Efficient Indexing of Heterogeneous Data Streams with Automatic Performance Tuning

Ken Q. Pu
University of Ontario
Institute of Technology
ken.pu@uoit.ca

Ying Zhu
University of Ontario
Institute of Technology
ying.zhu@uoit.ca

Abstract

We study the problem of indexing continuous data streams in which data are heterogeneous in structure. Such data streams arise naturally in many real-life scenarios such as sensor networks. Our index structure uses bitmap based techniques to efficiently sketch the structures to allow space-efficient lossless archiving of the data stream. It also allows very fast query processing on the archived data stream. Furthermore, our index structure adapts to structural evolutions of the stream to ensure good indexing and querying performance both in space and time. We developed a cost-based optimization framework so the indexing engine adjusts its configuration at run-time to adapt to changes in the data stream. By means of linear feedback controllers, structural clustering and steepest gradient ascent optimization, our indexing engine can achieve excellent performance without any human intervention.

1 Introduction and Motivation

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 [1]. For instance, in sensor networks, sensors transmit streams of monitoring data to the base stations, as depicted in Fig. 1. Typically, each base station receives data from multiple sensors, possibly at high streaming rates. The sensors sharing a base station could differ in the types of data they monitor and send; e.g., there are temperature sensors and pressure sensors, while some sensors might monitor both. Even the same sensor could emit different types of data at different times. In commercial applications where radio frequency identification (RFID) technology is used, items are equipped with

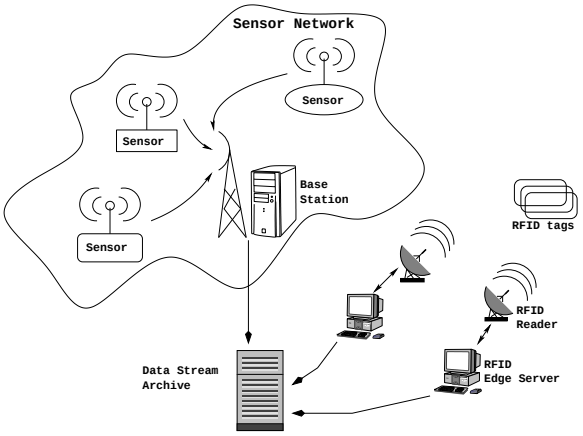


Figure 1. Data streaming scenario

RFID tags containing information of the tagged item. The 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. An inventory item that is an RFID tagged T-shirt has information on its size and color, whereas the tag on a carton of milk would correspond to expiration date and milk fat percentage. Thus, the edge servers will be sending highly heterogeneous data readings to the database. Each reading could contain different data types or attributes. One cannot predict what the structure of the data (i.e., the attributes) is in each reading before it is received and processed. Some systems that manage business data also face the same problems: With point-of-sales (PoS) with annotated sales records, different retail stores may have different record attributes. Heterogeneous record structures are also found in annotated biological databases [8].

In these applications of data streams, there are a number of common elements. (1) There are numerous multiple sources streaming data to the same receiver. (2) The data

could be streaming at high rates, and with multiple sources, the aggregated rate would be even higher. (3) As a result of the previous two properties, there are usually huge accumulations of data readings that must be stored in a relatively short period of time. (4) The data has the potential to be highly heterogeneous; data readings could have different attributes (in varying degrees of differing), regardless of their proximity in the data stream.

We consider not only the problem of efficiently archiving these heterogeneous data streams, but also that of querying them. Queries of interest include filtering for specific attributes in data readings and filtering them based on their values. Common queries are simple in complexity, but because of the huge volume of data streams and queries, the query response time must be minimized.

Thus, a system that manages these infinite heterogeneous data streams must satisfy the following requirements.

- Mechanism for rate control must be in place to deal with data streams that are nearly always *too* fast for any indexing system.
- There must be capacity control because the data streams are infinite while the storage space is finite.
- The structure of the data varies in time (i.e., heterogeneity of data streams), therefore adaptive indexing methods must be developed.
- Data should be efficiently stored.
- Volume can be huge, therefore everything must be append-only; both indexing and query processing must only use forward-only file access.
- Users can query and filter the index streams efficiently.

In this paper, we present a data stream indexing system, ArQSS (*Archiving and Querying Sensor Streams*), which meets all the above requirements. ArQSS uses an adaptive bitmap based index structure to efficiently store incoming data readings into separate indexing files. These files can be be quickly accessed by ad-hoc user queries or stream filters. ArQSS offers a number of tuning parameters so it can be configured to performance optimization and controlled indexing rate. We propose several optimization techniques to automatically configure ArQSS to achieve near-optimal performance, and a user-specified indexing rate. Finally, we apply structural clustering techniques to perform online clustering of the incoming data stream into more homogeneous streams, each of which can be indexed more efficiently by ArQSS.

The rest of this paper is organized as follows. In Section 2, we present the related work. The basic index structure used by ArQSS is presented in Section 3. In Section 4, we discuss how ArQSS can be automatically tuned using objective optimization techniques. We also demonstrate that accurate rate control can be achieved by feedback control using a simple PID (*proportional- integral-derivative*) controller. In Section 5, we apply online clustering to classify the incoming stream into multiple homogeneous streams which are then indexed separately by ArQSS.

2 Related Work

Management and querying of data streams have already received a great deal of attention from researchers in computer networking and database systems. In this section, we discuss the related work.

Query processing of data streams was proposed for online aggregation queries by Hellerstein [11]. 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. [4]. A number of important issues were raised concerning the interpretation of query processing over infinite streams. It was identified that windowing was essential for many classical query operators such as join. Their data model is focused on homogeneous data streams. Therefore, in the scenario depicted in Figure. 1, one needs to explicitly deal with multiple homogeneous streams (one for each *kind* of sensor). They can be *joined*, but the result must also be a homogeneous stream. However, in practice, multiple sensors communicate to a common base station; thus, to the base station, all sensor readings are interleaved into a single heterogeneous data stream. In this paper, we are primarily concerned with processing such heterogeneous data streams, and study the problem of efficient indexing, archiving and real-time query processing.

$$\begin{bmatrix} \mathsf{time} & \mapsto & 11:00:00 \\ \mathsf{temperature} & \mapsto & 25^oC \end{bmatrix}$$
(a) Temperature Sensor

$$\begin{bmatrix} \mathsf{time} & \mapsto & 11:00:00 \\ \mathsf{pressure} & \mapsto & 101.12 \end{bmatrix}$$
(b) Pressure Sensor

$$\begin{bmatrix} \mathsf{time} & \mapsto & 23:33:00 \\ \mathsf{pressure} & \mapsto & 102.12 \\ \mathsf{temperature} & \mapsto & 15^oC \end{bmatrix}$$
(c) Temp. / Pressure Sensor

Figure 3.

Several stream processing systems have been implemented (STREAM [12], Aurora [1]) based on the data

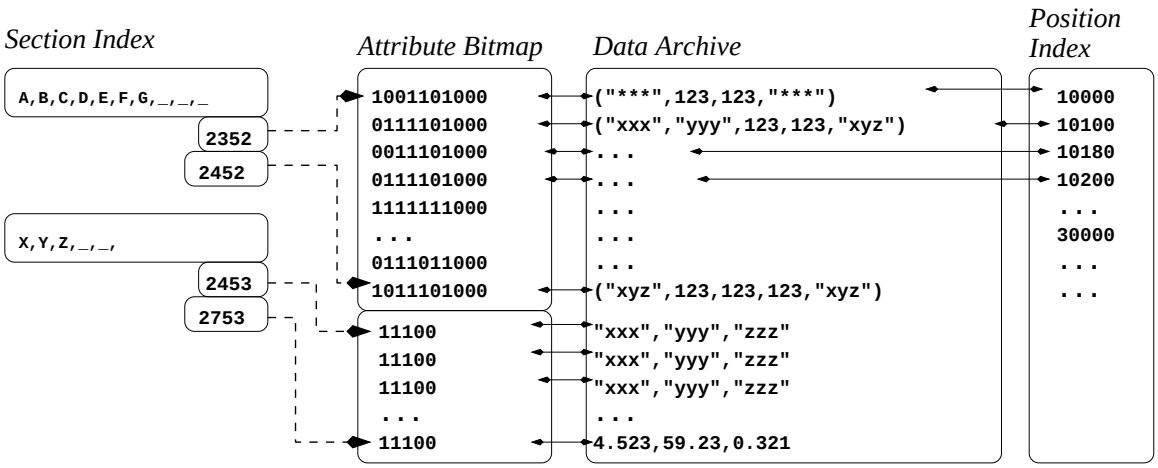


Figure 2. Index files and data archive file

model and query semantics in [4]. They focus on continuous queries on homogeneous data streams. Our implementation, ArQSS (Archiving and Querying System for Streams), focuses on efficient storage and querying of windowed streams. In particular, our system adapts to time-varying heterogeneity of the stream, and optimizes the storage layout and query processing accordingly.

In terms of optimization of stream queries, Berthold et al. [6] proposed a model of query performance in data stream processing systems. The mathematical cost model is used to predict quality-of-service (QoS) metrics. Their cost model does not apply to our indexing and query processing methods.

As part of performance optimization, we apply clustering to the incoming stream (see Section 5). Clustering of streamed data has been addressed by many researchers [5, 9, 3, 2, 7]. Guha et. al. [9] proposed a general framework and several extensions to the k-means clustering [10] to deal with streamed data. Aggarwal et. al. [2, 3] proposed a clustering algorithm, HPStream, for high-dimensional data streams. It uses projected clusters for dimensionality reduction, and fading cluster structures to adapt to time variation. Cao et. al. [7] proposed an online clustering algorithm based on density estimation. It has the advantage that the clusters need not be spatially compact (as is the case of k-means variants), but can take arbitrary shapes. Beringer and Hullermeier [5] proposed an online algorithm for clustering data streams rather than values in the stream.

3 ArQSS: a Queryable Index Structure

In this section, we describe a simple data model of heterogeneous streams, and the bitmap-based index structure for archiving heterogeneous data streams with sliding windows.

3.1 A simple model for heterogeneous streams

A data stream is assumed to be an infinite sequence of time-stamped records. Each record consists of key-value pairs, where the keys are the attributes of the reading, and the values are the corresponding data of the reading.

We are interested in cases where the incoming data stream consists of heterogeneous record structures, namely, the attributes of adjacent records in the stream can be quite different.

Example 1 *In an environmental monitoring application, a base station can receive data from different types of sensors. Some may detect temperature only, and thus only produce records as shown in Figure 3(a), others may detect only pressure, thus produce records shown in Figure 3(b). There could yet be sensors capable of collecting both temperature and pressure readings, as in Figure 3(c). As a result, the data stream assembled by the base station is a sequence of heterogeneous records from those three types of sensors.*

3.2 A bitmap based index structure

As motivated in Section 1, it is often desirable, or even necessary, to archive a sliding window of data streams for the purpose of ad hoc data analysis.

There are two naive approaches to the storage problem:

- Storing each record as a row in a relational table using a relational database management system (RDBMS).
- Serialize each record as an array of key-value pairs and store it in a flat file.

Both have limitations which make them impractical.

In the relational storage approach, one must assume that *all* potential attributes are known *a priori*, so no additional columns need to be created during run-time. Such assumption is severely limiting in its applicability. Furthermore, since each reading in the stream only has some of the attributes, the absent attributes must be padded by *null* values, making the resulting relational table highly sparse, and thus space inefficient. Furthermore, an RDBMS is not designed for handling sliding windows of streams, and when the allocated space is full, removal of out-of-date records is costly.

In the naive serialization approach, one is not required to know all potential attributes *a priori*, nor does one need to store *null* values in the archive. However, it is space inefficient in a different way. Because each record in the stream is serialized independently, all the attribute names of every record are stored in the archive. This is very wasteful if there are long running sequences of repeating record structures. The second serious drawback of the naive serialization is that it cannot be queried efficiently.

Our index structure is a hybrid of the two naive approaches which overcomes their shortcomings while preserving the respective advantages. The basic idea is to try to segment the incoming stream of records into different *sections* such that records in the same section are more homogeneous than records from different sections. Within each section, all attribute names are recorded statically only once as an array of strings $\langle A_1, A_2, A_3, \cdots, A_k \rangle$, where k is the number of distinct attributes found in all records in the section. Attributes of each record r belonging to the section can be encoded as a bitmap $\vec{b}(r) = \langle b_1, b_2, b_3, \cdots, b_k \rangle$ where b_i is set to true if and only if the record contains the attribute A_i. The values of r are encoded as a vector of values $\vec{v}(r)\, \langle v_{i_1}, v_{i_2}, v_{i_n} \rangle$ where n is the number of attributes present in r, and i_j is the offset of the j-th bit which is set in the bit vector $\vec{b}(r)$. Observe that this simple encoding of records:

- does not introduce *null* values, and
- uses only one bit per record for each absent attribute.

In order to facilitate fast query processing, we store the bit vector $\vec{b}(r)$ and the value vector $\vec{v}(r)$ in two separate files, which we refer to as the *bitmap index* and the *data archive*, respectively. Since the value vector $\vec{v}(r)$ varies in length from record to record, we maintain a *position index* in which we store the absolute offset of $\vec{v}(r)$ in the data archive. The length of the bit vector $\vec{b}(r)$ is fixed for all records in the same section. A fourth index file, *section index*, stores the information of the sections which includes: the array of attribute names in the section, the (absolute) bitmap index offset of the bit vector of the first record in the section, and the (absolute) offset of the bit vector of the last record in the section. The relationship of the four index files — section index, bitmap index, data archive and the position index — are shown in Figure 2.

```
update( r )
  r : next record in the stream
  if(current section cannot accomodate r)
  {
    close current section in section index
    create a new section in section index
  }
  add bit vector of r to bitmap index
  add value vector of r to data archive
  add position of position index
```

Figure 4. Update rules for index and data files

3.3 Updating index and data files

As we have stated, the index files should be accessed in append-only mode for the reason of disk I/O efficiency. The basic update rules are outlined in Figure 4. Based on the next reading in the stream and the current section, we decide whether a new section should be created. The decision procedure will be discussed shortly. Once the section index has been updated accordingly, the bit vector and value vector of the record are appended to their respective files. Finally the position index is updated.

Another limiting requirement on indexing data streams is finite disk usage. Since our indices are maintained as append-only, the ArQSS implementation uses circular files for all four index files. The four circular files are synchronized so that the head-of-queue pointers (in all four files) refer to the same record.

Maintaining sections: The incoming stream is segmented into sections in order to reduce the heterogeneity of records within a section. We choose to create a new section for two possible reasons:

- The next record in the stream has attributes which are not part of the current section.
- Certain attributes in the current section have not been seen for many readings, hence should be removed from future consideration.

In order to control the number of sections to be created, we have two performance tuning parameters: *attribute expiration* and *extra bit allocation*. *Attribute expiration* is an integer which dictates when an attribute in a section should be

considered *expired*. When creating a new section, we keep attributes from the previous section which have not expired, as well as the new attributes in the next reading. Additionally, we optionally allocate extra bits for the new section so that it can accommodate some future readings even if those readings contain attributes that are not yet present in the section. (That is, these extra bits do not correspond to any attributes at the creation of the new section, but will be used later for new attributes in future readings.)

Rate control: In addition to controlling disk usage D, the user may wish to specify the window of history, T, to be archived. However, these two specifications are potentially conflicting, because to fully archive all readings in the time period T, the system may well exceed the specified disk usage D. The solution is down-sampling. In order to be fair to all the data sources of the stream, we perform stochastic down-sampling: We flip a biased coin for each reading to independently determine whether it is kept or discarded. The indexing component of ArQSS has a control parameter p which is the bias of the coin toss.

3.4 Query processing

We consider only simple filter queries which filter incoming data stream with structural predicates or value predicates. By structural predicates, we mean conditions on the attribute names. For instance, a user may wish to select readings from temperature sensors by the structural predicate: *"records containing attribute temperature"*. By value predicates, we are referring to conditions on the values of the records, such as *"records with temperature* $> 100^oC$*"*.

Queries can be efficiently processed using the index structure we have presented. To evaluate a structural predicate, we first scan through the section index looking for candidate sections that 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 structural predicate. This determines if the reading satisfies the query predicate. Only those readings 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. This could drastically reduce the time it takes to zoom in to the desired records.

3.5 Performance Evaluation

In this section, we provide some performance comparisons with the naive relational and record serialization approaches. To conduct the comparison, we generated a stream of heterogeneous records. The attributes of each record are generated independently with probability $p = 0.5$. The expected number of attributes for each record is 50. Clearly, RDBMS is not designed for processing streamed data, so in order to improve its archiving performance, we have (i) turned off transaction logs, (ii) statically created all columns needed, and (iii) kept all historic readings, rather than just readings in the sliding window. For record serialization and ArQSS, we allocate 1 GB disk space to the archive files. For ArQSS, we have set *extra bit allocation* to be 5, and *attribute expiration* to be 10. These settings are arbitrary – we discuss how they can be tuned automatically to optimize performance in Section 4. Using the indices, we evaluated randomly generated queries with both structural and value predicates.

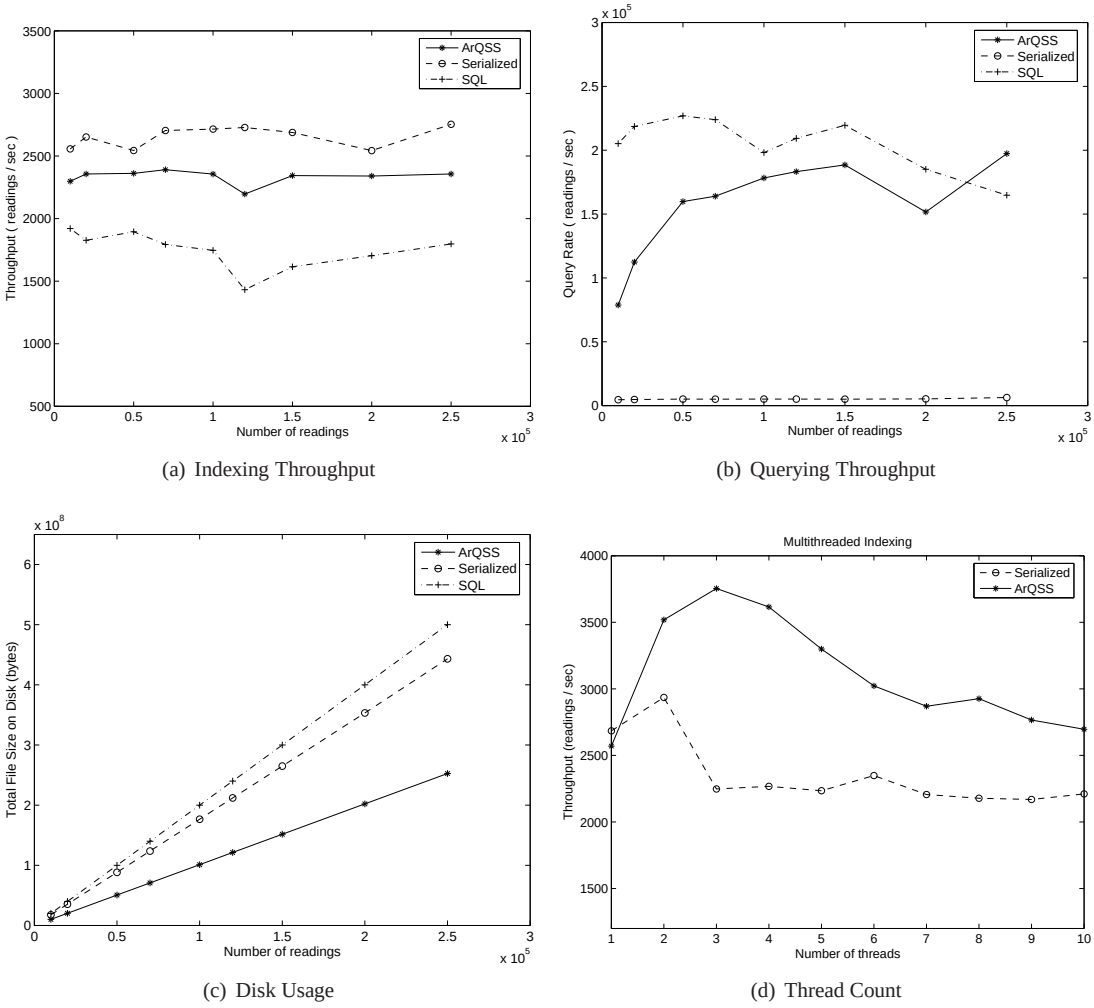
All experiments are conducted on an Intel workstation with 2 GB memory and Pentium Core 2 Duo Processor at 2.80 GHz. We compared the three approaches in terms of:

- Average indexing rate (readings / second)
- Average querying rate (readings / second)
- Growth rate of total disk usage
- Scalability of parallel indexing using multiple threads

In Figure 5(a), we observe that serialization is the fastest at storing the data stream – this is not surprising as it requires *no* pre-processing. Our approach is only marginally slower than serialization, demonstrating that the computational overhead of bitmap encoding is quite insignificant. RDBMS storage is clearly the slowest even when all logging and record removals are disabled. As shown in Figure 5(b), in terms of query processing, serialization is by far the slowest. RDBMS offers good query processing rates using SQL, but our approach is very comparable, and at times even superior. This shows that our approach truly borrows the best from both worlds of serialization and relational storage. In addition, one can see in Figure 5(c) that our approach offers the most compact storage for heterogeneous data streams, by taking as little as half of the space required for the two alternatives. Finally, it is interesting to observe in Figure 5(d) that our approach exhibits good scalability when run in parallel mode. The indexing throughput is at its maximum when running with three threads. We do not include the performance of parallel indexing using RDBMS, because its complex table locking mechanisms lead to very poor performance in multi-threaded mode.

4 Automatic Tuning and Rate Control

In this section, we present several self-tuning mechanisms used by ArQSS to adaptively optimize its performance.



(a) Indexing Throughput

(b) Querying Throughput

(c) Disk Usage

(d) Thread Count

Figure 5. Comparisons of three indexing techniques

4.1 Tuning parameters by objective function optimization

We begin with the selection of the run-time parameters *extra bit allocation* and *attribute expiration*, which will be referred to as extra-bits and expiration, respectively, in the remainder of this paper.

The effects of expiration and extra-bits on indexing and query processing performance are shown in Figure 6(a) and Figure 6(c), respectively. Note that by increasing both parameters, we improve the performance. However, increasing expiration and extra-bits also affects the segmentation of the data stream into sections and the storage efficiency. Figure 6(b) and Figure 6(d) show how the two parameters affect the rate of section creation and the resulting index size. In particular, there is a clear optimal operating point for extra-bits ($20 \sim 40$), for which the performance is ideal and the index size is kept small. According to Figure 6(c) and Figure 6(d), if extra-bits is too large, the index file will grow unnecessarily (i.e., with negligible improvement in indexing performance), causing a significant degradation in query performance.

From the experimental data shown in Figure 6(b) and Figure 6(d), we observe that the final performance is determined by two observable factors: (i) *uniformity*: i.e., the number of new sections created, and (ii) *efficiency*: how many *zero*'s are present in the bitmap index. At the optimal operating point, we wish to maximize both uniformity (small number of sections) and efficiency (small number of zeros in bitmap index). Unfortunately, the two objectives are conflicting. Our approach is to formalize the definition of uniformity and efficiency, and derive a combined objective function which incorporates both factors.

Definition 1 (Observable Performance Measures)

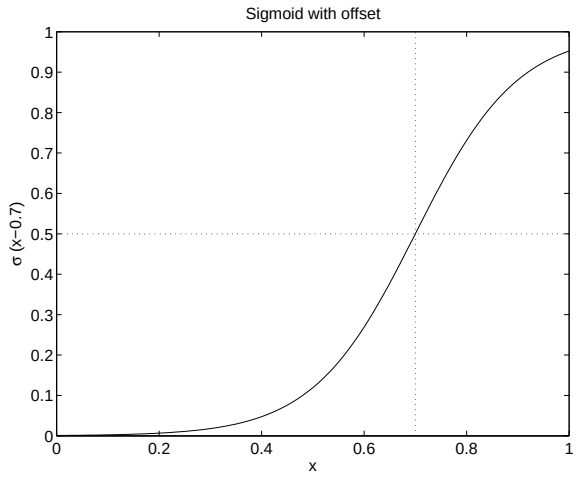


Figure 7. A scaled sigmoid function with offset

Given a time period T, the uniformity of the index is

$$u(T) = \frac{|\textit{sections created in } T|}{|\textit{readings received from stream in } T|}$$

. The efficiency of the bitmap index is

$$e(T) = \frac{|\textit{true bits written in } T|}{|\textit{total bits written in } T|}$$

.

Note that the two measures are bounded by the interval $[0, 1]$.

Since we wish to maximize these two conflicting measures, we need to derive a combined objective function $\eta(T)$ whose maximum corresponds to near-optimal points of both $u(T)$ and $e(T)$. We also wish to parameterize the objective function to be able to favor one measure (say $u(T)$) over the other (say $e(T)$), to varying degrees. One possibility is to take a linear combination: $c \cdot u(T) + (1 - c) \cdot e(T)$, where how much each measure is favored depends on c or $(1 - c)$, respectively. However, our experience shows that due to the sensitivity of the system, any linear combination of $u(T)$ and $e(T)$ would not work well – the tendency is that the system is always dominated by one measure (which one depends on the system), regardless of the setting for c. Another option is to take their product: $u(T) \cdot e(T)$. This does not offer any freedom or control over the degrees of emphasis (or favoring) of each measure.

We choose to combine the two measures by multiplying two scaled sigmoid functions.

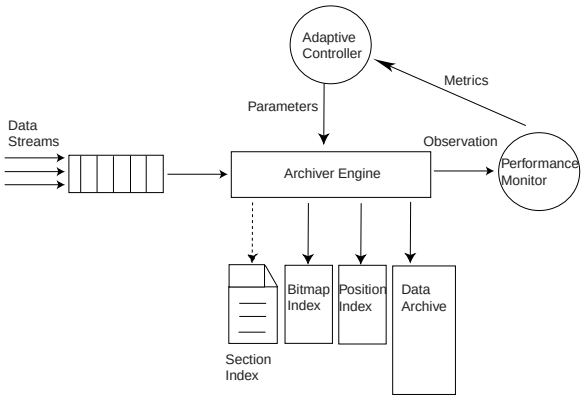


Figure 8. Archiving data streams with feedback performance control

Definition 2 (Combined Objective Function) *Consider a scaled sigmoid function:*

$$\sigma(x) = \frac{1}{1 + \exp(-10x)}$$

Given a time period T, define the combined objective function as:

$$\eta(T) = \sigma(u(T) - c_1) \cdot \sigma(e(T) - c_2)$$

where $0 < c_i < 1$ for $i = 1, 2$.

The scaled sigmoid function $\sigma(x - 0.7)$ is shown in Figure 7.

The two parameters c_1 and c_2 control the emphasis on the performance measures $u(T)$ and $e(T)$, respectively. If c_1 is set to be small (< 0.2), then $\sigma(u(T) - c_1)$ is nearly always close to 1, hence variation of $\eta(T)$ is dominated by $e(T)$.

We simply try to optimize $\eta(T)$ by dynamically varying the run-time parameters expiration and extra-bits , using the steepest gradient ascent method [13]. In order to overcome local optimums, we employ the simulated annealing technique in our optimization. Our run-time optimization architecture is outlined in Figure 8.

Figure 6(e) shows the variation of $u(T)$, $e(T)$ and $\eta(T)$ over the two run-time parameters expiration and extra-bits . In this experiment, we set $c_1 = c_2 = 0.5$, giving $u(T)$ and $e(T)$ equal emphasis. As we can observe empirically, the global maximum of $\eta(T)$ agrees well with the optimal performance point observed in the performance evaluation (Figure 6(a) and Figure 6(c)).

Finally, Figure 6(f) shows the run-time response of our system under automatic performance tuning. The top plot

shows the overall (normalized) indexing rate which gradually improves until it settles. The middle plot shows the rate of sections being created. During optimization, new sections are rapidly created until the system settles to an ideal operating point. The bottom plot shows the resulting objective function during optimization.

4.2 Index rate control by feedback control

In Section 4.1, our emphasis is to automatically tune run-time parameters to achieve optimal indexing and querying rate. However, at times, the maximal indexing rate may be much too high for the allocated disk space to hold all readings in the specified time window. If the readings are arriving too quickly, then the indexer must perform indexing rate control by means of down-sampling, in order to hold samples of all the readings. We adopt a simple uniform random sampling technique: Upon the arrival of each reading, the indexer flips a biased coin and decides whether the reading should be kept or not. Let p be the bias of the coin: if $p = 1$, then all readings are kept, and if $p = 0$, then all reading are discarded. We introduce a run-time feedback control loop which monitors the actual indexing rate, and adjusts p in order to achieve some desirable rate. It is possible to extend this rate control technique by using non-uniform sampling, where samples are selected based on the significance of their attribute and/or value information.

We utilize a standard proportional-integral-derivative (PID) controller to control p. The error signal at time t is defined as:

$$\mathrm{err}(t) = (\text{actual rate at } t) - (\text{desired rate at } t)$$

The control loop computes the value of p using:

$$p(t) = k_1 \cdot \mathrm{err}(t) + k_2 \cdot \int_{t-T}^{t} \mathrm{err}(t)dt + k_3 \cdot \frac{d}{dt}\mathrm{err}(t)$$

corresponding to the proportional, integral and derivative of the error function. Selecting the coefficients k_1, k_2 and k_3 is part of the controller tuning. We find that the setting of $k_1 = 1, k_2 = 0.02; k_3 = 0$ provides good tracking response, and is robust to a wide range of scenarios. We do not include the derivative term ($k_3 = 0$) because it is made particularly unreliable by the inherent noise in the error signal.

Figure 9 shows the time response of the index rate under control. The top plot shows the normalized index rate and the desired control rate over time. Observe that the controller decreases the value of p (shown at the bottom) when the actual rate exceeds the control rate. The system stabilizes to a steady state after only 10 milliseconds, afterwards, the observed rate closely tracks the control rate.

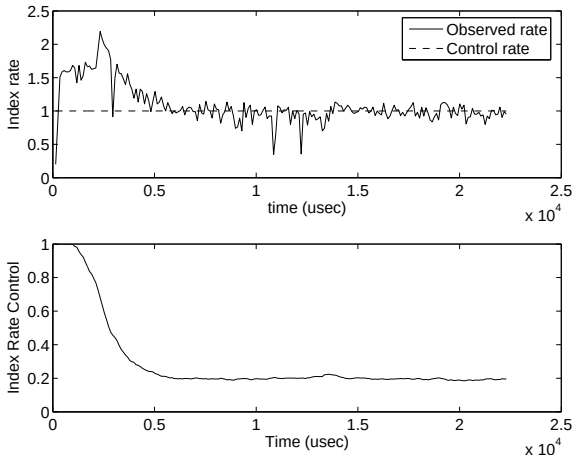


Figure 9. Feedback rate control

5 Structural Clustering

In the previous section, we observed that reducing both the number of sections (i.e., uniformity) and the number of zero bits (i.e., efficiency) leads to improvement in the overall performance (of both indexing and query processing). In this section, we propose to apply a simple online clustering algorithm to group the incoming data stream into multiple sub-streams each of which is more structurally homogeneous. By grouping similarly structured data records into the same section, we can keep the number of sections small without sacrificing efficiency — in fact, efficiency is increased (number of zero bits reduced) at the same time. The online clustering algorithm is similar to online algorithms proposed by Guha et. al. [9] and Aggarwal et. al. [2, 3]. Density based cluster algorithms [7] and clustering of streams [5] do not apply to our scenario since they cannot be used to group individual readings based on their structural similarity.

The main idea is to sample enough readings, r, from the stream and perform k-means clustering [10] according to their bit vector $\vec{b}(r)$ using the Hamming distance measure [10]. In order to maintain high indexing rate, the samples are collected and clustered by the k-means clustering in a separate *cluster advisor thread*. The main indexing thread continues until the cluster advisor thread detects a strong presence of multiple clusters. Once the clusters are detected, the advisor thread interrupts the main indexing thread which creates multiple indices, one for each cluster. Future readings are classified into nearest clusters and are indexed into the respective stream index. The cluster advisor thread continuously samples incoming readings to detect changes in the cluster structure. If significant changes are detected, then the main index thread is interrupted again,

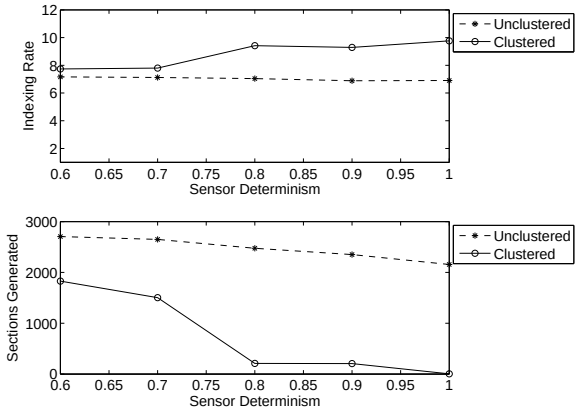


Figure 10. Indexing by clustering

and the index centroids (of the clusters) are updated.

To evaluate the performance gain from clustering, we generate a data stream which is a random interleaving of five simulated sensor outputs. In order to experiment with the presence of noise, the attributes of each sensor stream vary with probability $(1 - d)$, where $d \in [0.5, 1]$ is referred to as the sensor determinism. For $d = 0.5$, sensors cannot be distinguished statistically, and for $d = 1$, the sensors can be easily distinguished statistically.

Figure 10 shows the index performance for different values of sensor determinism. The top plot shows that there is as much as 20% performance gain in achievable indexing rate if the sensors can be statistically distinguished $(d \geq 0.8)$. The bottom plot shows that the number of sections are significantly reduced by means of structural clustering.

6 Conclusion and Future Work

In conclusion, we studied the problem of indexing and storing continuous data streams with heterogeneous structures that vary both over time and across different data sources. We developed a bitmap-based indexer that archives heterogeneous data and is both time- and space-efficient, while affording fast querying of the archived data. Our index structure is highly adaptive to dynamics of structures of data in the stream, to continually and consistently offer efficient storage and high rates of indexing and querying. Based on the cost-based optimization framework we developed, and by using mechanisms of feedback control and online structural clustering, our indexing engine is able to automatically make adjustments of system parameters to adaptively optimize performance. We implemented our indexing and archiving system, and through extensive experiments, demonstrated its superior performance in indexing/querying rates and storage efficiency.

There remains many unexplored issues we plan to investigate in the future. We briefly mention two here as future work. Instead of uniform sampling of the data in indexing rate control, we will develop non-uniform sampling methods based on attribute and value information of the data records. Another interesting problem is the indexing and storing of distributed data archives over a network.

References

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

[2] C. Aggarwal, J. Han, and P. S. Yu J. Wang. A framework for clustering evolving data streams. In *VLDB'03*, pages 81–92, 2003.

[3] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for projected clustering of high dimensional data streams. In *VLDB'04*, pages 852–863, 2004.

[4] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS '02*, pages 1–16, 2002.

[5] Jurgen Beringer and Eyke Hullermeier. Online clustering of parallel data streams. *Data Knowl. Eng.*, 58(2):180–204, 2006.

[6] Henrike Berthold, Sven Schmidt, Wolfgang Lehner, and Claude-Joachim Hamann. Integrated resource management for data stream systems. In *SAC '05*, pages 555–562, 2005.

[7] Cao F., Ester M., Qian W., and Zhou A. Density-based clustering over an evolving data stream with noise. In *SDM'06*, pages 326 – 337, 2006.

[8] Floris Geerts, Anastasios Kementsietsidis, and Diego Milano. mondrian: A visual tool to annotate and query scientific databases. In *EDBT'06*, pages 1168–1171, 2006.

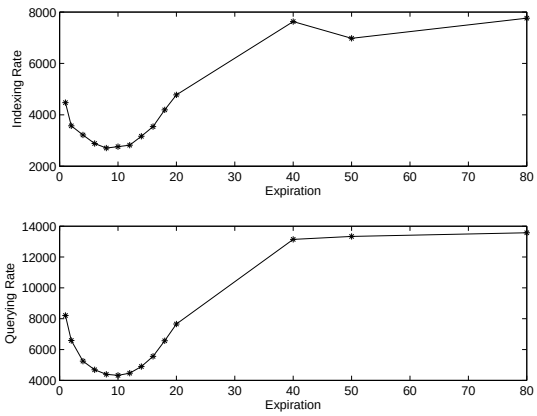
[9] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE TKDE*, 15(3):515 – 528, 2003.

[10] Jiawei Han and 2nd Ed. Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2006.

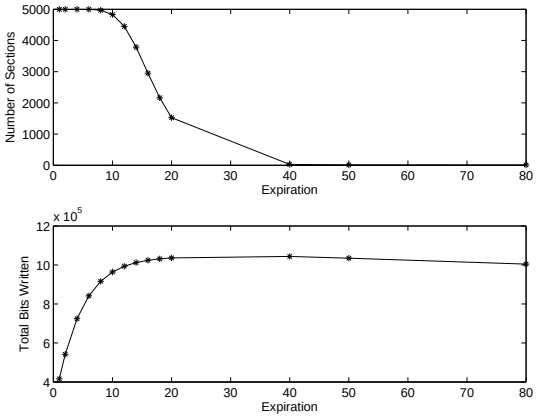
[11] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *SIGMOD '97*, pages 171–182, New York, NY, USA, 1997. ACM Press.

[12] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, , and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of First Biennial Conference on Innovative Data Systems Research*, 2003.

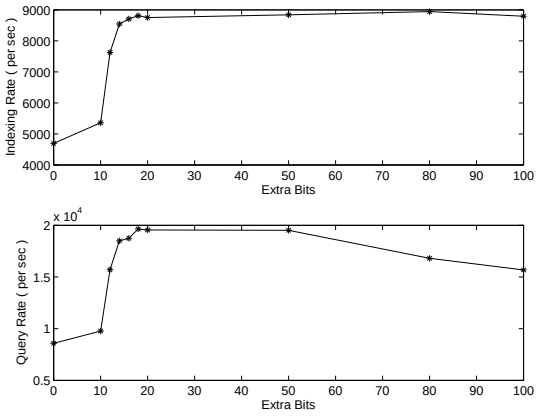
[13] Jan A. Snyman. *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*. Springer Publishing, 2005.



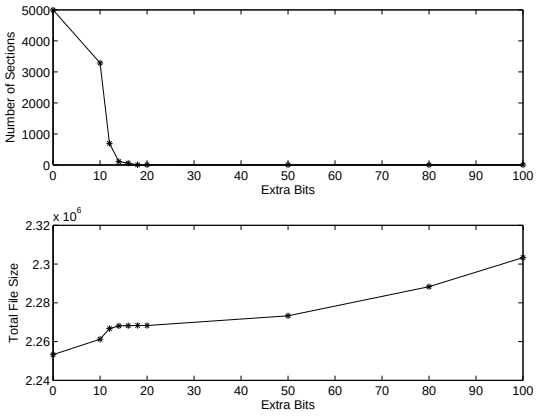
(a) Effects of expiration on performance



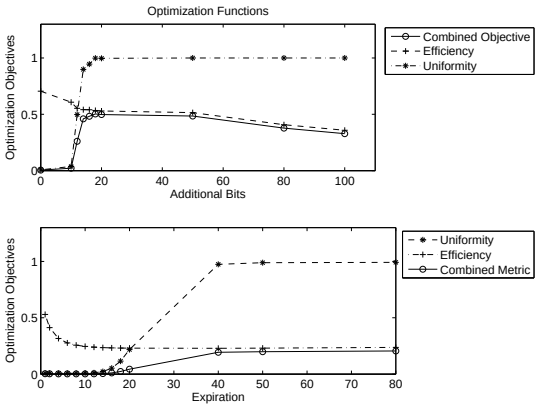
(b) Effects of expiration on sections and bitmap indices



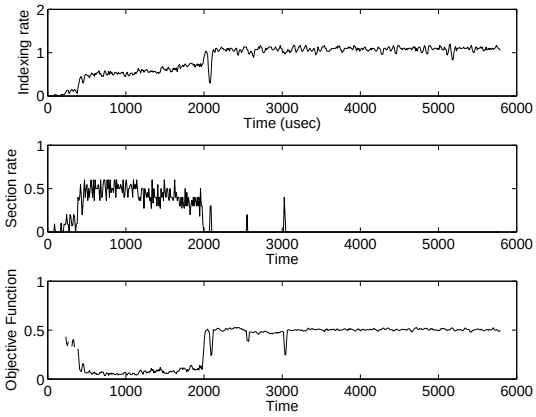
(c) Effects of extra-bits on performance



(d) Effects of extra-bits on sections and index size



(e) Performance objective over expiration and extra-bits



(f) Performance tuning by objective optimization

Figure 6. Performance Evaluation and Optimization