Approximate Processing of Massive Continuous Quantile Queries over High-Speed Data Streams

Xuemin Lin, Jian Xu, Qing Zhang, Hongjun Lu, Jeffrey Xu Yu, *Member*, *IEEE Computer Society*, Xiaofang Zhou, and Yidong Yuan

Abstract—Quantile computation has many applications including data mining and financial data analysis. It has been shown that an ϵ-approximate summary can be maintained so that, given a quantile query (ϕ, ϵ), the data item at rank $\lceil \phi N \rceil$ may be approximately obtained within the rank error precision ϵN over all N data items in a data stream or in a sliding window. However, scalable online processing of massive continuous quantile queries with different ϕ and ϵ poses a new challenge because the summary is continuously updated with new arrivals of data items. In this paper, first we aim to dramatically reduce the number of distinct query results by grouping a set of different queries into a cluster so that they can be processed virtually as a single query while the precision requirements from users can be retained. Second, we aim to minimize the total query processing costs. Efficient algorithms are developed to minimize the total number of times for reprocessing clusters and to produce the minimum number of clusters, respectively. The techniques are extended to maintain near-optimal clustering when queries are registered and removed in an arbitrary fashion against whole data streams or sliding windows. In addition to theoretical analysis, our performance study indicates that the proposed techniques are indeed scalable with respect to the number of input queries as well as the number of items and the item arrival rate in a data stream.

Index Terms—Query processing, online computation, data mining.

1 Introduction

Continuous queries are issued once and run continuously to update query results along with updates of the underlying data sets. Research in efficient query processing over data streams has recently received a great deal of attention and many techniques have been developed, such as processing of relational queries [2], [10], [14], [16], [29], [34], semistructured data and Web information [3], [11], [15], [21], sensor monitoring [22], data mining and clustering [4], [7], [24], [35], etc. In the area of continuous queries over data streams, the following conflicting goals are often involved: 1) real-time processing, 2) system scalability, and 3) continuous updates of the query results. To resolve this, the two general paradigms, *load sharing* and *adaptivity* [2], [17] of execution plans, have been developed.

Among continuous queries, the *quantile* computation is one of the most challenging because of the order complexity and the holistic nature. A ϕ-quantile $(\phi \in (0, 1])$ of an ordered data sequence with N elements is the element with rank $\lceil \phi N \rceil$. It has been shown that an ϵ-approximate summary can be continuously maintained with the space requirement $O(\frac{1}{\epsilon}\log(\epsilon N))$ [13] for a whole data stream. Our earlier work [19] shows that it requires space $O(\frac{\log(\epsilon^2 N)}{\epsilon} + \frac{1}{\epsilon^2})$ for a sliding window; the space bound has been improved to $O(\frac{1}{\epsilon}\log\frac{1}{\epsilon}\log N)$ in [1]. Randomized techniques for continuously maintaining ϵ-approximate summary may be found in [8], [12], [26]. A quantile summary is ϵ-approximate if it can be used to answer any quantile query within a rank error precision of ϵN (to be defined precisely in the next section).

Computing quantiles has many applications, including data summarization via *equal-depth histograms* [30], *data cleaning* [9], and *data partitioning* [31]. Quantile computation is also a key in the decision making (e.g., *portfolio* risk measurement in the stock market [20], [23]). In some applications, there may be a massive number of continuous quantile queries issued simultaneously. Below is such an example.

Example 1. An information provider may provide various real-time statistics of the stock market to its clients, through the Internet or telecommunication, for trends' analysis. The *quantile-quantile* (Q-Q) plot [33] is a popular chart for comparing two data distributions. Fig. 1 illustrates such a Q-Q plot by using two real data sets,

- *X. Lin and Y. Yuan are with NICTA and the School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia. E-mail: {lxue, yyidong}@cse.unsw.edu.au.*
- *J. Xu is with the Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver, B.C. V6T 1Z4. E-mail: xujian@cs.ubc.ca.*
- *Q. Zhang is with the e-Health Research Centre/CSIRO ICT Centre, Lvl 20, 300 Adelaide St., Brisbane, Qld 4000, Australia. E-mail: qing.zhang@csiro.au.*
- *H. Lu was with the Department of Computer Science, Hong Kong University of Science and Technology, China.*
- *J.X. Yu is with the Department of Systems Engineering and Engineering Management, the Chinese University of Hong Kong, Shatin, New Territories, Hong Kong. E-mail: yu@se.cuhk.edu.hk.*
- *X. Zhou is with the School of Information Technology and Electrical Engineering, the University of Queensland, Brisbane, QLD 4072, Australia. E-mail: zxf@itee.uq.edu.au.*

Manuscript received 24 Jan. 2005; revised 21 July 2005; accepted 1 Nov. 2005; published online 17 Mar. 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0035-0105.

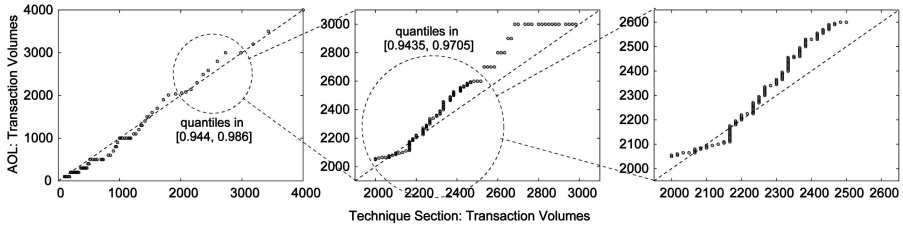

Fig. 1. A quantile-quantile (Q-Q) plot.

AOL and Technique Section. In AOL (TWX since 2003), 1.3M (millions) "tick-by-tick" transactions during the period of Dec./2000 to July/2001 sorted increasingly against the volume of each transaction (deal) are collected from the New York Stock Exchange for the stock AOL. In Technique_section (made by us), 27M tick-by-tick transactions are collected in the same period for the stocks CSCO, IBM, DELL, SUN, CA, etc., and also sorted on volumes. Fig. 1 demonstrates that clients can view the global chart (with very coarse information) for a general comparison and can also click on such a chart graph to zoom in a particular range of quantiles interactively for more accurate information. Such Q-Q plots combining with other statistic display tools greatly facilitate clients' detection of trade trends, so that they may make good trade decisions based on their knowledge.

In such applications, a large number of clients, such as day-traders, may continuously use the service to monitor a particular stock and may focus on different quantile ranges against this particular stock, especially when it is used to compare one-by-one with many other individual stocks. Therefore, the information provider has to support continuous processing of quantile queries across all quantile ranges. With real-time requirements, users may tolerate a bounded amount of imprecision for online information, and may register approximate quantile queries at a central stream processor from their personal devices (e.g., mobiles).

Olston et al. [29] studied the problem of continuously monitoring statistics, based on aggregates, over distributed data streams with the focus on minimizing communication overheads. Due to the nature of the applications, scalability against the number of queries is not an issue. Moreover, querying order statistics seems more complicated than those aggregates. Chen et al. [3] addressed scalability against the number of queries by grouping continuous queries for efficient evaluation. However, they focused on XML queries. Thus, the techniques are not relevant to continuously querying order statistics.

Motivated by the above, in this paper, we investigated the problem of online processing of massive continuous quantile queries over data streams. Unlike simply continuously maintaining a summary over data streams, the central focus of the investigation is to continuously provide up-to-date query results in real time to a large number of queries that may be registered and removed in an arbitrary fashion.

In the context of data stream computation, a key requirement is to take little processing time per data element. The main research focus in the area has been so far focused on effectively selecting sample data to be stored for approximate processing. Little has been found in the literature to handle a massive number of continuous queries. Consider that there are Γ continuous queries. Due to the nature of continuous queries, reevaluation of each query result may be necessary upon the arrival of a new data element. This leads to the system time complexity $\Omega(\Gamma)$ per data element even if processing one query takes constant time. Clearly, such a time complexity is not acceptable for online processing when Γ is large. On the other hand, in some applications (e.g., the stock market application mentioned above), the number Γ of sensible and distinct quantile queries could equal the number of data elements in a massive stream; thus, it could be huge. To resolve this, we propose to first reduce Γ by grouping "similar" queries together and treat them virtually as a single query. Then, we develop a novel trigger-based lazy update query processing technique to avoid reevaluating every query per new data element while the query result precisions can be retained.

To the best of our knowledge, no similar research results have been reported in the literature. Our main contribution can be summarized below:

- We propose a novel query clustering technique, such that each cluster of queries can be viewed as a single query while the required precisions can be retained.
- For a given set of queries, we develop efficient algorithms to cluster queries, so that the total number of times for reevaluating clusters may be minimized or the total number of clusters to be minimized. The techniques are extended to maintaining query clustering effectively when queries are dropped and inserted in an arbitrary fashion.
- We propose a novel trigger-based asynchronous query result update technique to efficiently process continuously quantile queries, including data structures and maintenance algorithms. Our techniques guarantee that a cluster may be reevaluated only logarithmic times, with respect to the number of data elements, in the worst case.
- We apply the techniques to both whole data streams and sliding windows.

The rest of this paper is organized as follows: In Section 2, we present some background information in quantile computation and continuous quantile queries. Section 3 presents our query clustering techniques and trigger-based lazy update techniques for a whole stream. Section 4 discusses an application of the techniques to sliding

windows. In Section 5, we report the results of our performance study. Section 6 concludes the paper.

2 Background Information

In this section, we present some background information about quantile, quantile summary, summary maintenance techniques, and continuous quantile queries for data streams.

2.1 Quantile and Quantile Summary

A *ϕ-quantile* ($\phi \in (0,1]$) of a sequence D of N data elements is the element with rank $\lceil \phi N \rceil$ in an ordered sequence of D according to some search key. For notation simplification, we always assume that a data element is a value and the ordered sequence follows an increasing order of the data values. We use N to denote the number of data elements.

It has been shown that any algorithm for computing exact ϕ-quantiles of a sequence of N data elements requires a space of $\Omega(N^{1/p})$ if only p scans of the data set are allowed [27]. This makes it impractical to compute exact quantiles for very large sequences for which multiple scans are very costly (e.g., very large databases), even infeasible (e.g., data streams). On the other hand, in most applications, an estimation of quantile is indeed sufficient. In such case, we can maintain certain computed quantile summary for a sequence so that quantiles can be estimated with a minor computational effort.

Generally, a quantile summary may be in any form. One form of quantile summary (abbreviated to "summary" hereafter) proposed for approximate query processing is defined as follows [13], [19]:

Definition 1 (Summary). *A* quantile summary *S of an ordered data sequence $\overrightarrow{D}$ is defined as an ordered sequence of tuples $\{(v_i, r_i^-, r_i^+) : 1 \le i \le m\}$ with the following properties:*

1. *Each $v_i \in \overrightarrow{D}$.*
2. *$v_i \le v_{i+1}$ for $1 \le i \le m-1$.*
3. *$r_i^- \le r_{i+1}^-$ and $r_i^+ \le r_{i+1}^+$ for $1 \le i \le m-1$.*
4. *For $1 \le i \le m$, $r_i^- \le r_i \le r_i^+$, where r_i is the rank of v_i in $\overrightarrow{D}$.*[1]

Clearly, a summary with more details gives a more precise estimation, but at the cost of space and time. To measure and control the estimation error of a summary, an *ϵ-approximate summary* was defined.

Definition 2 (*ϵ-approximate*). *A summary for a data sequence of N elements is* ϵ-approximate *if, for any ϕ ($\phi \in (0,1]$), it returns a value whose rank r' is guaranteed to be within the interval $[\phi N - \epsilon N, \phi N + \epsilon N]$.*

To guide the process of maintaining an *ϵ-approximate* summary, the following theorem has been proven in [13]:

Theorem 1. *For a summary S defined in Definition 1, if:*

1. *$r_1^+ \le \epsilon N + 1$,*
2. *$r_m^- \ge (1-\epsilon)N$, and*
3. *for $2 \le i \le m$, $r_i^+ \le r_{i-1}^- + 2\epsilon N$,*

then, for each $\phi \in (0,1]$, there is a (v_i, r_i^-, r_i^+) in S such that $\phi N - \epsilon N \le r_i^- \le r_i^+ \le \phi N + \epsilon N$; that is, S is ϵ-approximate.

In the original version [13] of the definition of ϵ-approximate and Theorem 1, the term $\lceil \phi N \rceil$ was used instead of ϕN. To simplify the mathematic notation, in this paper, we use ϕN. It can be immediately verified that Theorem 1 holds for the term ϕN. In this paper, we assume that an ϵ-approximate summary is maintained to satisfy the conditions specified in Theorem 1.

2.2 Quantile Queries

Given an ϵ-approximate summary, we can issue "approximate quantile" queries to obtain estimates within a required precision. In this paper, we study only approximate quantile queries, abbreviated to quantile queries hereafter.

Definition 3 (Quantile Query). *A quantile query q over an ordered data sequence $\overrightarrow{D}$ is represented as $q = (\phi_q, \epsilon_q)$. It requires an element u from $\overrightarrow{D}$ such that the rank r of u over $\overrightarrow{D}$ has the property that $|\phi_q N - r| \le \epsilon_q N$, where $|\overrightarrow{D}| = N$.*

From Theorem 1, it is immediate that, for a quantile query $q = (\phi_q, \epsilon_q)$ and an ϵ-approximate summary S where $\epsilon \le \epsilon_q$, we are always able to find a tuple (v_i, r_i^-, r_i^+) from S such that

$$\phi_q N - \epsilon_q N \le r_i^- \le r_i^+ \le \phi_q N + \epsilon_q N. \quad (1)$$

Consequently, v_i can be used as a result of q; such a tuple is referred as a *quantile query result tuple*. In other words, for a quantile query $q = (\phi_q, \epsilon_q)$, we can always query an ϵ-approximate ($\epsilon \le \epsilon_q$) summary instead of the data sequence itself.

Given an ϵ-approximate summary, a result tuple of a quantile query $q = (\phi_q, \epsilon_q)$ can be obtained using the *first-hit* approach [13], [19]. That is, we sequentially scan the summary until we find a tuple (v_i, r_i^-, r_i^+), such that the condition in (1) holds.

2.3 Maintaining ϵ-Approximate Summaries

For data streams, a virtually unbounded number of elements arrive rapidly and multiple scans over arrived elements are computationally infeasible. Therefore, maintaining a small space summary is the only feasible way to obtain estimates about quantile statistics for data streams. Recently, algorithms for efficiently maintaining quantile summaries for data streams under different computational models have been proposed [13], [19]: 1) Under the *data stream model*, a summary is maintained over all elements seen so far. 2) Under the *sliding window model*, a summary is maintained over the most recently seen N elements where N is predefined. 3) Under the *n-of-N model*, a summary for the most recent N elements is maintained in such a way that quantile can be estimated over any most recent n elements with $n \le N$.

In this paper, we will consider only two models, the data stream model and the sliding window model.

1. In case there are duplicated values, a rank r_i is not well defined; however, in the paper, we assume that such a rank is given in the ordered sequence and maintained thereafter.

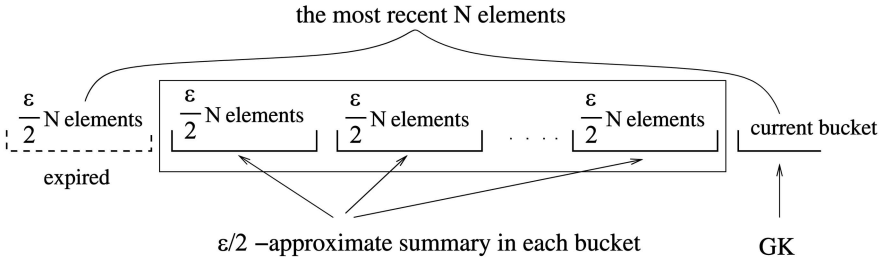

Fig. 2. Sliding window technique.

2.3.1 Whole Streams

A number of algorithms have been proposed in the literature to continuously maintain quantile summaries for data streams with different complexities and space requirements [8], [12], [13], [25], [26]. While [8], [12], [26] investigate randomized techniques, in our study, we focus on deterministic techniques for maintaining an ϵ-approximate quantile summary, $\{(v_i, r_i^-, r_i^+) : 1 \leq i \leq m\}$, under the data stream model. Specifically, we focus on GK-algorithm, proposed by Greenwald and Khanna [13], which achieves the best space bound among deterministic techniques and removes the requirement of preknowledge about the stream length [25]. For each tuple in the quantile summary, v_i is one of the data values in the data stream seen so far with v_1 and v_n being the first and the last element in the ordered data stream, respectively. The algorithm uses two parameters, g_i and Δ_i, to control r_i^- and r_i^+, where $g_i = r_i^- - r_{i-1}^-$ ($r_0^- = 0$), $\Delta_i = r_i^+ - r_i^-$, and $r_i^- = \sum_{j \leq i} g_j$. It is proven that if, for $2 \leq i \leq m$, $g_i + \Delta_i < 2\epsilon N$, then the summary is ϵ-approximate.

To continuously maintain an ϵ-approximate summary with bounded space for a data stream, the GK-algorithm carries two operations, *insertion* and *merge*, to update the summary when a new data element with value v_{new} arrives. The original version of the GK-algorithm was presented based on g_i and Δ_i. In order to apply the technique to our investigation of processing continuous quantile queries, we transform the presentation in terms of r_i^- and r_i^+.

Insertion. Scan the tuples in the summary S from the tail backwards and increase each r^- and r^+, respectively, by 1 for the tuples scanned, until a tuple (v_i, r_i^-, r_i^+) where $v_i < v_{new}$ is found or the head is reached. A new tuple $(v_{new}, r_i^- + 1, r_{i+1}^+ - 1)$ is inserted into the summary just after (v_i, r_i^-, r_i^+) in S.

Merge. For every batch of $\frac{1}{2\epsilon}$ new data elements, the algorithm scans the summary from the tail backwards. If tuples $t_{i-j}, \ldots, t_{i-1}$, t_i satisfy the conditions that $r_i^+ - r_{i-j-1}^- \leq 2\epsilon N$, and $t_{i-j}, \ldots, t_{i-1}$ all arrive in "recent" batches, then the algorithm removes the tuples from t_{i-j} to t_{i-1} as if they were merged into t_i.

2.3.2 Sliding Windows

Quantile statistics under the sliding window model (i.e., for the most recently seen N data elements) are different from that for a whole data stream.

In our study, we maintain an ϵ-approximate summary for a data stream under the sliding window model using the algorithms developed earlier by us [19]. Using this method, stream data against a sliding window with the window size N is continuously divided into the buckets, as depicted in Fig. 2, according to the arrival ordering of data elements, such that:

- Expire the first (left most in Fig. 2) bucket once the last bucket (right most) is full.
- Run the GK-algorithm in the first bucket to maintain an $\frac{\epsilon}{4}$-approximate local summary.
- The local summaries over the other buckets, as bounded by the big box, can be compressed into $\frac{\epsilon}{2}$-approximate summaries, respectively. Then, they can be merged by a merging technique in [19] into an ϵ-approximate summary over the sliding window.

Note that the interested readers may refer to [1] for another sliding window technique that theoretically improves the space bound in [19]. Since the main issue in processing continuous queries is the minimization of the summary space bound, we still adopt our summary construction algorithm in [19] in this paper.

2.4 Problem Statement and Challenges

Problem Statement. Suppose that an ϵ-approximate quantile sketch is continuously maintained over a data stream. We want to efficiently process continuous quantile queries (ϕ_q, ϵ_q) issued (possibly in an ad hoc fashion) by users, so that users can get up-to-date results of their queries. In such a system, we assume that all queries are processed at the central site, while query sites (e.g., mobiles) can only receive results but are not able to process any query. We always assume that the precision ϵ_q of a quantile query $q = (\phi_q, \epsilon_q)$ is lower than the precision ϵ of a quantile summary (i.e., $\epsilon_q \geq \epsilon$); if $\epsilon_q < \epsilon$, the system increases ϵ_q to ϵ for users while processing the query. Note that the precision ϵ (or ϵ_q) here means its actual value rather than the number of decimals.

With an ϵ-approximate underlying summary continuously maintained over a data stream, continuous quantile queries against the data stream can be processed using the summary. A summary maintained for a data stream is updated continuously; thus, the result tuples for those queries may change accordingly. While the techniques (e.g., first-hit algorithm [13]) for processing an individual quantile query seem rather straightforward, online processing of a massive number of continuous quantile queries over a data stream is a computationally intensive task.

For verification, we conducted an experimental study with a quantile summary maintained for 1 million data elements. Assuming that quantile queries are pregiven,

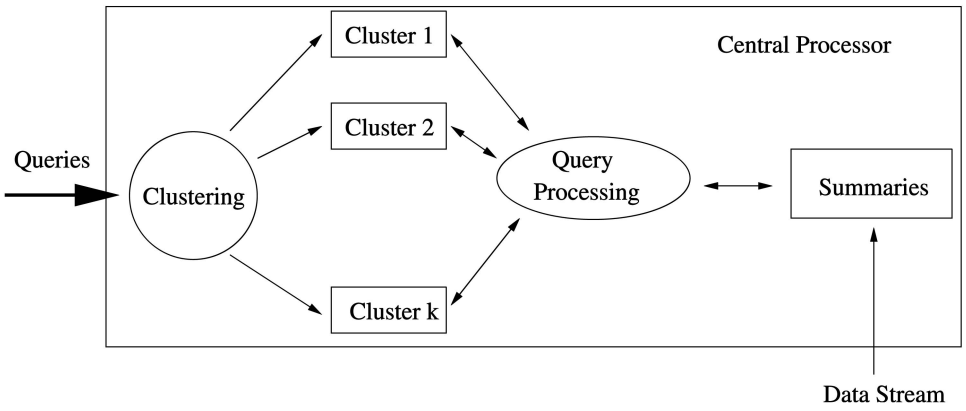

Fig. 3. Processing continuous quantile queries.

five sets of queries are generated with 1K, 5K, 10K, 50K, and 100K queries, respectively. We record the total costs of reprocessing each query upon the arrival of a new data element, but the costs for continuously maintaining quantile summaries are subtracted. Our experiment results show that, when the query number reaches 50K, the required processing time has already exceeded 10,000 seconds, while the process of 100K queries in this way cannot finish in 5 hours (thus, aborted by us). This makes it impossible to support a massive number of continuous quantile queries in real time against a rapid data stream by simply applying the existing technique.

3 Processing Whole Data Streams

In this section, we present novel techniques to process continuous quantile queries over a whole data stream. As depicted in Fig. 3, we propose to process continuous queries by the following two phases at a central processor:

Clustering. Cluster together the queries which share some common results.

Processing Clusters. Process a cluster of queries as a single query by trigger-based lazy update techniques and multicast the results of a query cluster to the relevant query sites.

As mentioned earlier, in such a system, we assume that all queries are processed at the central site, while query sites (e.g., mobiles) cannot process any query but receive results only. In the rest of the section, we assume that an ϵ-approximate summary S is continuously maintained by GK-algorithm over a whole data stream; that is, the GK-algorithm is used as a black box in our techniques.

This section is organized as follows: We first present fundamentals in our query processing techniques. Then, we present our clustering algorithms. Finally, we present our efficient query processing techniques using a trigger-based lazy update paradigm.

3.1 Queries, Intervals, and Clusters

In many applications, queries from different users may share some common answers. Below is a key observation:

Observation 1. *Suppose that* $q_1 = (0.50, 0.05)$ *and* $q_2 = (0.49, 0.06)$ *are two quantile queries, and the underlying summary S has the precision* 0.01 *(i.e.,* $\epsilon = 0.01$*). Intuitively, a result tuple for* q_1 *should also be a result tuple for a quantile* ϕ*, with the precision* 0.01*, in the interval* $[0.5 - 0.05 + 0.01, 0.5 + 0.05 - 0.01] = [0.46, 0.54]$ *and vice versa. Similarly, the second query* q_2 *can be transferred into the interval* $[0.44, 0.54]$*. Thus, the common interval is* $[0.46, 0.54]$ *and a result to any* $(\phi, 0.01)$ *for* $\phi \in [0.46, 0.54]$ *is a result for both queries. Consequently, we can group* q_1 *and* q_2 *together and query a quantile in* $[0.46, 0.54]$ *with the precision* 0.01*.*

Generally, we may expect that a quantile query $q = (\phi_q, \epsilon_q)$ over an ϵ-approximate ($\epsilon \leq \epsilon_q$) summary S can be transferred into the following interval:

$$I_q = [\phi_q - (\epsilon_q - \epsilon), \phi_q + (\epsilon_q - \epsilon)] \cap (0, 1], \tag{2}$$

so that an answer of (ϕ, ϵ) for a $\phi \in I_q$ can answer q. Let R_{ϕ_q,ϵ_q} denote the set of tuples that are valid results of query $q = (\phi_q, \epsilon_q)$ against S.

Theorem 2. *Suppose that* $q = (\phi_q, \epsilon_q)$ *is a quantile query. Then,* $\cup_{\phi \in I_q} R_{\phi,\epsilon} = R_{\phi_q,\epsilon_q}$ *against an* ϵ-approximate *summary* S*.*

Proof. Suppose that $(u_i, r_i^-, r_i^+) \in \cup_{\phi \in I_q} R_{\phi,\epsilon}$; that is, there is a $\phi \in I_q$ such that

$$\phi N - \epsilon N \leq r_i^- \leq r_i^+ \leq \phi N + \epsilon N. \tag{3}$$

Since $\phi \in I_q$, (3) immediately implies:

$$\phi_q N - \epsilon_q N \leq r_i^- \leq r_i^+ \leq \phi_q N + \epsilon_q N. \tag{4}$$

Thus, $(u_i, r_i^-, r_i^+) \in R_{\phi_q,\epsilon_q}$; consequently, $\cup_{\phi \in I_q} R_{\phi,\epsilon} \subseteq R_{\phi_q,\epsilon_q}$.

Suppose that $t_i = (u_i, r_i^-, r_i^+) \in R_{\phi_q,\epsilon_q}$ generated by the GK-algorithm, then this t satisfies (4). Therefore,

$$(\phi_q - (\epsilon_q - \epsilon))N - \epsilon N \leq r_i^-, \tag{5}$$

$$r_i^+ \leq (\phi_q + (\epsilon_q - \epsilon))N + \epsilon N. \tag{6}$$

Note that, according to the GK-algorithm, it is immediate that:

$$-\epsilon N < r_i^- \leq r_i^+ \leq (1 + \epsilon)N. \tag{7}$$

By the condition 3 of Theorem 1, we have that $r_i^+ - \epsilon N \leq r_i^- + \epsilon N$. This, together with (5), (6), and (7), immediately implies that $I_q \cap [\frac{r_i^+}{N} - \epsilon, \frac{r_i^-}{N} + \epsilon] \neq \emptyset$.

Consequently, $t_i \in R_{\phi,\epsilon}$ for any $\phi \in I_q \cap [\frac{r_i^+}{N} - \epsilon, \frac{r_i^-}{N} + \epsilon]$. Thus, $R_{\phi_q,\epsilon_q} \subseteq \cup_{\phi \in I_q} R_{\phi,\epsilon}$. □

We call I_q *the query interval* of q. Suppose that $Q = \{(\phi_q, \epsilon_q)\}$ is a set of quantile queries. We use I_Q to denote the common part of query intervals in Q (i.e., $I_Q = \cap_{q \in Q} I_q$). Theorem 2, together with the fact that $R_{\phi,\epsilon} \neq \emptyset$ ($\forall \phi \in (0, 1]$) against S, implies that all queries in Q with different ϕ_q and ϵ_q can be answered by a quantile query (ϕ, ϵ) against S, where $\phi \in I_Q$, if $I_Q \neq \emptyset$. That is, any tuple t in $\cup_{\phi \in I_Q} R_{\phi,\epsilon}$ can be a common answer to all queries in Q. Therefore, all queries in Q can be treated virtually as a single query and I_Q is called the *cluster interval* of Q. A tuple in $\cup_{\phi \in I_Q} R_{\phi,\epsilon}$ is called a *result tuple* of Q. These are fundamental to our clustering algorithm.

3.2 Preliminaries of Lazy Updates

We present a way to choose a "good" tuple, so that it can stay as long as possible as a result tuple against the arrival of new data elements. We first present a sufficient and necessary condition for a tuple (u, r^-, r^+) in an ϵ-summary, generated by the GK-algorithm, to be a result tuple of a query cluster Q_i.

Theorem 3. *Suppose that $t = (u, r^-, r^+)$ is a tuple in an ϵ-approximate summary S (generated by the GK-algorithm) of N data elements, and the interval I_{Q_i} of a query cluster is $[a_i, b_i]$. Then, t is a result tuple of Q_i (i.e., $t \in \cup_{\phi \in I_{Q_i}} R_{\phi,\epsilon}$) if and only if*

$$(a_i - \epsilon)N \leq r^- \leq r^+ \leq (b_i + \epsilon)N. \tag{8}$$

Proof. Suppose that t is a result tuple; that is, $t \in \cup_{\phi \in I_{Q_i}} R_{\phi,\epsilon}$. Then, according to the definition, there is a $\phi \in [a_i, b_i]$ such that

$$(\phi - \epsilon)N \leq r^- \leq r^+ \leq (\phi + \epsilon)N. \tag{9}$$

Consequently, (8) holds.

Now, we prove the sufficient condition. Suppose that (8) holds. We need to find a $\phi \in [a_i, b_i]$ such that (9) holds. As S is generated by the GK-algorithm, S follows the three conditions in Theorem 1. Thus, $r^+ - \epsilon N \leq r^- + \epsilon N$. From (8), it is immediate that $[a_i, b_i] \cap [\frac{r^+}{N} - \epsilon, \frac{r^-}{N} + \epsilon] \neq \emptyset$. We can immediately verify that any $\phi \in [a_i, b_i] \cap [\frac{r^+}{N} - \epsilon, \frac{r^-}{N} + \epsilon]$ meets the inequalities in (9). □

Remark. According to the definition, a query interval I_q may be open on a_i if $a_i = 0$. However, it can be immediately shown that Theorem 3 and other theorems in this section all hold for such an extreme situation. To simplify our discussions, we present our results based on closed query intervals.

Suppose that $I_{Q_i} = [a_i, b_i]$. Let:

$$\alpha_{Q_i,t,N} = \begin{cases} +\infty & a_i \leq \epsilon \\ \frac{r^- + \epsilon N - a_i N}{a_i - \epsilon} & \text{otherwise,} \end{cases} \tag{10}$$

$$\beta_{Q_i,t,N} = \begin{cases} +\infty & 1 \leq b_i + \epsilon \\ \frac{b_i N - r^+ + \epsilon N}{1 - b_i - \epsilon} & \text{otherwise,} \end{cases} \tag{11}$$

$$L_{Q_i,t,N} = \min\{\alpha_{Q_i,t,N}, \beta_{Q_i,t,N}\}. \tag{12}$$

From Theorem 3, the corollary below follows immediately:

Corollary 1. *Suppose that $t = (u, r^-, r^+)$ is a tuple in an ϵ-approximate summary S (generated by GK-algorithm) of N data elements. Then, t is a result tuple of query cluster Q_i if and only if $L_{Q_i,t,N} \geq 0$.*

Suppose that $t = (u, r^-, r^+)$ is a result tuple to Q_i with the cluster interval $[a_i, b_i]$; that is, (8) holds. Now, we calculate a lower bound on the number δ of new arrival data elements, which keep (8) valid. According to the GK-algorithm, if these δ new elements are all inserted before the tuple t, then t will be changed to $(u, r^- + \delta, r^+ + \delta)$. If all are inserted after t, then t remains unchanged in the summary. Since $a - \epsilon \leq 1$, it is immediate that if all these δ data elements are inserted before t and (8) holds, then $(a_i - \epsilon)(N + \delta) \leq r^- + \delta$ always holds. However, in this case, to make $r^+ + \delta \leq (b_i + \epsilon)(N + \delta)$ hold, the following inequality must hold:

$$\delta(1 - b_i - \epsilon) \leq (b_i + \epsilon)N - r^+. \tag{13}$$

On the other hand, if all δ elements are inserted after t, then $r^+ \leq (b_i + \epsilon)(N + \delta)$ always holds due to (8). However, to make $(a_i - \epsilon)(N + \delta) \leq r^-$ hold, the following equality must hold:

$$(a_i - \epsilon)\delta \leq r^- - (a_i - \epsilon)N. \tag{14}$$

Note that, regardless of the value of δ ($\delta > 0$), (13) always holds if $1 \leq b_i + \epsilon$, and (14) always holds if $a_i \leq \epsilon$. But, $1 \leq b_i + \epsilon$ and $a_i \leq \epsilon$ usually cannot hold simultaneously unless every query q in Q_i can share a result tuple to the quantile query $(0.5, 0.5)$; in this case, any tuple in the summary can be an answer forever.

Theorem 4. *Suppose that $t = (u, r^-, r^+)$ is a tuple in an ϵ-approximate summary S (generated by the GK-algorithm) of N data elements, and $L_{Q_i,t,N} \geq 0$ for a query cluster Q_i. Then, u can remain as a result tuple, with respect to Definition 3, to the queries in Q_i until the $\lfloor L_{Q_i,t,N} \rfloor + 1$th new element (exclusive) arrives regardless whether t is still kept as a tuple in the continuously changed summary.*

Proof. Suppose that v is always kept in the summary during when these $\lfloor L_{Q_i,t,N} \rfloor$ new elements arrive. Then, the theorem is immediate according to the above arguments.

If v disappears on the half way (merged into another tuple) when those elements are inserted, using similar arguments as above and those in the proof of Theorem 3 also immediately implies that v can be used as a result, according to Definition 3. □

In fact, $L_{Q_i,t,N}$ is quite tight. As analyzed, if $L_{Q_i,t,N} = \beta_{Q_i,t,N}$ and the $\lfloor L_{Q_i,t,N} \rfloor + 1$ new elements are inserted before t, then t is disqualified as a result tuple to Q_i. Similarly, $L_{Q_i,t,N}$ is tight when $\alpha_{Q_i,t,N} < \beta_{Q_i,t,N}$.

Corollary 1 and Theorem 4 state that we need only to reprocess a query cluster Q_i when the $(L_{Q_i,t,N} + 1)$th new element arrives. The larger the value $L_{Q_i,t,N}$ is, the longer (in terms of the number of new elements) we can wait to update the result. While a good estimation of $L_{Q_i,t,N}$ tends to be difficult to obtain due to the uncertainty of summary tuples maintained by the GK-algorithm, below we give a lower-bound on $L_{Q_i,t,N}$:

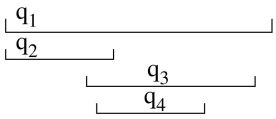

Fig. 4. Two clusters.

Theorem 5. *Suppose that $I_{Q_i} = [a_i, b_i]$ is a cluster interval. Then, there is a tuple t in an ϵ-approximate summary S maintained by the GK-algorithm, such that $L_{Q_i,t,N} \geq \frac{|I_{Q_i}|}{2(1-\epsilon)}N$, where $\epsilon < 1$.*

Proof. Let $\phi = \frac{a_i+b_i}{2}$. Then, there is a tuple $t = (v, r^-, r^+) \in S$ to answer the quantile query (ϕ, ϵ) according to Theorem 1. Thus, $r^+ - \epsilon N \leq \phi N \leq r^- + \epsilon N$. Note that:

$$\frac{|I_{Q_i}|}{2}N = \frac{b_i - a_i}{2}N = \left(\frac{b_i + a_i}{2} - a_i\right)N \leq r^- + \epsilon N - a_i N.$$

Thus, $\alpha_{Q_i,t,N} \geq \frac{|I_{Q_i}|}{2(1-\epsilon)}N$.

Similarly, we can show that $\beta_{Q_i,t,N} \geq \frac{|I_{Q_i}|}{2(1-\epsilon)}N$. □

Theorem 6. *Suppose that, for each cluster Q_i, a trigger-based lazy update paradigm chooses the maximum value of $L_{Q_i,t,N}$ every time after $L_{Q_i,t,N} + 1$ new elements, where N is the number of elements in a data stream when $L_{Q_i,t,N}$ is chosen. Then, this paradigm searches new values of $L_{Q_i,t,N}$ (thus, reevaluate Q_i) by $O(\log_{1+\frac{|I_{Q_i}|}{2(1-\epsilon)}} N)$ times if $|I_{Q_i}| \neq 0$.*

Proof. We give an initialization N_1. For the first N_1 data elements, we assume that the paradigm has to always find the new L values. We could make $N_1 = 1 + \frac{|I_{Q_i}|}{2(1-\epsilon)}$.

Then, Theorem 5 immediately implies that

$$\left(1 + \frac{|I_{Q_i}|}{2(1-\epsilon)}\right)^n N_1 \leq N,$$

where n is the number of times to search new L values after the first N_1 times. The theorem immediately follows. □

Theorem 6, together with Theorem 4, implies that such a trigger-based lazy update paradigm needs only to reevaluate a query cluster by logarithmic times instead of by $O(N)$ times.

3.3 Clustering Predefined Queries

In this section, we assume that a set Q of queries is predefined. Our goal is to cluster queries, so that the total number of times to reevaluate query clusters may be minimized.

Suppose that a query interval $I_q = [a_q, b_q]$ contains another query interval $I_{q'} = [a_{q'}, b_{q'}]$. According to the discussions in Section 3.1, we need only to process $I_{q'}$ by clustering q and q' together. Viewing each interval $[a, b]$ as a point (a, b) in a 2D-space, we can run a *skyline*-like computation algorithm [18] in 2D-space as a preprocess to remove the containment relationships by keeping only the query intervals which do not contain another interval. For example, in Fig. 4, after preprocessing there are two clusters: 1) queries containing the interval of q_2 and 2) queries containing the interval of q_4. Note that q_1 can be put into either the cluster with q_2 or the cluster with q_4 but not in both. Clearly, our optimal clustering problems below based on the output of such preprocess do not affect the optimality of the solution since the preprocess output is a subset of pregiven queries that also have to be processed.

As mentioned in Section 3.2, in our trigger-based lazy update algorithm, the number of total times to reevaluate query clusters equal the total summation of $L_{Q,t,N}$s and a good estimation of $L_{Q,t,N}$ seems impossible. We propose the following two alternative optimization problems.

3.3.1 Minimization against Worst-Cases

Theorem 6 shows that an upper-bound of the number of times to reprocess a query cluster is approximately proportional to $\frac{\log N}{|I_Q|}$ for a given ϵ. We aim to minimize the upper-bounds in Theorem 6.

Minimization against Worst-Cases (MWC)
Given: A set of quantile queries $Q = \{q_j : 1 \leq j \leq \Gamma\}$, $\epsilon < 1$ (the precision of the underlying summary), where $|I_q| \neq 0$ and query intervals do not contain each other.
Problem: Find a set of clusters $\{Q_i : 1 \leq i \leq k\}$ such that

1. $\cup_{i=1}^{k} Q_i = Q$,
2. $Q_i \cap Q_j = \emptyset$ for $1 \leq i < j \leq k$, and
3. $\sum_{i=1}^{k} \frac{1}{\ln(1+\frac{|I_{Q_i}|}{2(1-\epsilon)})}$ is minimized.

Note that after the preprocessing above, for a *point* query q (i.e., $|I_q| = 0$) I_q is not contained by another query interval; thus, it forms a query cluster and is removed for a further consideration. This is the reason why, in the input of MWC, we assume $|I_q| \neq 0$. Next, we show that the problem of MWC can be solved efficiently by the dynamic programming technique.

We can assume that the input query intervals in MWC are already sorted increasingly on query intervals (treating each interval as a point in 2D-space) after the above preprocessing. For each i, let Π_i denote the optimal clustering (i.e., an answer to MWC) against the first i intervals and $C(\Pi_i) = \sum_{Q_j \in \Pi_i} \frac{1}{\ln(1+\frac{|I_{Q_j}|}{2(1-\epsilon)})}$.

Lemma 1. *Suppose that, in an optimal clustering Π_i against the first i intervals, there is a cluster that uses the intersection of the ith interval and the jth interval as its query interval $(1 \leq j < i)$. Then, the clustering by having query clusters in Π_{j-1} and the query cluster consisting of q_l (for $j \leq l \leq i$) must be also optimal against the first i intervals and can replace Π_i. Note that $\Pi_0 = \emptyset$.*

Proof. Let $I_{i,j}$ denote the intersection. Clearly, each q_l (for $j \leq l \leq i$) contains $I_{i,j}$. Thus, modifying Π_i by putting q_l in the cluster while retaining the cluster interval $I_{i,j}$ will not increase $C(\Pi_i)$. The lemma immediately follows. □

Lemma 1 is the key observation by applying the dynamic programming techniques. We iteratively construct Π_i from $i = 1$ and output Π_Γ as the solution. The algorithm is presented in Algorithm 1.

Algorithm 1 MWC
Input:
$Q = \{q_i : 1 \leq i \leq \Gamma\}$. Assume that Q is already sorted increasingly, by the preprocessing, on I_i $(= I_{q_i})$.
Output:
A group of clusters $\{Q_j : 1 \leq j \leq k\}$.

Description
1: $\Pi_0 := \emptyset$; $C(\Pi_0) = 0$;
2: **for** $i = 1$ **to** Γ **do**
3: $C(\Pi_i) := C(\Pi_{i-1}) + \frac{1}{\ln(1+\frac{|I_i|}{2(1-\epsilon)})}$;
4: $\Pi_i := \Pi_{i-1} \cup \{\{q_i\}\}$; $j := i - 1$;
5: **while** $I_j \cap I_i \neq \emptyset$ & $j \geq 1$ **do**
6: $j := j - 1$;
7: **if** $C(\Pi_{j-1}) + \frac{1}{\ln(1+\frac{|I_j \cap I_i|}{2(1-\epsilon)})} < C(\Pi_i)$ **then**
8: $\Pi_i := \Pi_{j-1} \cup \{\{q_l : j \leq l \leq i\}\}$;
9: $C(\Pi_i) := C(\Pi_{j-1}) + \frac{1}{\ln(1+\frac{|I_j \cap I_i|}{2(1-\epsilon)})}$;
10: Report Π_Γ;

From Lemma 1, it can be immediately verified that Algorithm 1 is correct; that is, it gives a correct answer to MWC. Moreover, the algorithm runs in time $O(\sum_{i=1}^{\Gamma} k_i)$, where k_i is the number of intervals, located before I_{q_i} in the sorting list, overlapping with I_{q_i}. In theory, $\sum_{i=1}^{\Gamma} k_i = O(\Gamma^2)$ if all intervals have a common part. However, in practice, this rarely happens; moreover, Γ is not very large after the preprocessing. Our performance evaluation also confirms that the algorithm is very efficient in practice. Furthermore, this algorithm can be executed in a linear space $O(\Gamma)$ if we build a link between Π_i and Π_j when Π_i uses Π_j as a subplan in our algorithm.

3.3.2 Minimizing the Number of Clusters

It aims to cluster the queries so that the number of clusters is minimized. This problem is well studied in computational geometry [28], [32]. It can be solved in time $O(\Gamma)$ by a simple sweeping line technique if the preprocess is executed and the intervals are sorted. In our performance evaluation, we refer to this technique as $\mathrm{QC_B}$. The following theorem is immediate.

Theorem 7. *Suppose that, in a given set Q of continuous quantile queries, the highest approximate precision is ϵ_1; that is, for each query $(\phi_q, \epsilon_q) \in Q$, $\epsilon_q \geq \epsilon_1$. Further suppose that $\epsilon_1 > \epsilon$, where an ϵ-approximate summary is maintained by the GK-algorithm and l_{min} is the minimum query length in the given set of continuous queries. Then, $\epsilon_1 - \epsilon \leq l_{min} \leq 2(\epsilon_1 - \epsilon)$. Moreover, the number of clustered generated by Algorithm $\mathrm{QC_B}$ is bounded by $\frac{1}{l_{min}}$.*

Theorem 7 immediately implies that the number of clusters to be generated by Algorithm $\mathrm{QC_B}$ is bounded by $O(\frac{1}{\epsilon_1 - \epsilon})$. When $\epsilon_1 = \epsilon$, the number of clusters generated by $\mathrm{QC_B}$ does not have such a theoretic upper-bound guarantee and depends on query distribution. The experiment results in Section 5 demonstrate that our query clustering techniques are also very effective in such situations.

3.4 Online Clustering Algorithm

In many applications, ad hoc continuous queries, rather than predefined, may be registered and dropped in an arbitrary fashion. It may be computationally too expensive to invoke Algorithm 1 (or $\mathrm{QC_B}$), per query insertion and deletion, to generate the new optimal query clusters and then to reprocess all clusters.

In our algorithm, we enforce an online environment by treating each existing query cluster as a query while ignoring the detailed "geometric" information of already clustered individual queries. The existing clusters will remain unchanged unless the arrival of a new query causes clusters[2] to be merged together with the new query to form a new query cluster. In this case, reprocessing only the newly formed cluster does not potentially lead to extra processing overheads since a new query has to be processed anyway. Our algorithm, outlined in Algorithm 2, is based on a greedy paradigm to enforce the "local" optimality.

Algorithm 2. Online Clustering
Description
1: **Register a Query** q:
2: **if** $\exists I_{Q_j} \cap I_q \neq \emptyset$ **then**
3: **if** $\exists I_{Q_i} \subseteq I_q$ **then**
4: $Q_{new} := (\cup_{q_l \in Q_i} \{q_l\}) \cup \{q\}$; $I_{Q_{new}} := I_{Q_i}$;
5: **else**
6: find a Q_i such that $|Q_i \cap I_q|$ is maximized;
7: $I_{Q_{new}} := I_{Q_i} \cap I_q$;
8: $Q_{new} := (\cup_{I_{Q_i} \supseteq I_{Q_{new}}} \cup_{q_l \in I_{Q_i}} \{q_l\}) \cup \{q\}$;
9: **else**
10: $Q_{new} := \{q\}$; $I_{Q_{new}} := I_q$
11: **Drop a Query** q **from** Q_j:
12: $Q_j := Q_j - \{q\}$;
13: **if** $Q_j = \emptyset$ **then**
14: Drop Q_j.

To speed up computation, an in-memory $B+$ tree is used to store the clusters sorted based on the left end of intervals. Based on the property that the query intervals of existing clusters are not mutually inclusive, it is immediate that determining the query cluster to be merged with a new issued q takes $O(\log k + s)$, where k is the number of clusters and s is the number of clusters intersecting I_q. Note that the old clusters have to be removed from $B+$ tree. As all those clusters are in the consecutive positions, such an update of $B+$ can be done in $O(\log k + s)$ if we do it in a bottom-up fashion. Thus, the costs of registering a query takes $O(\log k + s)$. Once a query cluster drops due to a drop of a query, it takes $O(\log k)$ to update the $B+\mathrm{tree}$.

Once a query is removed, the corresponding query cluster may increase its length. This is not dealt with in our online algorithm because of the maintenance costs. To compensate this, we could set up thresholds for the number of new clusters created and the number of queries dropped. Once one threshold is reached, the reclustering algorithm (Algorithm 1 or $\mathrm{QC_B}$) runs at the background. Once the reclustering finishes, new clusters are used to replace old clusters. Our experiment results, nevertheless, indicated that the online algorithm without this offline process already performs very well.

3.5 Processing Continuous Queries by Lazy Updates

As shown in Section 2.4, eager updates per data element are not scalable enough to accommodate real-time processing against high-speed data streams. In this section, we present our trigger-based lazy update techniques.

2. It is possible that two or more existing clusters whose intervals have a common part and such a common part contains the cluster interval of a newly formed cluster.

3.5.1 Synchronous Update

To update a result tuple of Q_i, we always choose a tuple t_{Q_i} with the maximum $L_{Q_j,t_{Q_j},N}$ (in (12)). We use a global trigger T_{global} that takes the smallest value among all $L_{Q_j,t_{Q_j},N}$. T_{global} is reduced by 1 upon new data item arrival. When T_{global} is reduced to a negative value, the algorithm is invoked to compute new result tuples for all clusters, respectively.

We assume that a set $\mathbb{Q}$ of query clusters has been sorted on the query intervals increasingly. This is maintained either in our online clustering algorithm by $B+$ tree or in Algorithm 1 or in $\mathrm{QC_B}$.

Note that the cluster intervals do not contain each other. Further, $L_{Q_i,t,N}$ is monotonic regarding to a_i, b_i, r^-, and r^+, respectively. It can be easily shown that:

- For a tuple t in the summary S, if t is a result tuple of Q_i but not a result tuple of Q_{i+1}, then t cannot be a result tuple of Q_j for $j \geq i+1$.
- For a query cluster Q_l, if t_i in S is a result tuple to Q_l but t_{i+1} in S is not a result tuple to Q_l, then t_j cannot be a result tuple of Q_l for $j \geq i+1$.

Based on these, the algorithm, presented in Algorithm 3, runs in a "sort-merge" fashion between S and $\mathbb{Q}$ to update the results for all cluster.

Algorithm 3 Synchronous Update
Input:
List $\mathbb{Q}$: set of query clusters sorted on their intervals.
S: ϵ-summary.
Output:
Answers to every query cluster in $\mathbb{Q}$ and T_{global}.
Description:
1: List $H := \emptyset$; // list of intermediate results
2: Iterator $fq :=$ first query cluster in $\mathbb{Q}$;
3: Iterator $fs :=$ first tuple in S;
4: $T_{global} := \infty$;
5: **while** fq is not null or $H \neq \emptyset$ **do**
6: **if** fs is not null **then**
7: **for** each element $(Q, t_{best}, T_Q) \in H$ **do**
8: **if** $L_{Q,fs,N} < 0$ **then**
9: Report (Q, t_{best}); // t_{best} is the result of Q
10: $T_{global} := \min\{T_{global}, T_Q\}$;
11: Remove (Q, t_{best}, T_Q) in H;
12: **else if** $L_{Q,fs,N} > T_Q$ **then**
13: $t_{best} := fs$; $T_Q := L_{Q,fs,N}$; // update this intermediate result
14: **while** $L_{fq,fs,N} \geq 0$ **do**
15: append $(fq, fs, L_{fq,fs,N})$ to the tail of H;
16: $fq :=$ the next in $\mathbb{Q}$;
17: $fs :=$ the next in S;
18: **else**
19: Report H; Report T_{global};

It can be immediately verified that Algorithm 3 can always provide a quantile query with a result within its precision requirement. Once a new cluster Q' forms, Algorithm 3 is invoked with only Q' as the input. Then, update T_{global} to be $L_{Q',t_{Q'},N}$ if $L_{Q',t_{Q'},N}$ is smaller than the existing T_{global}. Clearly, if a set of continuous queries is pregiven and $c_{min} \neq 0$, Algorithm 3 will be invoked $O(\log_{1+\frac{c_{min}}{2(1-\epsilon)}} N)$ times according to Theorem 6 where c_{min} is the minimum value of cluster intervals. Note that, if $c_{min} = 0$, our performance study indicates this algorithm performs reasonably well though no such upper-bound exists.

3.5.2 Asynchronous Update

While the trigger maintenance costs in Algorithm 3 are low, it could happen very often that all the other valid query results are updated just because one query result needs to be updated. This may cause a great deal of unnecessary computation. Further, a synchronous update paradigm may potentially cause a significant delay of new results for processing the whole set of query clusters.

We present an asynchronous update technique by having one trigger T_{Q_i} for each cluster Q_i, which initially takes the value in (12): $T_{Q_i} = L_{Q_i,t,N}$.

Our asynchronous update-based query processing algorithm is outlined in Algorithm 4. Theorem 6 implies that Algorithm 4 will guarantee to reprocess a static (i.e., no query insertion or deletion to the cluster) query cluster Q_i by $O(\log_{1+\frac{|I_{Q_i}|}{2(1-\epsilon)}} N)$ times if $|I_{Q_i}| \neq 0$.

Algorithm 4 Asynchronous Update
Case 1. A new data element arrives.
Step 1: Reduce the trigger value by 1 for each trigger.
Step 2: If a trigger T_t is fired (i.e., trigger value is negative), then reprocess the query cluster, update its trigger value, and reinsert the trigger into the trigger list.
Case 2. Update triggers, whenever necessarily, after dropping or insert a query.

Now, we present the execution details for each step in Case 1 and Case 2, together with effective data structures.

Step 1. In Algorithm 4, each trigger value has to be updated upon a data item arrival. To avoid actually updating each trigger, we organize the trigger list $\top$ in a *min-heap* [5]. Instead of an actual trigger value, we store at each node i (apart from the root) the difference between the actual trigger value of i and that of the parent of i. At the root, we store the minimum trigger value.

Once a new data item arrives, we need only to reduce the root value by 1. Therefore, Step 1 can be implemented in constant time. Once the trigger value at root is negative, the trigger is fired and deleted. This can be handled by the standard heap technique in $O(\log|\top|)$.

Step 2. Once a trigger is fired, the corresponding query cluster Q_i has to be reevaluated to find a tuple t with maximum $L_{Q_i,t,N}$. Since the underlying summary S is ϵ-approximate, such a tuple must lead to a nonnegative $L_{Q_i,t,N}$.

It is immediate that the tuples in the summary leading to nonnegative L values must be in the consecutive positions because of the monotonic properties of r^-, r^+, $\alpha_{Q_i,t,N}$, and $\beta_{Q_i,t,N}$.

While searching for a new result tuple for Q_i, the system certainly can start from one end of the summary. However, this may be inefficient if the tuple t is too far from the end. To avoid this, we propose to link a trigger to a tuple used in this trigger. Therefore, once the trigger fires, query reprocessing starts from the tuple. Note that a tuple t in the summary may be merged into another tuple t' in the

GK-algorithm. In this case, the unfired triggers hanging on t are all moved to t'. To do this in constant time, at each tuple, triggers are organized as a linked list. Therefore, we need only to move the whole group.

As $r^+ - r^- \leq 2\epsilon N$ for every tuple $t = (u, r^-, r^+)$ in S, it is immediate that $\alpha_{Q_i,t,N}$ and $\beta_{Q_i,t,N}$ cannot be both negative. Further, $\alpha_{Q_i,t,N}$ is monotonically increasing regarding r^-, and $\beta_{Q_i,t,N}$ is monotonically decreasing regarding r^+. Thus, Q_i can be reprocessed starting from t against the three cases below:

- If $\alpha_{Q_i,t,N} \geq 0$ and $\beta_{Q_i,t,N} < 0$, run the search technique of Algorithm 3 to the left from t.
- If $\alpha_{Q_i,t,N} < 0$ and $\beta_{Q_i,t,N} \geq 0$, run the search technique of Algorithm 3 to the right from t.
- If $\alpha_{Q_i,t,N} \geq 0$ and $\beta_{Q_i,t,N} \geq 0$, run the search technique of Algorithm 3 to the both side alternately from t.

Once the largest $L_{Q_i,t,N}$ is obtained, insert it into the trigger heap; this takes $O(\log \top)$.

Case 2. Dropping or inserting a query may either cause a drop of a query cluster, or merge of several clusters into one cluster, or creation of a new cluster. A deletion or an insertion in a trigger heap $\top$ takes $O(\log \top)$.

When several clusters are merged into one cluster, we reprocess (run search technique in Algorithm 3) the merged cluster from any tuple linked to an old cluster using one of the above three cases. To process a new cluster, we run the search technique in Algorithm 3 from the beginning of the summary.

4 Processing Sliding Windows

In this section, we present our algorithm against sliding windows. We use the algorithm in [19] to continuously maintain an ϵ-approximate summary for a sliding window; it was described briefly in Section 2.3.2.

It has been proven [19] that the middle part bounded by the big rectangle (as depicted in Fig. 2) is always an ϵ-approximate summary for the N elements in the whole window. Further, the right most part will become the tail of the middle part once it is full and the left most part will be removed consequently. According to the properties, we need only to focus on the middle part which is static before every whole batch of the most recent $\frac{\epsilon N}{2}$ data elements arrive. Therefore, we invoke Algorithm 3 to process the middle part every time after the arrival of a batch of new $\frac{\epsilon N}{2}$ data elements. This is how we process continuous quantile queries under sliding windows. It can be immediately combined with our query clustering algorithms.

5 Performance Studies

In our performance study, we focus on our techniques. This is because the only existing technique *first-hit* is not immediately applicable to real-time processing of a large set of queries, as shown in Section 2.4. The following algorithms have been implemented and evaluated:

- $\mathrm{QC_A}$: The query clustering algorithm, Algorithm 1, for a given set of continuous quantile queries.

TABLE 1
Parameters

Notation	Definition (Default Values)
N_d	Size of a data stream (2M)
D_d	Data Model/Value Distribution (semisort)
N_q	Number of continuous queries (1M)
$r_{(\phi_q,\epsilon_q)}$	Ranges of ϕ_q ($(0,1]$) and ϵ_q ($(0,1)$)
$D_{(\phi_q,\epsilon_q)}$	Distribution of (ϕ_q, ϵ_q) (uniform)
A_q	Query arrival patterns (pre-defi ned)
ϵ	Precision of a Summary (0.005)

- $\mathrm{QC_B}$: The query clustering algorithm based on an application of the swap-line technique in [28], [32] to minimize the number of clusters, as described in the second part of Section 3.3.
- OQC: Our online query clustering algorithm, Algorithm 2, in Section 3.4.
- SU: The synchronous update algorithm, Algorithm 3 in Section 3.5, for processing continuous quantile queries. (Note that, here, we run the algorithm without clustering queries).
- CSU: A combination of Algorithm 3 and $\mathrm{QC_A}$ (or $\mathrm{QC_B}$) or a combination of Algorithm 3 and OQC depending on if queries are issued dynamically.
- CASU: A combination of the asynchronous update algorithm (Algorithm 4 in Section 3.5) and $\mathrm{QC_A}$ (or $\mathrm{QC_B}$) or a combination of Algorithm 4 and OQC depending on whether queries are issued dynamically.

We also implemented the summary maintenance techniques for whole streams [13] and for sliding windows [19] to support our experiments.

We implement all the algorithms in C++ and compile them using GNU gcc v2.95 without a special code optimization. We conduct our experiments on a PC with an Intel P4 2.4G Hz CPU and 512MB memory. The operating system is Debian Linux.

Table 1 lists possible factors that may affect the performance of our algorithms. Two parameters are used to describe a data stream, the size N_d, and the distribution D_d (specifying the *sortedness* of a data stream). In our experiments, D_d takes two random models—*uniform* and *normal*, as well as a partially sorted data—semisort [13], [19]. In a *semisort* data set, data elements are grouped by disjointed groups such that elements arrived in "later" blocks have larger values than those in earlier blocks though the elements in each block are not sorted. To describe a set of queries, the three parameters are used, 1) ranges of ϕ_q and ϵ_q, 2) distribution of (ϕ_q, ϵ_q), and 3) query arrival patterns. Finally, the guaranteed precision ϵ of an underlying summary is also used.

In our performance study, we evaluate the efficiency of our query techniques, as well as the effectiveness. We also evaluate the scalability of our techniques with respect to the number of input queries, query patterns, the number of

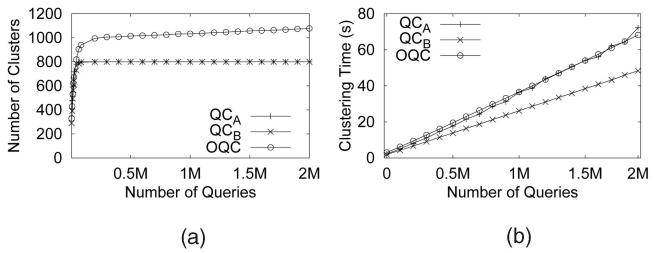

Fig. 5. Uniform distribution. (a) Cluster number. (b) Cluster running time.

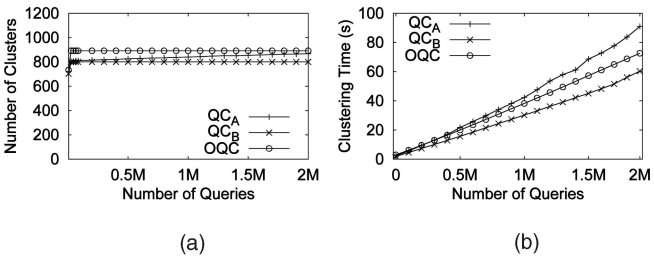

Fig. 6. Zipf distribution. (a) Cluster numbers. (b) Clustering running time.

data items, and the arrival rate of data items. Finally, we evaluate the performance of our sliding window techniques. As our techniques have a proven accuracy guarantee for any ϵ, it is less interesting to report our accuracy evaluation; thus, it is omitted.

5.1 Performance Study of Query Clustering

In the first set of experiments, we evaluate the performance of $\mathrm{QC_A}$, $\mathrm{QC_B}$, and OQC regarding the two query sets below.

Both query sets take 2M (2 millions) for its size and the default values for the ranges of ϕ_q and ϵ_q, respectively, with the restriction that $\epsilon_q < \phi_q$. In the first query set, $D_{(\phi_q,\epsilon_q)}$ follows a uniform distribution (i.e., a uniform distribution on (ϕ_q, ϵ_q)), while $D_{(\phi_q,\epsilon_q)}$ in the second query set follows a *zipf* distribution [36] with the zipf parameter 1.0.

To evaluate the online query clustering algorithm OQC, we randomize a query arrival order. To run these three algorithms, we make ϵ be the the minimum value of ϵ_q in the generated query sets; thus, $l_{min} = 0$ and $c_{min} = 0$. This makes Theorem 7 invalid; however, our experiment demonstrates that these three algorithms still generate (relatively) very small numbers of clusters. We record the numbers of clusters generated by OQC for the first arrived 20K queries, 60K queries, 100K queries, 200K queries, 400K queries, and so forth, respectively, as well as their running time. We also record the corresponding numbers of clusters generated by $\mathrm{QC_A}$ and $\mathrm{QC_B}$ and their running costs (time), respectively. The experiment results for the first query set (uniform distribution) are presented in Fig. 5, while the results for the second query sets are presented in Fig. 6. The results indeed indicate a great reduction of the number of queries if query clusters are used instead of each individual query. As demonstrated, cluster techniques are also very efficient. Note that both $\mathrm{QC_A}$ and the online clustering algorithm OQC require more computation time for Zipf query distribution. This is because, in such a distribution, there are many queries intersecting together; this makes $\mathrm{QC_A}$ and OQC less efficient.

In the second experiment, we evaluate the performance of OQC against deletion of queries. We take the two sets of queries generated in the first experiment, then randomly divide a query set into 10 batches and 20 batches, respectively. Consequently, we have four different query sets: uni10, uni20, zipf10, and zipf20. For instance, uni20 means that we randomly divide the query set, generated according to a uniform distribution into 20 batches.

In the experiment, each batch of queries are inserted in a random order. After the insertion of the first element of the ith ($i \geq 2$) batch and before inserting the $(i+1)$th batch, the queries in the $(i-1)$th batch are randomly dropped. Before inserting the ith batch, we record the ratio of the number of clusters generated by $\mathrm{QC_A}$ (against the remaining queries) and the number of clusters generated by OQC, as well as the average ratio of the number of clusters by $\mathrm{QC_A}$ over OQC. We also record such ratios between $\mathrm{QC_B}$ and OQC. These average ratios, with respect to these four data sets, are depicted in Fig. 7a. It demonstrated that OQC is very effective in such a dynamic environment. This, together with the evaluation of running time as depicted in Fig. 7, suggested that our online clustering technique is efficient and effective.

5.2 Query Processing Costs by $\mathrm{QC_A}$ and $\mathrm{QC_B}$

A query set with 2M queries is generated in a similar way as the query set (with uniform distribution of $D_{(\phi_q,\epsilon_q)}$) in the first experiment in Section 5.1 except $\epsilon_q \in [0.005, 0.02]$. Two data stream models are used (semisort and uniform). The others settings adopt system default values.

We run both $\mathrm{QC_A}$ and $\mathrm{QC_B}$ to generate clusters, respectively, and then call CASU to process these clusters in these two data streams. We record the number of total times triggers are fired, respectively, at a set of moments in

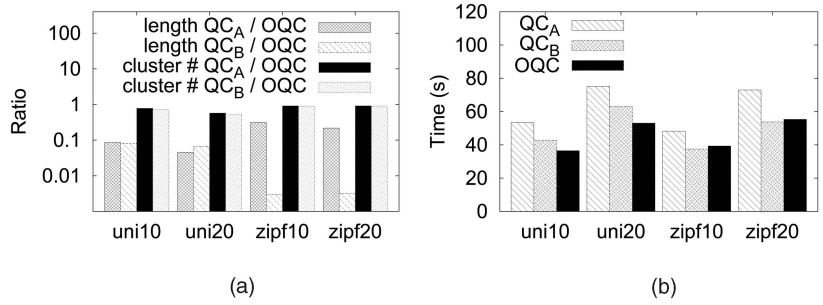

Fig. 7. Evaluation against different query patterns. (a) Ratio. (b) Computation time.

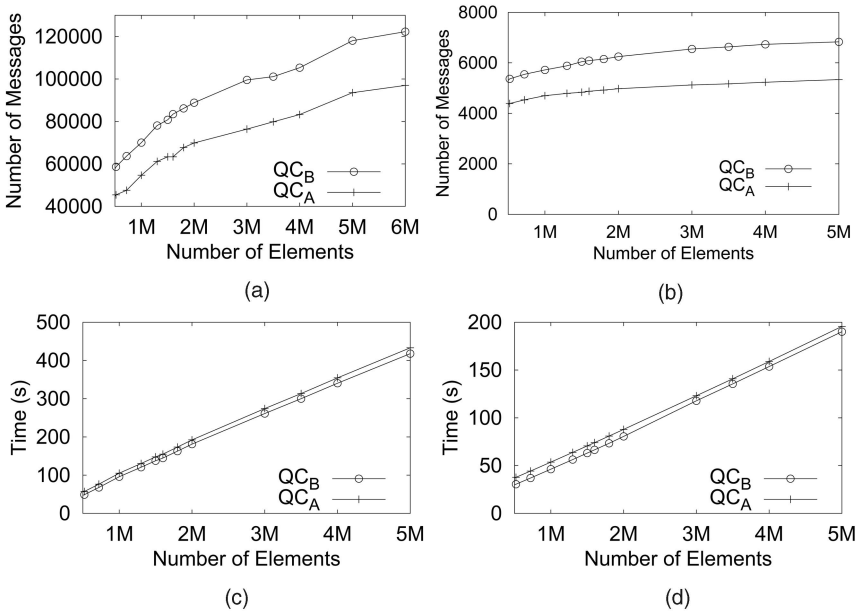

Fig. 8. Clustering algorithm comparison. (a) Semisort CASU. (b) Uniform CASU. (c) Semisort CASU. (d) Uniform CASU.

the data streams, as well as the total processing time. The experiment results, reported in Fig. 8, demonstrate that CASU based on $\mathrm{QC_A}$ leads to very similar processing time to that by CASU based on $\mathrm{QC_B}$. However, $\mathrm{QC_B}$ causes more trigger fires; thus, more times query clusters are reprocessed and a greater number of messages sent out. Therefore, we will use $\mathrm{QC_A}$ as the offline query clustering algorithm, thereafter, to illustrate our performance evaluation results in combining with CASU and CSU.

5.3 Evaluation of Continuous Processing Techniques

We evaluate the synchronous update technique (Algorithm 3) and the asynchronous update technique (Algorithm 4). To discount the great impact by our clustering techniques, we do not apply them.

In this set of experiments, the precision guarantee ϵ of a summary is assigned to 5×10^{-3}. The number N_q of queries is assigned the default value. The $1M$ queries are randomly generated with the precisions $e_q \in [5 \times 10^{-3}, 2 \times 10^{-2}]$ and are enforced without any query interval containment in order to apply Algorithms 3 and 4, where mutual containment is disallowed.

Three data streams are used; each has 5,000,000 data elements. In the three sets, the data distributions (D_d) are uniform, normal, and semisort, respectively. We assume that the query set is pregiven and the system continuously runs the query processing SU and CASU (but without clustering). Since *SU is too slow to finish*, we do not present its evaluation results; it will not be further evaluated in our performance study. In our experiment, we want to report the processing time involved (query processing and summary maintenance) per data item. As such time is too short to be recorded precisely, we record the total processing time for every batch of 1,000 elements, and then divide it by 1,000; this average time is used as the processing time "per data item." The experiment results reported in Fig. 9 demonstrated that our asynchronous technique can process a quite rapid stream with an average arrival rate at about 1,000 element per second against 1M unclustered continuous queries.

In the experiment results, the time per item in the initial part is significantly higher than that of later arrival element. This is because, with a small volume of stream, triggers are fired more frequently and, thus, the query reprocessing costs are higher.

5.4 Scalability against Streams

Now, we evaluate the scalability of CSU and CASU against the whole data stream model regarding data stream sizes and arrival rates.

In this set of experiments, the precision guarantee ϵ of a summary is assigned to 5×10^{-3}. The set of queries is generated in a similar way as the uniform query set in the first experiment in Section 5.1, except that we now limit ϵ_q in the interval $[5 \times 10^{-3}, 2 \times 10^{-2}]$. N_q is 2,000,000.

The three data streams in the experiments conducted in the last section are used. We record the processing time per data item in the same way as that in the last section. The experiment results, reported in Fig. 10a, Fig. 10b, and Fig. 10c, show that CASU can process a very rapid data stream in real time with a stream arrival rate, more than 10K elements/second on average. Although CSU is much slower than CASU, it still can support rapid data streams

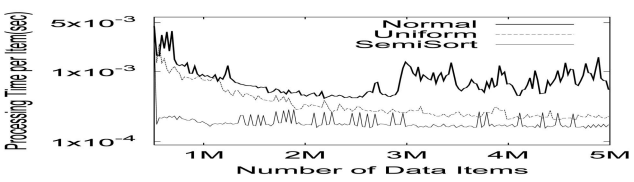

Fig. 9. Asynchronous update.

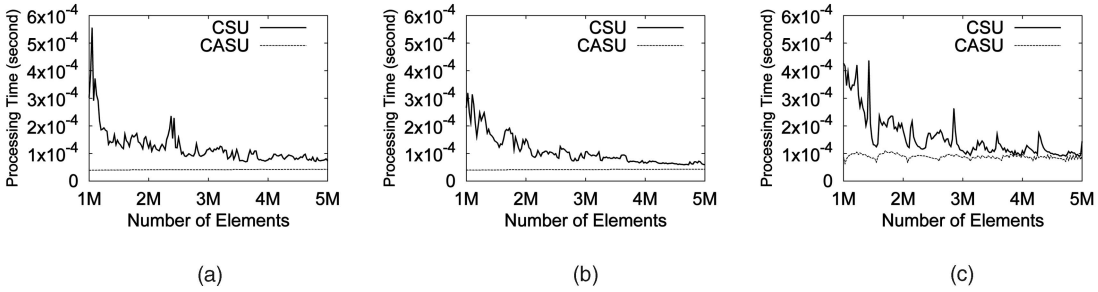

Fig. 10. 2M Predefined queries on 5M data elements. (a) Uniform data. (b) Normal data. (c) Semisort data.

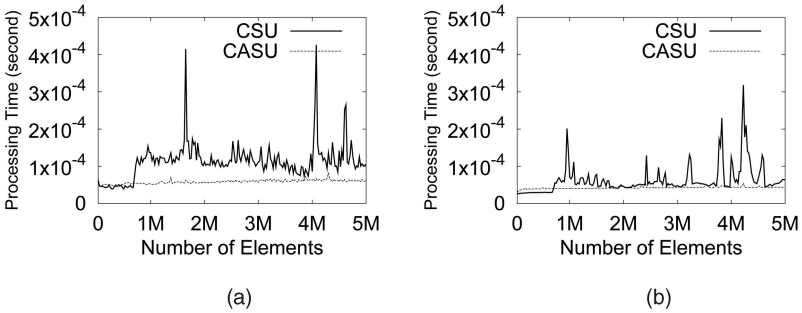

Fig. 11. 2M dynamically issued queries. (a) Semisort data. (b) Normal data.

with an arrival rate more than 1,000 elements/second on average.

In the performance study, the semisort data set performs slightly worse in contrast to the results in Section 5.3. This is due to 1) query processing costs are very dominant without query clustering, 2) a summary of semisort data gives more tuples and is more expensive (than others) to continuously maintained, and 3) continuous query processing costs against a summary of semisort data are less expensive than others.

5.5 Evaluation of Query Dynamic Behavior

Suppose that a summary maintained in the experiments is 0.005-approximate. We evaluate the performance of CSU and CASU against a stream of queries that are dynamically registered and removed. A stream of queries follow two models: 1) insertions only and 2) insertions and deletions.

In the first set of our experiments, the two data sets are used 5M semisort data set and 5M normal data set. A set of 2M queries are used in order to have a reasonably frequent updates (insertion or deletion) of query clusters; it is generated in a similar way as the uniform query set in the first experiment in Section 5.1 except that ϵ_q is in the interval $[5 \times 10^{-3}, 2 \times 10^{-2}]$. The arrival of a query is randomly assigned among the 5M data elements.

We measure the total processing time per data item e, (i.e., the time of maintaining the summary, the time spent on OQC, and the time spent on processing queries after e arrives but before the next item arrives), in the same way as in Section 5.3. The experiment results, illustrated in Fig. 11a and Fig. 11b, also demonstrate that CASU can support a very rapid data stream with stream arrival rate at least 10K items/second.

In the second set of experiments, we do a performance study on a possible impact of dynamic deletions and insertions of queries. The four sets of queries are generated in a similar way to those for the second experiment in Section 5.1, except ϵ_q is in the interval $[5 \times 10^{-3}, 2 \times 10^{-2}]$. They are also named uni10, uni20, zipf10, and zip20, respectively. Note that each query set has 2M queries. We assume that a batch of queries are issued simultaneously and the arrival time of each batch is randomly chosen to span the whole data set. Before a query batch is issued, the queries in the previous batch are gradually removed in a random order, like a "sliding window." The data set used in this experiment is a semisort data set with 2M data elements. In our experiment, we record the total computation time and the total message numbers (i.e., the number of total times to reprocess clusters) for result notification. The experiment results are reported in Fig. 12. It shows that CASU is not very sensitive to different dynamic behavior of queries regarding the computation costs.

5.6 Evaluation of Sliding Window Techniques

In this section, we evaluate our technique (CSU) for processing continuous quantile queries for sliding window data stream model. We use default parameter settings for queries as described in Table 1 except ϵ.

We first examine the possible impacts of different data distributions. Three data sets are generated with uniform, normal, and semisort distributions, respectively. We fix the window size to be 500K and ϵ to be 0.005. We record the total time in our experiment for each data set. In the results depicted in Fig. 13, the dark shadow parts illustrate the costs (time) spent on maintaining quantile summaries.

In the second experiment, we evaluate the performance against different window sizes and ϵ. A window size N is chosen from one of $\{200K,\ 600K, 1M\}$, while ϵ is chosen

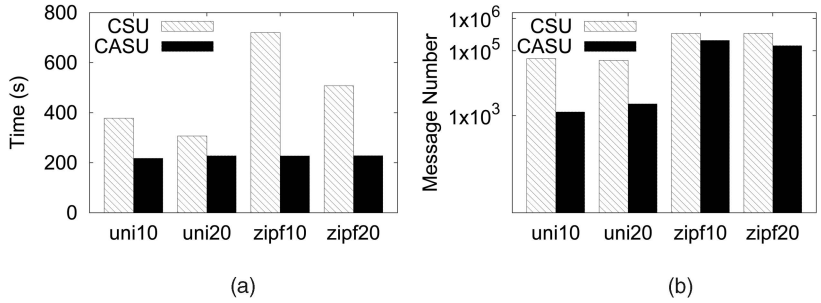

Fig. 12. Query stream acting as a sliding window. (a) Time consumption. (b) Message number.

from one of $\{0.005, 0.01, 0.02\}$. We record the total time for processing queries against the data set under a sliding window model. The data set is chosen using the default setting values in Table 1. The experiment results, reported in Fig. 14, show that the smaller a window size is, the more computation time is needed. This is because, in our technique, the trigger value for CSU is proportional to the window size. The smaller the window size is, the more times CSU is invoked.

5.7 Evaluation for Real Data

In this section, we do the performance study of CSU and CASU against a real data set downloaded from (http://www.ldc.upenn.edu/Projects/TDT3/). We have archived the news stories received through Reuters real-time datafeed, which contains 365,288 news stories and 100,672,866 (duplicate) words. All words such as "the" and "a" are removed before term stemming. We have done similar experiments as those in the second set of experiments in Section 5.5, but replace them with this new data set. In the experiment, we report the total computation time, which is illustrated in Fig. 15.

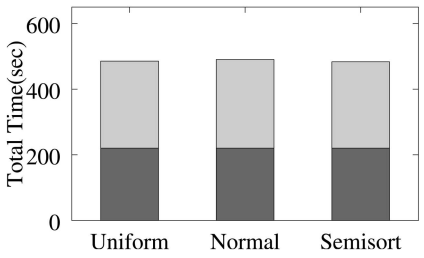

Fig. 13. Time consumption against different data.

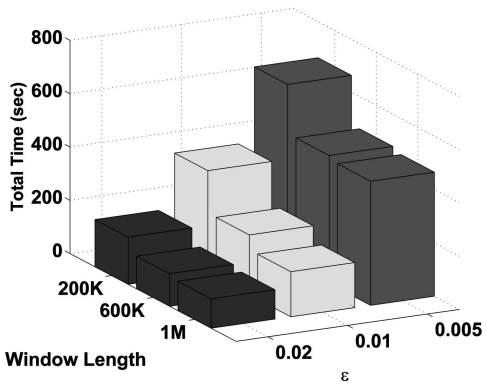

Fig. 14. Time against different settings.

As a short summary, our comprehensive performance study clearly demonstrates that the proposed query clustering techniques and query processing techniques are both effective and efficient. Under the constraint of meeting users' precision requirements, CSU and CASU can both support real-time processing of rapid streams and the CASU is a much better choice.

6 Conclusions and Remarks

In this paper, we presented our techniques for online processing of massive continuous quantile queries over data streams. While many research results on continuous queries and data stream computation have been recently reported in the literature, the research in this paper is among the first attempts to develop scalable techniques to deal with massive query streams and data streams. Our query processing techniques are not only efficient and scalable, but also can guarantee the query precision requirements. Further, the techniques are applicable to both whole streams and sliding windows. Our experiment results demonstrated that the techniques are able to support online processing of massive queries over very rapid data streams.

Note that our techniques already cover a "composite" quantile query (i.e., a set of quantile queries); in this case, a composite quantile query is just decomposed into a set of single quantile queries. The techniques presented in this paper against whole data streams may be immediately applied to any ϵ-approximate quantile summary that follows Definition 1 and the three conditions in Theorem 1. Our sliding window techniques may also be extended to support the deterministic quantile summary technique in [1] by applying our synchronous update technique for sliding windows to each level. Very recently, Cormode et al. [6] develop a framework for computing biased quantiles over data streams. Our techniques are not immediately

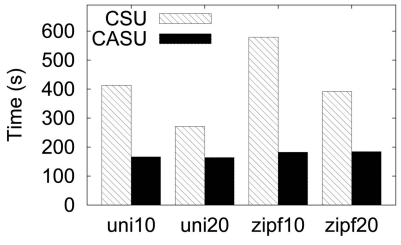

Fig. 15. Time consumption for NewsData.

applicable to such a biased quantile computation problem. As possible future work, we will investigate this.

ACKNOWLEDGMENTS

This research was conducted when J. Xu and Q. Zhang were with the University of New South Wales. It was partially supported by an ARC Discovery Grant (DP0346004).

REFERENCES

[1] A. Arasu and G.S. Manku, "Approximate Counts and Quantiles over Sliding Windows," *Proc. ACM Symp. Principles of Database Systems,* pp. 286-296, 2004.
[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," *Proc. ACM Symp. Principles of Database Systems,* pp. 1-16, 2002.
[3] J. Chen, D.J. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for internet Databases," *Proc. SIGMOD Conf.,* pp. 379-390, 2000.
[4] Y. Chen, G. Dong, J. Han, B.W. Wah, and J. Wang, "Multi-Dimensional Regression Analysis of Time-Series Data Streams," *Proc. Conf. Very Large Databases,* pp. 323-334, 2002.
[5] T. Cormen, C.S.C. Leiserson, and R. Rivest, *Introduction to Algorithms.* The MIT Press, 2001.
[6] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Effective Computation of Biased Quantiles over Data Streams," *Proc. Int'l Conf. Data Eng.,* pp. 20-31, 2005.
[7] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding Hierarchical Heavy Hitters in Data Streams," *Proc. Conf. Very Large Databases,* pp. 464-475, 2003.
[8] G. Cormode and S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and Its Applications," *J. Algorithms,* vol. 55, no. 1, 2004.
[9] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk, "Mining Database Structure; or, How to Build a Data Quality Browser," *Proc. SIGMOD Conf.,* pp. 240-251, 2002.
[10] A. Dobra, M.N. Garofalakis, J. Gehrke, and R. Rastogi, "Processing Complex Aggregate Queries over Data Streams," *Proc. SIGMOD Conf.,* pp. 61-72, 2002.
[11] M. Garofalakis and A. Kumar, "Correlating XML Data Streams Using Tree-Edit Distance Embeddings," *Proc. ACM Symp. Principles of Database Systems,* pp. 143-154, 2003.
[12] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "How to Summarize the Universe: Dynamic Maintenance of Quantiles," *Proc. Conf. Very Large Databases,* pp. 454-465, 2002.
[13] M. Greenwald and S. Khanna, "Space-Efficient Online Computation of Quantile Summaries," *Proc. SIGMOD Conf.,* 2001.
[14] S. Guha, N. Koudas, and K. Shim, "Data-Streams and Histograms," *Proc. Ann. ACM Symp. Theory of Computing,* pp. 471-475, 2001.
[15] A. Gupta and D. Suciu, "Stream Processing of Xpath Queries with Predicates," *Proc. SIGMOD Conf.,* pp. 419-430, 2003.
[16] M. Hammad, M. Franklin, W. Aref, and A. Elmagarmid, "Scheduling for Shared Window Joins over Data Streams," *Proc. Conf. Very Large Databases,* pp. 297-308, 2003.
[17] S. Krishnamurthy, S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S.R. Madden, F. Reiss, and M.A. Shah, "Telegraphcq: An Architectural Status Report," *IEEE Data Eng. Bull.,* 2003.
[18] H. Kung, F. Luccio, and F. Preparata, "On Finding the Maximum of a Set of Vectors," *J. ACM,* vol. 22, no. 4, pp. 469-476, 1975.
[19] X. Lin, H. Lu, J. Xu, and J.X. Yu, "Continuously Maintaining Quantile Summaries of the Most Recent n Elements over a Data Stream," *Proc. Int'l Conf. Data Eng.,* pp. 362-374, 2004.
[20] T. Linsmeier and N. Pearson, "Risk Measurement: An Introduction to Value at Risk," Working Paper 96-04, Office for Futures and Options Research (OFOR), 1996.
[21] L. Liu, C. Pu, and W. Tang, "Continual Queries for Internet Scale Event-Driven information Delivery," *IEEE Trans, Knowledge and Data Eng.,* vol. 11, no. 4, pp. 610-628, 1999.
[22] S. Madden and M.J. Franklin, "Fjording the Stream: An Architecture for Queries over Streaming Sensor Data," *Proc. Int'l Conf. Data Eng.,* pp. 555-566, 2002.
[23] S. Manganelli and R. Engle, "Value at Risk Models in Finance," European Central Bank Working Paper Series No. 75, 2001.
[24] G. Manku and R. Motwani, "Approximate Frequency Counts over Data Streams," *Proc. Conf. Very Large Databases,* pp. 346-357, 2002.
[25] G.S. Manku, S. Rajagopalan, and B.G. Lindsay, "Approximate Medians and Other Quantiles in One Pass and with Limited Memory," *Proc. SIGMOD Conf.,* pp. 426-435, 1998.
[26] G.S. Manku, S. Rajagopalan, and B.G. Lindsay, "Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Data Sets," *Proc. 1999 ACM SIGMOD Int'l Conf. Management of Data,* pp. 251-262, 1999.
[27] J.I. Munro and M.S. Paterson, "Selection and Sorting with Limited Storage," *Theoretical Computer Science,* 1980.
[28] F. Nielsen, "Fast Stabbing of Boxes in High Dimensions," *Theoretical Computer Science,* 2000.
[29] C. Olston, J. Jiang, and J. Widom, "Adaptive Filters for Continuous Queries over Distributed Data Streams," *Proc. SIGMOD Conf.,* pp. 563-574, 2003.
[30] V. Poosala, P. Haas, Y. Ioannidis, and E. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates," *Proc. SIGMOD Conf.,* pp. 294-305, 1996.
[31] V. Poosala and Y. Ioannidis, "Estimation of Query-Result Distribution and Its Application in Parallel-Join Load Balancing," *Proc. Conf. Very Large Databases,* pp. 448-459, 1996.
[32] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction.* Springer-Verlag, 1985.
[33] Version2.0 Financial Analysis Package MT for Gauss™—Version 2.0, Aptech System, Inc., 2003.
[34] S. Viglas, J.F. Naughton, and J. Burger, "Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources," *Proc. Conf. Very Large Databases,* pp. 285-296, 2003.
[35] Y. Zhu and D. Shasha, "Statstream: Statistical Monitoring of Thousands of Data Streams in Real Time," *Proc. Conf. Very Large Databases,* pp. 358-369, 2002.
[36] G.K. Zipf, *Human Behaviour and he Principle of Least Effort.* Addison-Wesley, 1949.

Xuemin Lin received the BSc degree in applied math from Fudan University in 1984 and the PhD degree in computer science from the University of Queensland in 1992. He is an associate professor in the School of Computer Science and Engineering, the University of New South Wales (UNSW). He has been the head of database research group at UNSW since 2002. Before joining UNSW, he held various academic positions at the University of Queensland and the University of Western Australia. During the period of 1984-1988, he studied for the PhD degree in applied math at Fudan University. His current research interests lie in data streams, approximate query processing, spatial data analysis, and graph visualization.

Jian Xu received the BSc degree from Nanjing University of Aeronautics and Astronautics, China, in 2002. He joined the database group at the University of New South Wales in 2002 and received the MSc degree in 2004. Now, he is a graduate student in the Computer Science Department, University of British Columbia. His research interests include data streams, data mining, distributed system, and algorithms.

Qing Zhang received the BS degree in applied physics from Tsinghua University, China, and the PhD degree in computer science from the University of New South Wales, Australia, in 2005. Presently, he is a postdoctoral fellow at e-Health research center/CSIRO ICT center, Australia. His research interests are approximate query processing, data stream, and query processing in multidatabase.

Hongjun Lu received the BSc degree from Tsinghua University, China, and the MSc and PhD degrees from the Department of Computer Science, University of Wisconsin-Madison. Before joining the Hong Kong University of Science and Technology, he worked as an engineer at the Chinese Academy of Space Technology, as a principal research scientist in the Computer Science Center of Honeywell inc., Minnesota, (1985-1987), and as a professor in the School of Computing at the National University of Singapore (1987-2000). His research interests were in data/knowledge base management systems with an emphasis on query processing and optimization, physical database design, and database performance. His research work included data quality, data warehousing and data mining, and management of XML data. He was also interested in development of Internet-based database applications and electronic business systems. Dr. Lu was a trustee of the VLDB Endowment, an associate editor of the *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, and a member of review board of the *Journal of Database Management*. He was the chair of the steering committee of the International Conference on Web-Age information Management (WAIM), and the chair of the steering committee of Pacific-Asia Conference of Knowledge Discovery and Data Mining (PAKDD). He served as a member of the ACM SIGMOD Advisory Board in 1998-2002.

Jeffrey Xu Yu received the BE, ME, and PhD degrees in computer science, from the University of Tsukuba, Japan, in 1985, 1987, and 1990, respectively. He was a research fellow (April 1990-March 1991) and a faculty member (April 1991-July 1992) at the Institute of information Sciences and Electronics, University of Tsukuba, and a lecturer in the Department of Computer Science, Australian National University (July 1992-June 2000). Currently, he is an associate professor in the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong. His major research interests include data mining, data streaming mining and processing, data warehouse, online analytical processing, XML query processing, and optimization. He is a member of the IEEE Computer Society.

Xiaofang Zhou received the BSc and MSc degrees in computer science from Nanjing University, China, in 1984 and 1987, respectively, and the PhD degree in computer science from the University of Queensland in 1994. He is currently a professor at the University of Queensland, Australia. He is the research director of the Australia Research Council (ARC) Research Network in Enterprise information infrastructure (EII), a chief investigator of ARC Centre in Bioinformatics, and a senior researcher of National ICT Australia (NICTA). His research interests include spatial information systems, high-performance query processing, Web information systems, multimedia databases, data mining, and bioinformatics.

Yidong Yuan received the BS and MS degrees in computer science from Shanghai Jiao Tong University, P.R. China, in 1998 and 2001. Currently, he is a PhD candidate in the School of Computer Science and Engineering, the University of New South Wales. His research interests include efficient query processing and query optimization for spatio-temporal database and data stream systems.