrics for query performance in data stream processing systems. Their cost model does not apply to our methods. InterMon [3] is a stream analysis system, intended for analysis and mining of the *data value* network traffic data streams, so it cannot be applied to heterogeneous data streams.

3 Indexing Methodology and Query Processing

In this section, we describe the data model and indexing techniques for heterogeneous data stream.

3.1 Data Model and Queries

Data Model: We consider a very simple but time-varying data model. Each reading is a record consisting of *attribute-value* pairs. The attribute names must be unique, and the order of the attributes does not matter. Formally, each reading is a mapping from a finite set of

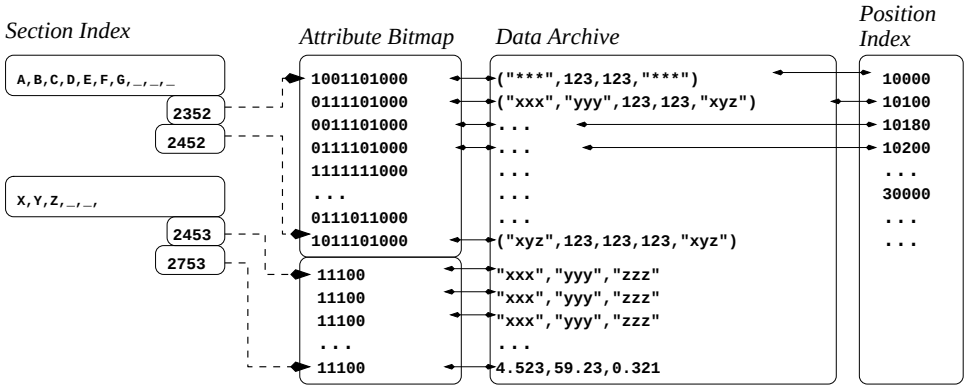


Figure 1. Structure of index and data files

attributes to some value domain, $r : \mathbf{Attr} \rightarrow \mathbf{V}$ where **Attr** is the set of attributes and **V** the set of possible values. A heterogeneous data stream is an infinite sequence of readings: $\langle r_1, r_2, r_3, \cdots \rangle$. The readings $\{r_k\}$ may have distinct attributes.

Example 1. *Consider sensors measuring temperature. Its readings may look like:*

$$\begin{bmatrix} \text{time} & \mapsto & 11:00:00 \\ \text{temperature} & \mapsto & 25^oC \end{bmatrix}$$

Readings from a pressure sensor look like:

$$\begin{bmatrix} \text{time} & \mapsto & 11:00:00 \\ \text{pressure} & \mapsto & 101.12 \end{bmatrix}$$

A base station observing these types of sensors will receive a stream consisting of interleaved readings. □

QUERIES: At the most primitive level, we support basic selection queries involving structural and value predicates. By structural predicates, we mean conditions on the attributes of readings, and similarly, by value predicates, we mean conditions on the values of readings. More complex queries such as *join* and *aggregation* queries can be implemented using the basic selection queries. Our focus is on fast real-time performance of selection queries.

Example 2. *Continue with the previous example. One may use structural queries to separate readings of the two types of sensors using the following structural predicates*[1]*:*

- *WITH ATTRIBUTES*
  INCLUDING (temperature) AND
  NOT INCLUDING (pressure)

- *WITH ATTRIBUTES*
  INCLUDING (pressure) AND
  NOT INCLUDING (temperature)

[1] We omit the formal syntax of the query language. The meanings are clear from the context.

Structural predicates can be combined using any logical connectives. One may also wish to monitor alarming sensor readings by setting a threshold on the temperature reading using the following data value predicate.

- *WITH DATA*
  VALUE-OF(temperature) $> 100^oC$ □

3.2 A Bitmap Index

In order to deal with possibly rapid time variations of reading attributes, we store the data stream into sections with attributes within a section more uniform, and attributes across different sections highly variant.

In order to facilitate efficient query processing, we store the incoming data stream into three index files and a data file. All index and data files are maintained using *append-only* file access, so indexing and archiving can be performed highly efficiently on disk.

The three index files are: *Section index*: It describes the summary of sections which the incoming stream is broken into. In most cases, there are far fewer sections than sensor readings as each section contains many readings. Each section consists of: (1) An array of attributes with possibly empty elements *at the tail* of the array. (2) The start position in the attribute bitmap index. (3) The end position in the attribute bitmap index.

Attribute bitmap: We allocate a bit vector, or bitmap, per sensor reading to describe the attributes present in the reading. Bitmaps in the bitmap file need not be of equal length; though bitmaps belonging to the same section are equal in length. Using the attribute bitmap index, in conjunction with the section index, one can uniquely decode the attributes of readings in the stream.

Data archive: Only data values are kept in the data archive file. This provides efficient storage of readings. Because of the multi-layered index structure, we simply store the data values without any null values. Due to

the heterogeneity of the readings and the difference in data content, entries of the data archive are variable in length.

Position index: Because entries in the data archive are variable length, in order to facilitate efficient fast-forward, we explicitly store the file position of each entry in a position index file. Each entry in the position index is simply an 8-byte long integer which is the position of the corresponding entry in the data archive.

Example 3. *Figure. 1 shows the content of the various files of a sample index. The first section shown contains seven attributes* A,B,C,D,E,F,G*, but at its maximal capacity it could have held nine attributes. Not all slots were used before the system decided to create a new section which contains three attributes* X,Y,Z*. The first reading shown in the figure is encoded as the bitmap* `1001101000` *and data values* `"***", 123, 123, "***"`*. The bitmap can be easily decoded, using the section information, to the attribute present in the reading. In this case, the first, fourth, fifth, and seventh attributes of the section are present. Therefore, the first reading shown in the figure is:*

$$\begin{bmatrix} A & \mapsto & \text{“}***\text{”} \\ D & \mapsto & 123 \\ E & \mapsto & 123 \\ G & \mapsto & \text{“}***\text{”} \end{bmatrix}$$

□

3.3 Archiving Using Indices

In order to obtain high throughput in streaming environment, we must use append-only file access to maintain the index files.

SECTION INDEX: When a new reading r arrives, we first check if the current section (last section of the section index file) can accommodate this reading's attributes. The section *can* accommodate the attributes if: all attributes of r are found in the section attribute array; **or** some attributes of r are not in the section attribute array, but there are enough unused slots to append these attributes to the end of the array. If the attributes of r cannot be accommodated by the current section, then we choose to close the section, and create a new section based on the attributes of the previous last section and attributes of r. This ensures that the last section can always accommodate incoming readings.

BITMAP INDEX: Using the section information, we can encode the attributes of r to a bitmap, which is written to the bitmap index. Some care is required to ensure that the start position and end position of bitmaps belonging to a section are recorded in the section index using append-only file I/O. This entails writing the start position first when a section is created, and writing the end position of the section as the file position of the last bitmap written to the bitmap index when the section is closed.

POSITION INDEX AND DATA ARCHIVE: It is straightforward to update the data archive: simply write the data values according to the order in which the attributes are encoded in the bitmap (from the most significant bit to the least significant bit). The corresponding file position is recorded in the position index.

Performance Tuning: Three tunable parameters are built into the index structure: *extra-bit allocation*, *attribute expiration* and *threading*.

Creating new sections is undesirable due to overhead in file I/O and storage space, as well as penalty on query processing rate. New sections are needed when unknown attributes appear and the current section does not have enough empty slots left. However, all bitmaps of the same section must be of equal length and file I/O is append-only, thus the extra attribute allocation (i.e., extra bits allocated to bitmaps) must be determined at the section creation time. This *extra-bit allocation* is a run-time parameter of our archiving engine.

In order to control the size of the bitmaps, we must identify attributes which have disappeared from the data stream, and thus should be removed by creating a new section. If attributes are removed too eagerly, then many sections will be unnecessarily created. Yet, if stale attributes are kept too long, then bitmaps would be long in length but with few bits set, reducing storage efficiency. To control the attribute removal rate, we have a tuning parameter *attribute expiration* which controls how long, after inactivity, should an attribute be deemed expired.

One can see that our index structure can easily be executed in multi-threaded mode, archiving fast incoming streams using parallel indexing threads, and producing multiple disjoint indices.

3.4 Query Processing

Queries can be efficiently executed using our index structure. To evaluate a structural predicate, we first scan through the section index looking for candidate sections which contain attributes of interest. For each of the candidate sections, we scan through the bitmap index and use the bitmap to very efficiently evaluate the validity of the predicate. Only those readings that satisfy the query predicate are further processed for the purpose of evaluating value predicates or computing aggregations. Because of our index structure, the query processor can skip over entire sections of bitmaps if those sections do not apply to the structural predicate. Furthermore, because of the bitmap index, the query processor can skip over large sections of the data archive whenever possible. Our query processor only uses forward-only file access, thus can be embedded in real-time stream filters [2, 6] or sliding window query operators [1].

4 ArQSS: A System Implementation

In this section, we present our system implementation *ArQSS* for archiving and querying streams. We implemented our system in Java. It is capable of interfacing with sensor servers and RFID edge servers, to receive and index data streams for archiving. *ArQSS* supports multiple sensor inputs and multi-threading. The system architecture is illustrated in Fig. 2. The main components of *ArQSS* are: archiver, performance monitor, adaptive controller and query processor.

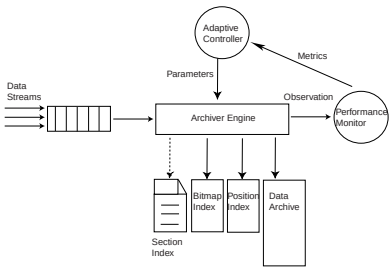


Figure 2. System Architecture

The system runs in multiple threads. The archiver engine runs in one thread. Multiple data streams are input into a queue of data readings that are processed by the *ArQSS* archiver engine, using the bitmap-based indexing method described previously.

The performance monitor runs in a separate thread, thus it is able to asynchronously observe the performance of the archiver. It monitors the *moving average* of the archiving rate, which is a much more accurate measure of the current rate than either instantaneous rate or overall rate (that gives equal weight to distant or recent rates). The moving average is obtained as follows. For each r_i reading archived, the time elapsed for this single reading, t_i, is recorded. The instantaneous rate is $R_i = 1/t_i$ and is placed in a sliding window of the past k archiving times, $\{R_{i-k+1}, R_{i-k+2}, \ldots, R_{i-1}, R_i\}$ The current archiving rate $\bar{R}_{\text{curr}}$ calculated by the performance monitor is the average of the past k rates, $\bar{R}_{\text{curr}} = (R_{i-k+1} + \ldots + R_i)/k$.

Furthermore, the monitor evaluates two performance metrics: *efficiency* and *uniformity*. Efficiency is ratio of the cardinality of the bitmaps (i.e. number of bits set to 1) over the total number of bits. Uniformity is the number of sections in the section index. These metrics are reported to the controller which also runs asynchronously in a separate thread. In our system, the controller is able to send control signals to the archiver engine at run-time to change its performance. Based on the metrics given by the performance monitor, the controller can adaptively control the archiver performance at run-time.

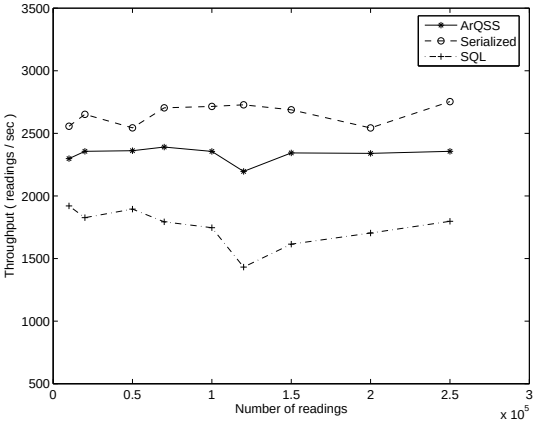


Figure 3. Indexing Throughput

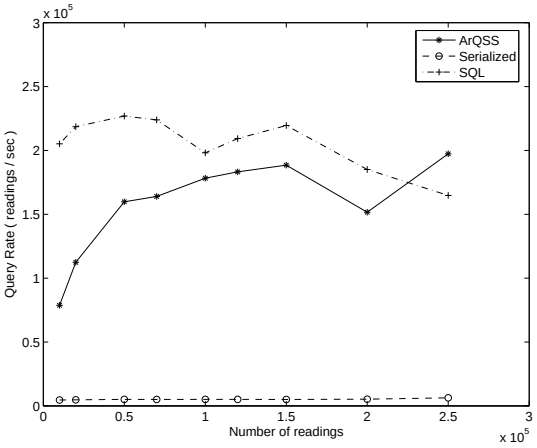
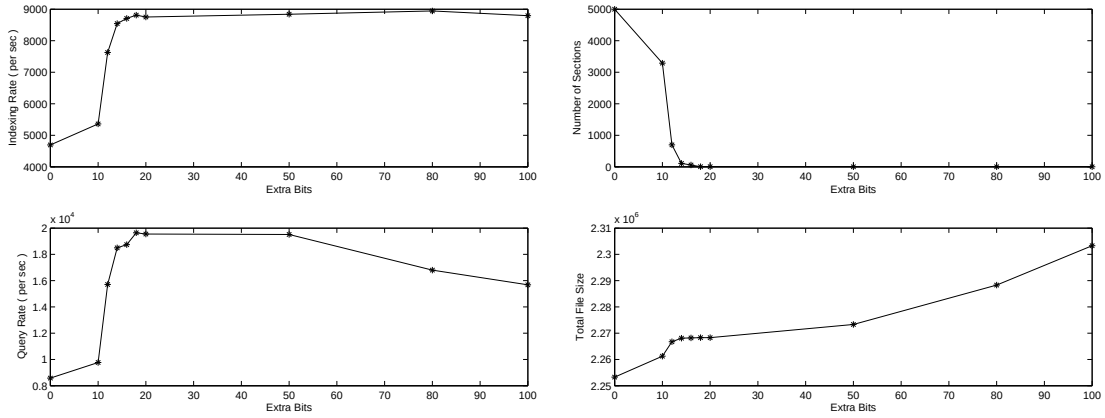


Figure 4. Querying Throughput

5 Performance Evaluation

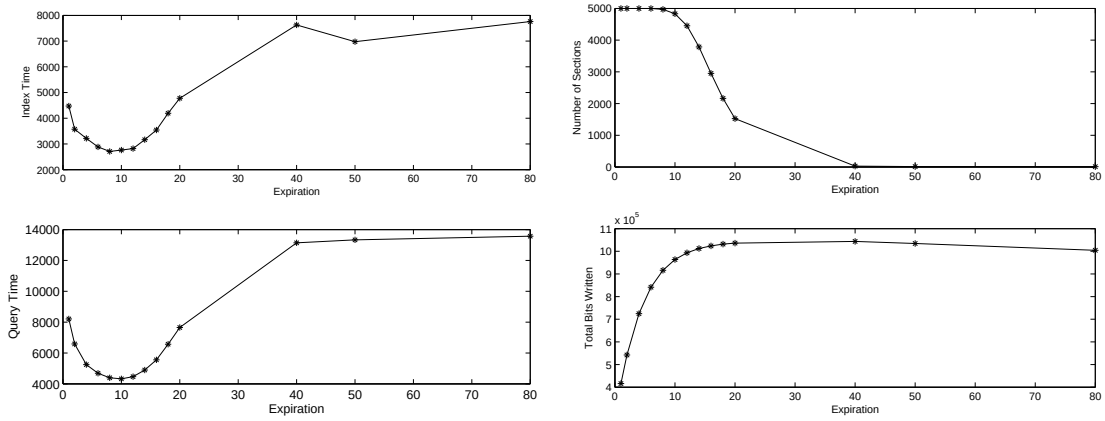
For our evaluation of ArQSS, we generated large numbers of sensors with random attribute profiles. Each sensor generates readings, and the interleaved sensor reading data stream is indexed. We selected two alternative approaches for comparison with ArQSS. One is commercial relational database management system (RDBMS), and the other is complete serialization of each sensor reading into a flat file. To make the relational databases comparable in indexing throughput, we assumed that *all* attributes are known *a priori* to the relational database. Such an assumption is severely limiting in practice. ArQSS as well as the serialization approach do not require such prior knowledge of the attribute names.

In the figures, we refer to the performances of serialization and RDBMS as *Serialized* and *SQL*, respectively. The experiments are performed on an Intel workstation with 2 GB memory and Intel Duo Core 2.80 GHz processor running Linux operating system.



(a) Effect of *extra-bit allocation* on indexing and query processing rates.

(b) Effect of *extra-bit allocation* on number of sections in section index and final index file size.



(c) Effect of *expiration* on indexing and query processing rates.

(d) Effect of *expiration* on sections and bits in bitmap index

Figure 7. Effects of *attribute expiration*

Scalability. First, we evaluate how indexing and query rates change as the number of readings in the sliding window increases. The sensors generate 100 attributes in random continuously. Data values are randomly generated with size ranging from 10 - 100 bytes. The ArQSS indexing engine runs with the default expiration time of 100, and extra bit allocation of 8 bits (or 1 byte). Figure 3 shows the indexing rates of the three techniques with respect to the number of readings in the index. As expected, serialization has the highest throughput, and SQL has the lowest. Our indexing engine, ArQSS, incurs minimal overhead in maintaining additional index files.

Figure 4 shows the query throughput of the three techniques with respect to the number of readings in the index. The observation agrees with our expectation that SQL is the fastest (in most cases), and the full serialization the slowest. Serialization requires inefficient sequential file scan. Observe that the ArQSS rapidly approaches the SQL performance and eventually **exceeds** SQL. We conjecture that SQL functions well due to the large memory buffer, and when the number of kept readings is large, disk I/O starts to degrade the performance. In our case, the forward-only access of bitmaps remains highly efficient regardless of the number of kept readings.

Figure 5 shows disk usage by the three techniques. Our techniques is, by far, the most efficient in disk space. The reason is because that ArQSS does not need to store attribute names of readings multiple times. The attribute names of readings belonging to a common section are stored in the section index only once, and attribute information of each sensor reading is very efficiently encoded as a bit vector.

In conclusion, ArQSS offers excellent performance for both index throughput and query processing, and displays superior efficiency in data storage.

Parallel Indexing. Figure 6 shows indexing throughput with respect to the number of threads. With more than two threads, concurrent ArQSS out-

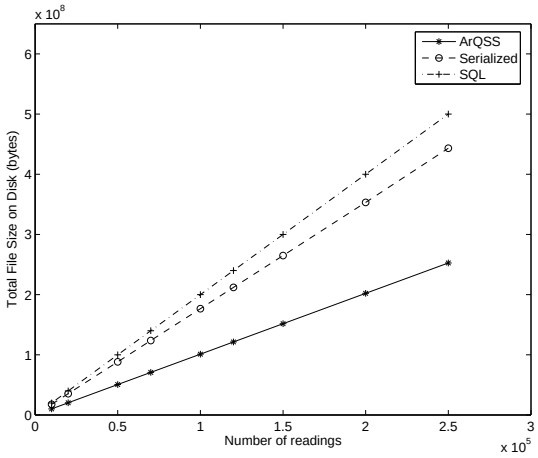


Figure 5. Disk Usage

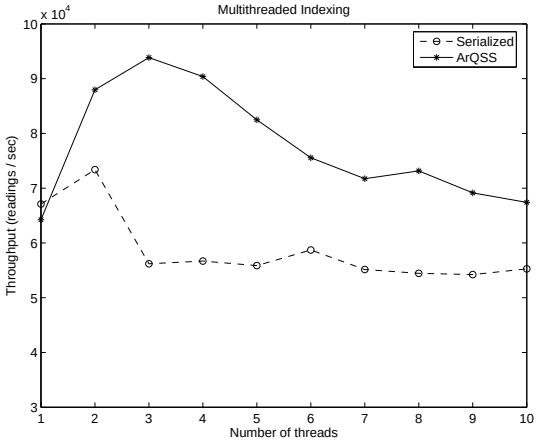


Figure 6. Thread Count

performs concurrent serialization. The reason is that ArQSS is more computationally intensive, thus parallel processing brings greater benefits, whereas serialization is disk I/O intensive.

Performance Tuning of ArQSS. Figure 7(a) shows the effect of extra-bit allocation on indexing throughput and query processing rate. The top figure in Figure 7(a) shows the indexing throughput with respect to varying extra-bit allocation, and the bottom figure shows the query processing rate. For both rates, there is a sharp improvement when the extra-bit allocation exceeds 15 bits. Because with the additional bits, the indexer creates fewer sections and performs faster bitmap encoding. With fewer sections, the query processor can scan through the bitmap index more efficiently using block I/O. However, with too many extra bits, it takes longer for the query processor to process the longer bit vectors.

Figure 7(b) shows the effects of extra-bit allocation on the rate of creation of new sections in the section index, and the size of index files needed for 400,000 sliding window size. With enough extra bits allocated, the number of sections drops rapidly, which improves indexing and querying rate, but the observed increase in index size is small ($\sim 10\%$).

Figure 7(d) and Figure 7(c) show the effects of increase in expiration time on the index files and performance. Not surprisingly, with larger expiration times, the number of sections decrease (see Figure 7(d)). Also with the increase of expiration time, more bits are left unremoved in the bitmap, so there is an increase in the total bits written to the bitmap file.

Indexing throughput and query processing rate are plotted against expiration time in Figure 7(c). Initially, as the expiration time increases, there is a *decrease* in performance – because "dirty" attributes are *not* cleaned up in time, resulting in longer bit vectors. Yet expiration time is not long enough to decrease the number of sections significantly. As expiration time grows large enough, the number of sections drops sufficiently to produce better performance rates.

In conclusion, we demonstrated that ArQSS offers flexible tuning parameters. Those parameters allows one to tune the system to adapt to different application scenarios based on the characteristics of the streams.

6 Conclusion

We addressed the problem of efficient management of high-rate, heterogeneous data streams. We designed an efficient bitmap-based indexing method and a system implementation *ArQSS* of an archiver and query processor. We conducted extensive experimental evaluation and demonstrated that our system exhibited superior overall performance.

References

[1] B. Babcock et al. Models and issues in data stream systems. In *PODS '02*, pages 1–16, 2002.

[2] D.J. Abadi et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[3] E. Hoke et al. Intemon: continuous mining of sensor data in large-scale self-infrastructures. *SIGOPS Oper. Syst. Rev.*, 40(3):38–44, 2006.

[4] H. Berthold et al. Integrated resource management for data stream systems. In *SAC '05: Proc of the 2005 ACM symposium on Applied computing*, pages 555–562, 2005.

[5] J.M. Hellerstein et al. Online aggregation. In *SIGMOD '97*, pages 171–182, New York, NY, USA, 1997. ACM Press.

[6] R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. of First Biennial Conference on Innovative Data Systems Research*, 2003.