Filtering Data Streams for Entity-based Continuous Queries

Reynold Cheng, Ben Kao, Alan Kwan, Sunil Prabhakar, and Yicheng Tu

***Abstract*—The idea of allowing query users to relax their correctness requirements in order to improve performance of a data stream management system (e.g., location-based services and sensor networks) has been recently studied. By exploiting the maximum error (or *tolerance*) allowed in query answers, algorithms for reducing the use of system resources have been developed. In most of these works, however, query tolerance is expressed as a numerical value, which may be difficult to specify. We observe that in many situations, users may not be concerned with the actual value of an answer, but rather which object satisfies a query (e.g., "who is my nearest neighbor?"). In particular, an entity-based query returns only the names of objects that satisfy the query. For these queries, it is possible to specify a tolerance that is "non-value-based". In this paper, we study *fraction-based* tolerance, a type of non-value-based tolerance, where a user specifies the maximum fractions of a query answer that can be false positives and false negatives. We develop fraction-based tolerance for two major classes of entity-based queries: (1) non-rank-based-query (e.g., range queries) and (2) rank-based-query (e.g., k-nearest-neighbor queries). These definitions provide users with an alternative to specify the maximum tolerance allowed in their answers. We further investigate how these definitions can be exploited in a distributed stream environment. We design adaptive filter algorithms that allow updates be dropped conditionally at the data stream sources without affecting the overall query correctness. Extensive experimental results show that our protocols reduce the use of network and energy resources significantly.**

***Index Terms*—data streams, continuous queries, adaptive filters, fraction-based tolerance**

I. INTRODUCTION

Due to the rapid development of low-cost sensors and networking technologies, stream applications have attracted tremendous research interests lately. In particular, long-standing *continuous* queries are common in a stream environment for monitoring various network activities. Some examples include intrusion detection over security-sensitive regions; identification of Denial-of-Service (DOS) attacks on the Internet [4]; road traffic monitoring; natural habitat monitoring; network fault-detection; email spams detection; and web statistics collection.

In such applications, *stream sources* are installed to collect and report the states of various entities. A large number of sources (e.g., GPS-enabled devices, Internet hosts, wireless sensors) report their updated values (e.g., locations, TCP packets, and temperature values) continuously to the processing server. These systems often have limited bandwidth or energy resources. For example, in sensor networks, sensing devices usually have scarce battery power and communicate in a low-bandwidth environment. Moreover, since the number of stream sources could be large, a stream server could be crippled by the large volume of data. This slows its response to standing queries that require real-time processing [3]. It is thus important to lower the message volume so that transmission cost as well as server's load can be reduced.

Reynold Cheng, Ben Kao and Alan Kwan ({ckcheng,kao,klkwan}@cs.hku.hk) are with the Department of Computer Science, the University of Hong Kong, Pokfulam Road, Hong Kong.

Sunil Prabhakar (sunil@cs.purdue.edu) is with the Department of Computer Science, Purdue University, West Lafayette, IN 47907-1398, USA.

Yicheng Tu (ytu@cse.usf.edu) is with the Department of Computer Science and Engineering, University of South Florida, Tampa, FL 33620, USA.

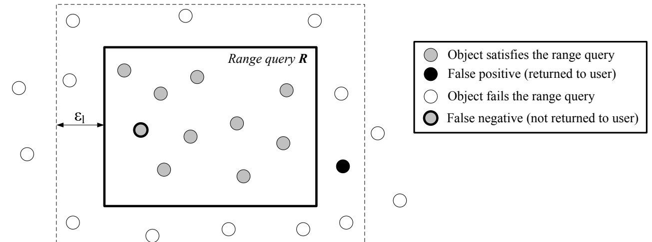


Fig. 1. Illustrating query tolerances.

One direct way of lowering transmission cost is to drop some of the data items generated from the stream sources. The drawback is that the server may have to process queries based on inaccurate data. However, if we can carefully select the items to be dropped, the accuracy of a query answer may only be affected slightly. Consider a transportation monitoring system, where vehicles within a geographical region (e.g., the city centre) can be tracked [35]. For vehicles that are stationary, or are located far away from the region, it may not be necessary for those vehicles to always report their positions to the system (in order to have high query accuracy). More generally, for many standing queries, a user may accept an answer with the maximum error allowed (or *tolerance*) in exchange for lower resource consumption and better timeliness in query processing. Other examples for which controlled query errors are acceptable include wide-area resource accounting and load balancing in replicated servers. Intelligent protocols have been proposed [28], [19] to wisely control when stream sources should report updates. The goal of the protocols is to reduce communication overhead while at the same time user-specified tolerances are met. These protocols make use of *filter bounds* — a system-specified range of values. A stream source only reports an update if its value crosses the bound.

To illustrate the above concepts, let us consider a *continuous range query*, a service commonly found in location-based and sensor applications. In this query, a user is interested in stream values that fall within a region called *query range*.

This query can be used in an intelligent transportation system, which monitors the movement of GPS-enabled vehicles inside a geographical region for an extensive amount of time. Another example is wildlife tracking, which allows real-time analysis of the movement of animals that gather around a landscape structure, e.g., a water hole [20]. The animals, attached with location sensors, can be monitored for a month's time by a continuous range query with the surrounding area of the water hole as a query range. Figure 1 illustrates a continuous range query over locations of moving objects. A rectangle R (in solid lines) is specified by the user. The query returns the identities of objects whose locations are inside R (these objects are colored in grey).

How would a user express a tolerance for this query? One possibility is to let the user choose a numerical tolerance, say ϵ, and the system guarantees that the ID of any object returned to the user must be located inside the dashed-line rectangle (but not necessarily inside R). This kind of tolerance, expressed as a *numerical value*, is often assumed by filter-bound-based approximation techniques. In Figure 1, the filter bound is exactly the dashed-line rectangle, and is installed in each moving object. This means that even if an object is outside R,it needs not report its location to the system as long as it does not cross the filter bound.

While numerical tolerance is useful, choosing an appropriate value of it may not be straightforward. In particular, specifying a numerical tolerance requires some knowledge about the relative distances or spread of the objects. For instance, should ϵ be one meter or 100 meters? In a sensor network, various kinds of data such as humidity, temperature, and UV-index are collected [12]. If only a numerical tolerance is allowed, the user may need to know a reasonable range of error for each data type. Also, if a data stream contains multi-dimensional data (e.g., location, speed) or multimedia data (e.g., images), a numerical, or *value-based error*, could be difficult to specify. A bad choice of the numerical tolerance may significantly weaken the value of a query. From Figure 1, we can see that a large number of objects whose locations are outside R (i.e., those colored in black and white) are also included in the answer. To solve this problem, a user has to be careful not to set ϵ too large. However, if ϵ is too small, it may not be very useful for improving the system performance. Thus, finding a reasonable value of ϵ can be difficult.

Alternatively, query tolerance can be expressed in terms of a *fraction* rather than an absolute value. To illustrate, Figure 1 shows an object, colored in black, that is included in the answer although the object is not inside R. This object is called a *false positive* [27]. A *fraction-based tolerance* can then be defined as the maximum fraction of query answers that can be false positives. For example, if the fraction-based tolerance is 0.1, then the black-colored object can still be included in the answer, which contains nine other objects. Another possibility to express the fraction-based tolerance is through the use of *false negatives* [27], which specifies the maximum fraction of objects that belong to the query result but are not included in the answer returned to the user. An example of a false negative is illustrated as a circle with a thick boundary in Figure 1; although it is inside R, it is not included in the user's result. Notice that the concepts of false positives/negatives are based on *precision* and *recall* in the IR literature [17].

The main goal of this paper is to study how filter bounds can be deployed in data stream sources in order to exploit fraction-based tolerance for different continuous query types. Let us use the range query example in Figure 1 again to illustrate our solutions. Based on the maximum fraction of false positives allowed, we first compute the number m of stream sources that can be false positives. Then, m stream sources that are currently satisfying the query are requested to stop sending their data to the system. For the remaining streams that satisfy the query, they are assigned with the filter bound with the range R. Notice that regardless of whether the values of the m "shut-down" stream sources satisfy the range query, the query answer is still acceptable within the tolerance. Hence, as long as no updates are received by the system, the query answer remains correct with respect to the tolerance. Similarly, a maximum number of false negative stream sources are stopped from reporting updates. Since only the stream sources with the filter bound R send their updates when their values cross the bound, our protocol saves network and energy costs of data transmission.

Besides range query, we also study filter bound protocols for a *rank-based query*, another important query type in streaming applications. Contrast to a range query, a ranked-based query returns IDs of objects based on their relative rankings. For example, in a transportation system, a long-standing k-nearest-neighbor (k-NN) query can be issued, which continuously returns the IDs of k vehicles closest to a given query point [18] for a long period of time. In a habitat monitoring system, scientists may be interested in tracking the k areas that yield the highest temperature, in which case a top-k query can be used [12]. As another example, in network traffic analysis, it is important to identify heavy hitters [5], [15]. A heavy hitter is an IP source that delivers a large number of packets to the monitored network, which is also likely involved in Denial-Of-Service (DOS) attacks. If the user is interested in monitoring the k sources that yield the largest traffic volume, then a continuous top-k query can be used. Notice that both the range query (a *non-rank-based query*) and the k-NN/top-k query (a *rank-based query*) return sets of object IDs, rather than numerical values as answers. These *entity-based* queries are good candidates for using fraction-based tolerance, which does not use numerical values.

A few technical challenges need to be addressed for filter bound protocols that exploit fraction-based tolerance. First, as we will show, when value updates are received from stream sources, the tolerance requirement can be violated (e.g., the fraction of false positives is larger than the allowed threshold). We tackle this problem by carefully adjusting the filter bounds of the stream sources, so that query correctness can be restored. Second, this correctness maintenance process, which involves several message exchanges between the server and the stream sources, can be expensive. Moreover, some stream sources that are considered as false positives/negatives and are not involved in data transmission may have to be "waken up" to send data in order to maintain query correctness. We propose two new techniques, namely *incremental deploy-*

ment and *immediate compensation*, in order to address these problems. The incremental deployment technique reduces the chance of expensive error fixing by only allowing a fraction of false positive/negative stream sources to be "shut down". The immediate compensation method is designed to force an active data stream source to stop transmitting data, in order to compensate the "waking-up" of another stream source source.

While most previous works in filter bound algorithms assume *value-based* queries (e.g., SUM and AVG), we study the *entity-based* queries. We investigate the computation and communication costs of our protocols. We also perform simulations to examine the effectiveness of our protocols. Although we assume one-dimensional data here, our techniques can be generalized to higher-dimension cases. To summarize, our contributions are:

- Define fraction-based tolerance for rank-based and non-rank-based queries;
- Present protocols that exploit fraction-based tolerances;
- Derive the computation, communication and energy costs of the protocols; and
- Perform extensive experiments on the protocols, using both real and synthetic data.

The rest of this paper is organized as follows. We discuss related work in Section II. Section III defines the semantics of fraction-based tolerance constraints. Section IV presents protocols for maintaining filter constraints for fraction-based tolerance of non-rank-based queries. Then Section V explains how the non-rank-based query protocol can be extended to support rank-based query. In Section VI we analyze the resource consumption for our protocols and present our experimental results. We conclude the paper in Section VII.

II. RELATED WORKS

Tolerance classification. Due to the high-volume and continuous nature of data streams, an important goal of a stream management system is to conserve system resources such as battery [13], memory [3], computation [22], [29], [16] and communication costs [14], [28], [19]. Most of these works reduce resource consumption by relaxing correctness requirements. Typically, a user specifies a maximum tolerance, and the tolerance is exploited by various techniques such as approximate data structures, load shedding, filters etc. The tolerance is often assumed to be in the form of a numerical value. Also, they are mostly applicable to value-based queries only. Our work investigates the possibility of exploiting fraction-based tolerance, a type of non-value-based tolerance, for continuous entity-based queries.

Figure 2 illustrates a classification of tolerances. The rank-based tolerance, which describes the error in terms of the distance from the desired order to the actual answer, is studied in [7]. For example, given that the query returns the object with the largest value, the user can specify that he can accept an object which ranks the second or the third. In this paper, we focus on fraction-based tolerance (shaded in Figure 2). We can see that fraction-based tolerance has a broader application than rank-based tolerance, since it can be applied to both rank- and non-rank-based queries.

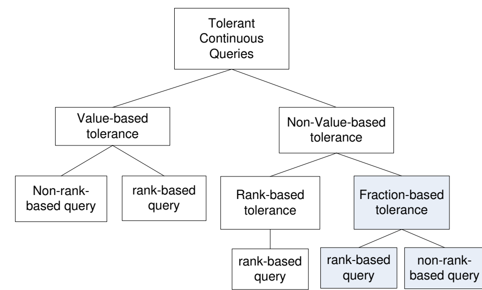


Fig. 2. Classification of Query Tolerances.

Adaptive filters. The idea of using adaptive filters in which filter bounds are installed to reduce communication costs was first proposed in [28]. However, that paper only considers value-based tolerance over aggregate queries such as average value and minimum value. In [4], a similar idea is applied to answer top-k queries for distributed stream sources, but again the tolerance is value-based. In [19], Kalman Filters are used to exploit value-based tolerance. The Kalman Filter is installed at every stream source, and with its prediction techniques it is shown to be more effective than previous methods in conserving communication costs. The extension of adaptive filters in a sensor network is studied in [11]. Our work differs from theirs in that we use adaptive filters to exploit non-value-based tolerance. In addition, we study continuous k-NN queries that are used in applications such as computer aided manufacturing and traffic monitoring [22]. Notice that k-NN queries are more general than top-k queries studied in [19]. Adaptive filters for k-NN queries are also studied in [26], but the use of non-value-based tolerance is again not considered.

Non-value-based tolerance. The classification of queries into value-based and entity-based has been proposed in [6]. To our best knowledge, the use of non-value-based tolerance for entity-based queries has not been well-studied. In [34], approximate answers for set-valued queries are proposed, where a query answer contains a set of objects. An exact answer E is approximated by two sets: a *certain set* C which is a subset of E, and a *possible set* P such that $C \cup P$ is a superset of E. This notion can be used to generate approximate query results if a portion of the relational database is unavailable, or if there is not enough time to produce an exact answer. A rank precision model is proposed in [21]: an answer a is called α-precise if the true rank of a lies in the interval $[r-\alpha, r+\alpha]$, where r is the rank of a informed to the user. In [10], precision and recall are used as quality metric for approximating k-NN queries. The issues of defining and exploiting fraction-based tolerance in stream systems has only been addressed in [7]. However, that work uses a preliminary protocol to handle the tolerance. Here we propose a more efficient protocol, which performs better than that in [7]. We also examine energy consumption issues, which have not been studied before.

Other query evaluation techniques. The idea of viewing a k-NN query as a range query was proposed in [18]. They use a bound which encloses at least k objects so that continuous k-NN queries can be answered efficiently. For our filter bound protocol for continuous k-NN query, we also convert the query to a range query.

In [25], a cost-based algorithm for MAX/MIN query was proposed for sensor environments. The algorithm provides a provably-smallest cost in probing the data from the stream sources. Their method is only used for "snapshot queries" (i.e., queries that are executed once only), whereas our approach (which uses adaptive filters) is designed for continuous queries. We also study the effectiveness in energy consumption, which is not studied in that paper.

Another cost-based approach for sensor network monitoring was recently studied in [30]. Their energy consumption model used for optimization is described in terms of message cost. Specifically, for every x-byte message sent, $\sigma + \delta x$ units of energy will be consumed, where δx is the energy spent for sending x bytes, and σ is a per-message overhead. In our paper, we further exploit the fact that sensors can be in "sleep mode", which consumes a minimal amount of energy. By using fraction-based tolerance, we show that it is possible to save energy by allowing a controlled portion of stream sources to be in the sleep mode.

Load shedding In this method, the database server drops data tuples that arrived at the server site, in order to lower the resource demand. Query processing also has to be accomplished under some Quality-of-Service (QoS) requirements [31], [32], [8], [1]. Although filtering techniques also attempt to reduce resource utilization by dropping tuples, they are generally different from load shedding. For filtering, the dropping of tuples occurs in stream sources rather than in the server. Hence, filtering can save more communication costs.

III. Problem Definition

We assume a distributed stream management model similar to those described in [4], [28], [19]. The system consists of a set $S = \{S_1, \ldots, S_i, \ldots, S_n\}$ of n data stream sources with stream source S_i reporting a value $v_i \in \Re$. We assume that stream values are updated at discrete time instants. Each stream source may be associated with an *adaptive filter* that specifies a *constraint*. With the filter mechanism, not all updates are reported to the server. A filter constraint is a closed interval $[l_i, u_i]$, where $l_i, u_i \in \Re$. Let v'_i be the last reported value from stream source S_i. When the stream's value (v_i) changes, the filter constraint is *violated* if either (1) $v'_i \in [l_i, u_i] \wedge v_i \notin [l_i, u_i]$ or (2) $v'_i \notin [l_i, u_i] \wedge v_i \in [l_i, u_i]$. Only when the constraint is violated will the updated value be sent to the server. If no filter is installed at a stream source, all updates from the stream source are reported.

Figure 3 shows a general architecture of such systems. Each stream source is equipped with a filter that is *adaptive* whose parameters can be changed at any time by the processor. A user submits her queries and tolerance requirements to the central processor. The constraint assignment unit then determines the relevant filter constraints to be installed in each stream source. The query processing unit processes user queries and updates their results if necessary. It also receives updates from the stream sources. It communicates with the constraint assignment unit, which decides if constraints need to be revised for relevant filters.

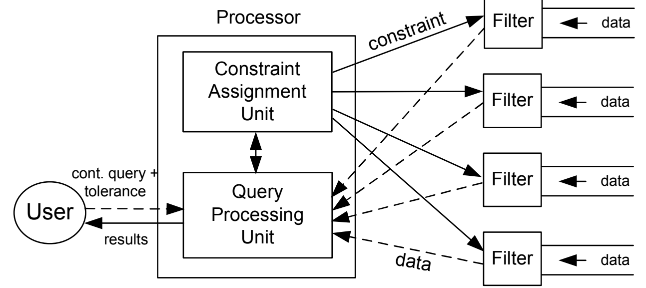


Fig. 3. Data Stream Management System Model.

A. Query Model

We are interested in *entity-based queries* – those that return identifiers of objects as answers [6]. We classify entity-based queries into *rank-based* queries and *range* queries:

(1) Rank-based query. Given a number $k \in \aleph$ (k is called the *rank requirement*), a rank-based query returns IDs of objects that rank kth or above. Here we use k-NN queries to illustrate filter protocols, since such queries are common in systems like computer aided manufacturing (CAM) in a product-line monitoring system, mobile environments, and network traffic monitoring [18], [22], [10]. A CAM, for example, uses k-NN queries to discover similar patterns over multi-dimensional data obtained from sensors installed in production lines. A k-NN query can also answer k-min and k-max queries. Notice that a k-min (k-max) query is just a k-NN query by setting the query point q to $-\infty$ (respectively $+\infty$).

(2) A **range query** is specified by an interval $[l, u]$. Streams whose values fall within $[l, u]$ are returned to the user. A range query is non-rank-based since the decision of whether a stream contributes an answer is independent of others.

Here we use Q to denote an entity-based standing query and $A(t)$ to denote the answer set returned at time t. We use $|A(t)|$ to denote the cardinality of $A(t)$.

A standing query Q is associated with a *tolerance constraint*. We focus on the fraction-based tolerance, a kind of non-value-based tolerance. The remaining of this section presents and explains the definition of this tolerance.

B. Fraction-based Tolerance

As explained, fraction-based tolerance adopts the concept of false positives and negatives. This tolerance applies to all entity-based queries, i.e., both rank- and non-rank-based queries. An example of fraction-based tolerance for non-rank-based queries is the reporting of alert messages about network sources, which yield a traffic volume within an abnormal range. For security purposes, it may be acceptable that these messages are sent to the network administrator even if the message is a false alarm. The network source wrongly reported can then be regarded as a *false positive*. As for rank-based queries, consider a computer-aided manufacturing (CAM) system [22], where feedback information is returned from production lines to adjust the parameters. In these systems, sensors are installed to monitor the parts and discover the patterns of similar features. Users issue the k-NN queries to mine multimedia data streams (e.g., images) for unknown patterns, where the features are multi-dimensional such as

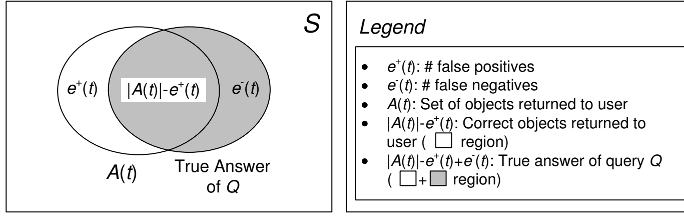


Fig. 4. Illustrating $A(t)$, $e^+(t)$ and $e^-(t)$.

position, shape, size, surface characterization, and material properties, etc. Those features are difficult to specify based on a numerical tolerance (because the user may not have a sense of how much the error value should be, as discussed in Section I). A fraction-based tolerance, on the other hand, is more intuitive in quantifying the quality of results [10]. Based on the fraction-based tolerance, the system will trigger an alert if certain *percentage* of discrepancies in the result is reached.

Definition 1: **False Positive and False Negative**. Given query Q and answer set $A(t)$ returned to the user, let $e^+(t)$ denote the number of streams in $A(t)$ that fail Q, and $e^-(t)$ be the number of streams that satisfy Q but are not in $A(t)$. The *fraction of false positives* and the *fraction of false negatives* of Q at time t, denoted by $f^+(t)$ and $f^-(t)$ respectively are:

$$f^+(t) = \frac{e^+(t)}{|A(t)|} \quad (1)$$

$$f^-(t) = \frac{e^-(t)}{|A(t)| - e^+(t) + e^-(t)} \quad (2)$$

Equation 1 describes the portion of objects returned to the user that are not correct (false positives). Equation 2 is essentially the fraction of objects in the true query answer that are not returned to the user (i.e., false negatives). The denominator of Equation 2 (i.e., $|A(t)| - e^+(t) + e^-(t)$) is the size of the true answer set, where the false positives that do not satisfy Q are excluded and false negatives that satisfy Q but not in $A(t)$ are included. Figure 4 illustrates the relationship among these quantities. Notice that the notions of false positives/negatives are based on the concept of *precision* and *recall* [17]. In particular, $f^+(t) = 1 - \textit{precision}$, and our goal can be viewed as achieving the minimum precision required for query answers. Similarly, $f^-(t) = 1 - \textit{recall}$.

Definition 2: **Fraction-based Tolerance.** Given query Q, answer set $A(t)$, maximum false positive tolerance ϵ^+, and maximum false negative tolerance ϵ^-, the answer set $A(t)$ is correct w.r.t. ϵ^+ and ϵ^- iff $f^+(t) \leq \epsilon^+$ and $f^-(t) \leq \epsilon^-$.

The parameters ϵ^+ and ϵ^- are user-specified. The system has to guarantee that the fraction-based tolerances are met. We assume that ϵ^+ and ϵ^- are both smaller than 0.5, because in most scenarios, users are not interested in results with more wrong answers than correct ones. This assumption is also required for guaranteeing the correctness of our protocols.

Let $e^{max+}(t)$ be the maximum number of answers that can be incorrect in $A(t)$ and $e^{max-}(t)$ be the maximum number of stream sources that satisfy the query but are excluded from $A(t)$. From Equations 1 and 2, we have:

$$f^+(t) \leq \frac{e^{max+}(t)}{|A(t)|} = \epsilon^+, \quad (3)$$

$$f^-(t) \leq \frac{e^{max-}(t)}{|A(t)| - e^{max+}(t)} = \epsilon^-. \quad (4)$$

More about Fraction-based Tolerant k-NN queries. Note that each k-NN query has only k correct answers. Hence Equation 2 becomes

$$f^-(t) = \frac{e^-(t)}{k}, \quad (5)$$

which means that at any time t, the number of false negatives ($e^-(t)$) cannot exceed k. Moreover, the number of correct objects in the answer returned to the user, i.e., $|A(t)| - e^+(t)$, must not be larger than k. Equivalently, $1 - \frac{e^+(t)}{|A(t)|} \leq \frac{k}{|A(t)|}$. Since $\frac{e^+(t)}{|A(t)|} \leq \epsilon^+$ (Equation 1), we get:

$$1 - \epsilon^+ \leq 1 - \frac{e^+(t)}{|A(t)|} \leq \frac{k}{|A(t)|}. \quad (6)$$

$$|A(t)| \leq \frac{k}{1-\epsilon^+}, \quad (7)$$

$$|A(t)| \leq 2k. \quad (8)$$

Equation 8 is obtained by assuming that $\epsilon^+ < 0.5$. Hence, the size of the answer set may not be equal to k. For example, if the 10 nearest neighbors are queried with a fraction-based tolerance $\epsilon^+ = 0.1$, 11 streams can be returned, where at most one of them is not correct. (That is, all correct ones are returned.) In fact, the answer set size can be controlled by ϵ^+, and is upper-bounded by $2k$. Finally, since the true answer size is always k, we have:

$$|A(t)| \geq k(1-\epsilon^-) \quad (9)$$

$$|A(t)| \geq \frac{k}{2} \quad (10)$$

when ϵ^- is less than 0.5. Hence, a k-NN query answer is between $\frac{k}{2}$ and $2k$. We will explain how use this property in the protocol design later.

C. Maintaining Query Correctness

Our protocols translate tolerance constraints into filter constraints installed in the data stream sources. As long as the data value of a stream does not violate the filter constraint, no update is sent from the stream source to the server. When it is necessary that an update be sent to the server, the server may need to reconfigure the filter constraints. We call such reconfiguration *constraint resolution*. Similar to [4], there are two correctness requirements for our protocols:

Correctness Requirement 1: At every point in time, if no resolution is required, then the results of all running continuous queries remain valid within their tolerance constraints.

Correctness Requirement 2: Immediately after filter resolution is completed, the tolerance of a query is satisfied assuming that stream values do not change during resolution.

Next, let us study how fraction-based tolerance can be exploited for non-rank-based queries.

IV. Non-rank-based Queries

We now study how to exploit fraction-based tolerance for a range query – a non-rank-based query. Consider a protocol that use no tolerance: each stream filter is assigned the constraint $[l, u]$ at the beginning. Any violation in a filter has to be reported to the server, and query answers are updated correspondingly. Correctness is guaranteed, since essentially each filter evaluates the range query on the stream it is responsible for. We call this algorithm *zero-tolerance protocol for non-rank-based query* (**ZT-NRP**).

Although **ZT-NRP** can reduce communication costs, it may generate unnecessary updates. Consider Figure 1 again, where the maximum allowed fraction of false positives is 0.1. Suppose the black-colored object, previously inside the range R, has just moved outside. This triggers an update by ZT-NRP. This is not needed, since the black object is the only false positive out of the 10 answer objects. It can thus be included in the answer.

To solve the above problem, we propose a *fraction-tolerance protocol for non-rank-based query* (or **FT-NRP** in short). Section IV-A describes the framework of **FT-NRP**. Section IV-B investigates how tolerance can be restored if it is violated (by the arrival of data updates). We propose an enhanced protocol in Section IV-C. Section IV-D extends **FT-NRP** to handle multiple queries.

A. The FT-NRP framework

The **FT-NRP** protocol ensures that at any time during query execution, no more than a fraction ϵ^+ of query answers are false positives, and no more than a fraction ϵ^- of results are false negatives. As shown in Figure 5, FT-NRP consists of two phases: **Initialization** and **Maintenance** of filter constraints.

Initialization. To ensure that no more than a fraction ϵ^+ of the answer set (i.e., $A(t)$) can be wrong at any time t, the server first captures the states of the streams at time t_0 (Step 1). Then, $A(t_0)$ and $Y(t_0)$,the set of objects which do not belong to $A(t_0)$, are evaluated (Steps 2 and 3). Next, the algorithm broadcasts a filter bound $[l, u]$ to all the stream sources involved (Step 4). A subroutine called `calError` is invoked in Step 5, which computes the maximum number of false positives (e^{max+}) and false negatives (e^{max-}) allowed, without violating query correctness (refer to the bottom of Figure 5). According to Equation 3, $e^{max+}(t_0) = |A(t_0)| \cdot \epsilon^+$. Also, $e^{max-}(t_0) = |A(t_0)| \frac{\epsilon^-(1-\epsilon^+)}{1-\epsilon^-}$ (See [1].)

Next, let $n^+(t)$ be the number of stream sources allocated the *false positive* filter constraint (denoted by $[-\infty, \infty]$) at time t. Stream sources equipped with these filters do not report their values at all. Step 6 computes the value of $n^+(t)$, which is a fraction $(1-\omega)$ of the maximum false positives allowed (with $\omega \in [0, 1]$). Out of the $|A(t_0)|$ answers that satisfy the range query, we assign the $[-\infty, \infty]$ filter constraints to $n^+(t_0)$ of them (Step 7). Since $n^+(t)$ does not exceed the maximum false positives allowed, if no $[-\infty, \infty]$ stream sources reply, the false positive requirement is met i.e., $f^+(t) \leq \epsilon^+$. Moreover, as $n^+(t)$ stream sources are "shut down" from emitting updates, the amount of communication is reduced. As illustrated in our experimental results, this approach can also save battery power in a sensor network, since the sensors can be "shut down" and consume less energy than active sensors.

[1]From Equations 2, we have $\epsilon^- = \frac{e^{max-}(t_0)}{|A(t_0)|-e^{max+}(t_0)+e^{max-}(t_0)}$. By substituting $e^{max+}(t_0) = \epsilon^+|A(t_0)|$ (Equation 3) into ϵ^+, we get $e^{max-}(t_0) = |A(t_0)| \frac{\epsilon^-(1-\epsilon^+)}{1-\epsilon^-}$.

Initialization (at time t_0)
1. request all stream sources S_i to send their values
2. $A(t_0) \leftarrow \{S_i | v_i \in [l, u]$ at time $t_0\}$
3. $Y(t_0) \leftarrow S - A(t_0)$
4. Broadcast the $[l, u]$ filters to all stream sources
5. `calError(`$A(t_0)$`)`
6. $n^+ = (1-\omega)e^{max+}$ // ω controls no. of false +ve/-ve filters
7. For all stream sources in $A(t_0)$,
   (I) Select n^+ stream sources for installing the $[-\infty, \infty]$ filters
8. $n^- = (1-\omega)e^{max-}$
9. For stream sources in $Y(t_0)$,
   (I) Select n^- stream sources for installing the $[\infty, \infty]$ filters

Maintenance
Upon receiving a new update, v_i from stream source S_i,
1. **if** $v_i \in [l, u]$ **then**
   (I) insert S_i to $A(t)$
   (II)`calError(A(t))`
   (III) **if** $e^{max+} - n^+ > 0$ **then**
      (a) install $[-\infty, \infty]$ to S_i
      (b) $n^+ \leftarrow n^+ + 1$
2. **else**
   (I) remove S_i from A
   (II) `calError(A(t))`
   (III) **if** $\frac{n^+}{|A(t)|} > \epsilon^+$ **or** $\frac{n^-}{|A(t)|-n^++n^-} > \epsilon^-$ **then**
      (a) **execute** `fixError` // Restore correctness
   (IV) **else if** $e^{max-} - n^- > 0$ **then**
      (a) install $[\infty, \infty]$ to S_i
      (b) $n^- \leftarrow n^- + 1$

`calError(A)` // Find max no. of false +ve/-ves allowed
1. $e^{max+} = |A|\epsilon^+$
2. $e^{max-} = |A|\frac{\epsilon^-(1-\epsilon^+)}{1-\epsilon^-}$

Fig. 5. Maintaining fraction-based tolerance at the server.

In Step 6, ω is a system parameter that we call the *deployment fraction*. Intuitively, ω controls the number of false positive filters assigned to stream sources in the initialization phase. Why don't we assign these filters to all the $e^{max+}(t_0)$ and $e^{max-}(t_0)$ stream sources in Step 7(I)? The main reason is to provide more flexibility in choosing the appropriate number of stream sources to shut down. In particular, we may not know in advance which subset of the streams contributing to the answer $A(t_0)$ would be better associated with the $[-\infty, \infty]$ filters during the initialization phase. If a stream source is wrongly assigned a $[-\infty, \infty]$ filter (e.g., the values generated from that stream are constant), this filter is "wasted" since it does not save any potential update that crosses the $[l, u]$ bounds. On the other hand, for a stream whose value changes around the $[l, u]$ bounds, it is better to associate this stream source with a false positive/negative filter. The maintenance phase, explained next, allocates unused filters generated in the initialization phase to stream sources only when they signal to the system that their values have crossed the filter bounds. We term this method, which gradually allocates false positive/negative filters to the

stream sources, as **incremental deployment**. We will also explain later how incremental deployment can be beneficial to **FT-NRP**. We will also illustrate in the experimental results that it is often better to have a non-zero value of ω, rather than deploying the filters all at once.

False negative tolerance can be exploited in a similar way. Let $n^-(t)$ be the number of stream sources allocated the *false negative* filter constraint (denoted by $[\infty, \infty]$) at time t. Recall that $|Y(t_0)| = |S - A(t_0)|$ streams do not satisfy Q. By assigning $[\infty, \infty]$ filters to $n^-(t_0)$ of them, these stream sources are "turned off". Since $n^-(t) = (1-\omega)e^{max-}(t)$ (i.e., a fraction of the maximum number of false negative filters), if no data are received from $Y(t_0)$, we are guaranteed at any time t, $f^-(t) \leq \epsilon^-$ (Steps 8-9). Thus, after Initialization Phase, correctness requirement 1 is satisfied. That is, if no update is received at time t, $f^+(t) \leq \epsilon^+$ and $f^-(t) \leq \epsilon^-$.

Filter Selection. To choose which stream sources are assigned the false positive/negative filters in Steps 7(I) and 9(I). We propose two heuristics: (1) *random* – stream sources are randomly selected, and (2) *boundary-nearest* – only stream sources with values closest to the user-defined query range $[l, u]$ are considered. While *random* requires only $O(1)$ time, *boundary-nearest* is more expensive – it requires sorting the distance of the stream value from the query range, in $O(n \log(n))$ times. For *boundary-nearest*, however, objects closer to the query boundary has a higher chance for being assigned the false positive/negative filter constraints. These objects are also the ones that are likely to cross the boundaries and trigger updates. The assignment of filters to these objects can thus increase the chance that updates are dropped by the filters, resulting in better performance.

Maintenance. We now discuss how updates generated from stream sources with $[l, u]$ filters should be managed. Assume the server receives an updated value v_i from S_i at time t_u. Immediately prior to receiving v_i, according to correctness 1, the following must hold (using Equations 3,4):

$$f^+(t) \leq \frac{e^{max+}(t_u)}{|A(t_u)|} \leq \epsilon^+ \tag{11}$$

$$f^-(t) \leq \frac{e^{max-}(t_u)}{|A(t_u)| - e^{max+}(t_u)} \leq \epsilon^- \tag{12}$$

Let t be the current time instant with $t \geq t_u$. There are two different cases of updates to consider:

Case 1: $\boldsymbol{v_i \in [l, u]}$. This means S_i, previously not in the result, is now an answer. We handle this by inserting S_i into $A(t_u)$ (Step 1(I)). The number of false positives, $e^+(t)$, is unchanged. As $|A(t)|$ becomes $|A(t_u)| + 1$, $f^+(t)$ cannot be more than $\frac{e^{max+}(t_u)}{|A(t_u)|+1}$ (Equation 3), and is less than ϵ^+ (Equation 11). Since $e^-(t)$ is also unchanged, Equation 12 also holds. Thus correctness 2 is upheld.

The rest of Step 1 uses the false positive filters not deployed in initialization. Specifically, Step 1(II) invokes `calError` to update e^{max+}. Step 1(III)(a) calculates the quota of $[-\infty, \infty]$ filters that can be allocated to stream sources with $[l, u]$ filters. If this value (equal to $e^{max+} - n^+$) is larger than zero, we assign $[-\infty, \infty]$ to S_i. We choose S_i with the assumption that if S_i crosses the $[l, u]$ bound now, it is likely to cross the same bound again later. Thus, any future updates generated from S_i can be filtered. Step 1(III)(b) then increments the number of false positive filters allocated.

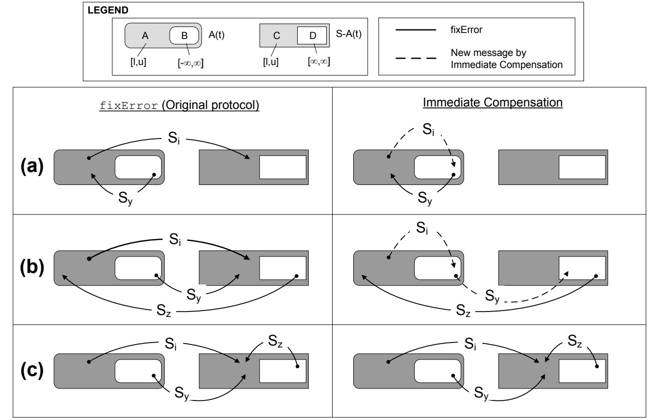


Fig. 6. Cases handled by `fixError`.

Case 2: $\boldsymbol{v_i \notin [l, u]}$. Stream S_i satisfied Q immediately after $[l, u]$ was installed to its filter, but S_i is no longer the answer to Q at time t_u. Step 2(I) removes this "bad answer" from $A(t_u)$, so that the size of $A(t_u)$ is reduced by one. Step 2(II) then updates e^{max+} and e^{max-}. We check in Step 2(III) whether one of the constraints in Equations 1 and 2 are violated. If so, correctness is violated, and `fixError` will be invoked (Step 2(III)(a)). Now, let t_c be the time when either the false positive or negative fraction (i.e., f^+ and f^-) attains its maximum value without violating query tolerance. Thus, immediately before `fixError` is run, we have

$$|A(t)| = |A(t_c)| - 1 \tag{13}$$

Since `fixError` can be expensive, its execution should be avoided if possible. Recall that our protocol uses the *incremental deployment* technique – that is, only some false positive/negative filters are deployed initially. Intuitively, the correctness is "stronger" than required by the tolerance, and thus reduces the chance for `fixError` to be called.

If `fixError` is not invoked, we check whether there are still some false negative filters not assigned by incremental deployment yet (Step 2(IV)). If so, we assign the $[\infty, \infty]$ filter to stream source S_i. This assignment does not violate the false negative requirements, since there are false negative filters not allocated. We assume that S_i is likely to have values crossing the $[l, u]$ bound in the near future, so that the $[\infty, \infty]$ filter can suppress this update.

B. Error Fixing

The `fixError` routine restore query correctness by replacing the false positive/negative filters with $[l, u]$ filters. Recall from Step 2 in Figure 5 that the violation of correctness requirements is due to the removal of an answer (in Step 2(I)). Let us investigate how `fixError` tackles this problem. For the purpose of explanation, we classify the streams into four disjoint sets, based on the status of filters, and whether they belong to the query answer:

- A: Stream sources in $A(t)$ with $[l, u]$ filter;

- B: Stream sources in $A(t)$ with $[-\infty, \infty]$ filter;
- C: Stream sources not in answer set ($S - A(t)$) but with $[l, u]$ filter, where S is the set of all stream sources;
- D: Stream sources in $S - A(t)$ with $[\infty, \infty]$ filter.

Notice that B is the set of stream sources installed with false positive filters, and D contains all stream sources with false negative filters. The legend on the top of Figure 6 illustrates these four sets of stream sources. The shaded boxes represent sets of stream sources with the $[l, u]$ filters, while the white ones depict those with $[-\infty, \infty]$ or $[\infty, \infty]$ filters.

S_y: Set of stream sources with false positive filters
S_z: Set of stream sources with false negative filters
1. **if** $n^+ > 0$ **then**
   (I) request value from S_y with $[-\infty, \infty]$ constraint
   (II) **if** $v_y \in [l, u]$ **then**
      (a) install $[l, u]$ for the filter of S_y
      (b)$n^+ \leftarrow n^+ - 1$
      (c)**quit**
   (III)remove S_y from $A(t)$
2. **if** $n^- > 0$ **then**
   (I) request value from S_z with $[\infty, \infty]$ constraint
   (II) **if** $v_z \in [l, u]$ **then** insert S_z into $A(t)$
   (III)install $[l, u]$ for the filter of S_z
   (IV) $n^- \leftarrow n^- - 1$

Fig. 7. The `fixError` routine (at the server side).

Recall that `fixError` is invoked because a stream S_i returns an update which is outside the $[l, u]$ bound. As shown in Figure 7, when $n^+ > 0$, a stream source S_y with a false positive filter (i.e., Set B) is requested to send its value (Step 1(I)). There are two cases, depending on whether $v_y \in [l, u]$.
Case 1: $v_y \in [l, u]$. This means S_y is currently a true answer. We install a $[l, u]$ filter to S_y. Hence, $v_y \in [l, u]$ when no update is received from it (Step (II)(a)). The left side of Figure 6(a) illustrates this, where solid lines represent the change of the sets for S_i and S_y. Notice that as S_y has been assigned a false positive filter, v_y has already been in $A(t_u)$, and so $|A(t)|$ remains unchanged (i.e., equal to $|A(t_c)| - 1$ according to Equation 13). Moreover, S_y is no longer a false positive, and so $e^+(t)$ is decremented. Thus, $f^+(t)$ is now less than $\frac{e^{max+}(t_c)-1}{|A(t_c)|-1}$, which is also less than $f^+(t_c)$, meeting the false positive constraint. The false negative constraint is also satisfied: by Equation 4, $f^-(t) \le \frac{e^{max-}(t)}{|A(t)|-e^{max+}(t)}$, or $\frac{e^{max-}(t_c)}{(|A(t_c)|-1)-(e^{max+}(t_c)-1)}$, which is less than ϵ^-.
Case 2: $v_y \notin [l, u]$. S_y is now a true negative, and so we remove S_y from $A(t)$ (Equation 13). Then $|A(t)|$ becomes $|A(t_c)| - 2$ (Step 1(III)). Since $e^+(t)$ is also decremented, $f^+(t)$ is less than $\frac{e^{max+}(t_c)-1}{|A(t_c)|-2}$. As $\epsilon^+ \le 0.5$, $\frac{e^{max+}(t_c)-1}{|A(t_c)|-2}$ cannot be larger than $\frac{e^{max+}(t_c)}{|A(t_c)|}$, and is less than ϵ^+.

However, $f^{max-}(t)$ is at most $\frac{e^{max-}(t_c)}{(|A(t_c)|-2)-(e^{max+}(t_c)-1)}$, and can be more than ϵ^-. To remedy this, we pick one stream source with a false negative filter (say S_z) from D (Step 2(I)). If $v_z \in [l, u]$, we include S_z in the answer (Step 2(II)). We also install the $[l, u]$ filter to S_z (Step 2 (III)). Now $|A(t)|$ is increased to $|A(t_c)| - 1$, and $f^-(t)$ is at most $\frac{e^{max-}(t_c)-1}{(|A(t_c)|-1)-(e^{max+}(t_c)-1)}$, which is smaller than ϵ^-. Further, $f^+(t)$ is at most $\frac{e^{max+}(t_c)-1}{|A(t_c)|-1}$, which is less than ϵ^+. Thus correctness 2 is met, as shown by the solid lines in Figure 6(b).

On the other hand, if $v_z \notin [l, u]$, $|A(t)|$ and $e^+(t)$ remain unchanged and thus the false positive constraint is still satisfied. Since $e^-(t)$ is at most $e^{max-}(t_c) - 1$, $f^-(t)$ is at most $\frac{e^{max-}(t_c)-1}{(|A(t_c)|-2)-e^{max+}(t_c)}$, which is smaller than ϵ^- because $\epsilon^- \le 0.5$. By assigning the constraint $[l, u]$ to the filter of S_z, correctness 2 is also met. This case is shown in Figure 6(c).

Notice that `fixError` involves the sending of three messages in the worst case. It also reduces the number of false positive/negative filters. By preserving some of these filters in the initialization phase and deploying them carefully (i.e., using *incremental deployment*), the penalty due to the execution of `fixError` can be reduced, as shown by our experiments.

C. Immediate Compensation

A problem of `fixError` is that the false positive/negative filters deployed to the stream sources will be replaced by the $[l, u]$ filters. This causes more communication, since an $[l, u]$ filter allows updates to be generated. To alleviate this, we introduce the *Immediate Compensation*, which attempts to maintain the number of false positive/negative filters.

Let us examine Figure 6(a) again, where the original `fixError` protocol (on the left) puts S_y into the set A by giving it an $[l, u]$ bound, while S_i is removed from the answer i.e., put in set C. As a result, the false positive filter associated with S_y is "lost". Our new approach avoids this by treating S_i as a false positive instead (shown in dotted lines on the right). This is possible because although S_i is no longer a valid answer, we can put it into any of the sets B, C and D. More importantly, by putting S_i into set B and S_y into set A, the "loss" of S_i from the answer is *immediately compensated* by the presence of S_y. Thus, correctness is restored, and the number of false positive filters remain the same as before.

The same principle can be applied to the scenario in Figure 6(b), where S_y is not a query answer. Instead of putting S_i into B and S_y into C (as in the original `fixError`), we can do the following (shown in dotted lines):

1) Put S_i into B by associating S_i with the $[-\infty, \infty]$ filter.
2) Put S_y into D by associating S_y with the $[\infty, \infty]$ filter.

From Figure 6(b), S_y is replaced by S_i. Also, the displacement of S_z from D is compensated by the arrival of S_y. Thus, the sizes of all the four sets remain the same, and correctness is fixed. Also, no false-positive or false-negative filters are compromised after the fixing procedure.

To conclude, the new `fixError` can retain the false positive/negative filters in cases (a) and (b), but not in (c) (Figure 6). Thus, in most situations, the performance of the protocol will not deteriorate due to error fixing. The extra cost of immediate compensation compared to the old `fixError` is that a message is sent to S_i, for converting its filter from $[l, u]$ to $[-\infty, \infty]$. Our experiments show that this is worthwhile.

D. Evaluation of Multiple Queries

Our single-query protocol can be extended to handle situations where more than one continuous query are executed

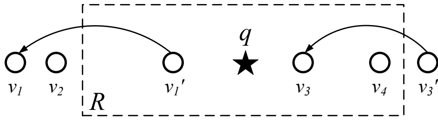


Fig. 8. False positives and false negatives for a k-NN query.

at the same time. First, we apply the initialization of FT-NRP to each query involved. The filter bounds computed for each query are sent to the stream sources. If there are m concurrent queries, then each stream source will be associated with m filter bounds. Second, during maintenance, the value of a stream is checked against the m filter bounds. The new value will only be sent to the server if it violates any of these bounds. Since the update is only sent once even if the value crosses the boundaries of one or more filter bounds, some communication overhead can be saved.

V. RANK-BASED QUERIES

A k-NN query can be viewed as a range query: if we know the bound R that encloses the k-th nearest neighbor of the query point q, then any objects with values located within R will be an answer to the k-NN query.

We can use this idea to design a filter scheme for k-NN query (with zero-tolerance). We call this protocol **ZT-RP**. During initialization, it computes R and then distributes R to all the stream filters. If no responses are received from the streams, the server is assured that all k objects are within R, and they are still the k nearest neighbors of q. Since no error is allowed, if any object enters or leaves R, we have to recompute R so that R still encloses the k nearest objects. In addition, the new R has to be announced to every stream.

The main drawback of this simple protocol is that it is sensitive to an object's value crossing R. When this happens, R has to be recomputed and announced to every stream! Let us investigate how this problem can be alleviated.

A. *Using* **FT-NRP** *for k-NN Query*

We just discussed how to view a k-NN query as a range query for the purpose of constraint deployment. Recall that the definition of fraction-based tolerance is the same for k-NN query and range query. To develop a fraction-based tolerance protocol for a k-NN query, one may consider transforming a k-NN query to a range query and then directly apply **FT-NRP**. Unfortunately, this is incorrect. As we show shortly, the ϵ^+ and ϵ^- parameters of a k-NN have to be first converted to two other values.

Specifically, let ρ^+ and ρ^- be the maximum false positive and negative tolerance value used by **FT-NRP**, in order to answer a k-NN query with tolerance ϵ^+ and ϵ^-. Let R be the smallest region that initially bounds the kth-ranked object and thus contains k objects. Similar to the initialization of **FT-NRP**, for objects with values in R we assign false positive filters to $k\rho^+$ stream sources; for stream sources with values outside R, we apply false negative filters to $k\rho^-$ stream sources. Other stream sources use R as the filter bounds. Let us examine how ρ^+ and ρ^- should be set.

Meeting false positive requirement. Suppose R encloses the k nearest objects of q. Let S_1 be part of the answer set, and $v_1' \in R$ is the value of S_1 last reported to the server. Hence, S_1 is one of the k nearest neighbors. If S_1 is associated with a false positive filter, the new value of S_1, i.e., v_1, may not be located within R. Consider the situation shown in Figure 8. Suppose there exists a stream S_2 such that $v_1 < v_2$. Then S_1 is no longer a correct answer, since S_2 now ranks higher and it pushes the rank of S_1 to $k+1$. Therefore S_1 becomes a false positive. Since at most $|A(t)|\rho^+$ stream sources are assigned with false positive filters, at most $|A(t)|\rho^+$ false positives may be produced in this way.

Another kind of false positive is caused by false negative filters. Suppose S_4, being ranked k-th and lies within R, is an answer. Also assume S_3 has a false negative filter, whose last reported value, v_3', is outside R. Figure 8 shows that when the new value of S_3, i.e., v_3, is inside R, the rank of S_3 is k or higher. The rank of S_4 is demoted to $k+1$ and thus S_4 becomes a false positive. Since false negative filters can be assigned to at most $k\rho^-$ stream sources (Equation 5), at most $k\rho^-$ false positives are created.

The sum of the false positives generated by these two scenarios is $|A(t)|\rho^+ + k\rho^-$, where $|A(t)| < \frac{k}{1-\epsilon^+}$ (Equation 7). Also, there cannot be more than $|A(t)|\epsilon^+$ false positives, with a minimum value of $k(1-\epsilon^-)\epsilon^+$ (Equation 9). Therefore,

$$\rho^- \leq \frac{\rho^+}{\epsilon^+ - 1} + (1-\epsilon^-)\epsilon^+ \tag{14}$$

Meeting false negative requirement. Again there are two types of false negatives for a k-NN query. As shown in Figure 8, the first type of false negatives is caused by streams like S_3, whose last reported value v_3' is not within R, and is assigned with false negative filters. Later its new value v_3 is within R and its rank is raised to k or higher. The server does not know this, and so S_3 is a false negative. The number of false negatives is at most $k\rho^-$, the maximum number of false negative filters. The second type is caused by stream sources with false positive filters like S_1. Again S_1 was among the top-k objects since its last reported value v_1' is within R. However its new value v_1 is less than v_2, so S_2 ranks k or higher (without notifying the server). The maximum number of this kind of false negatives is thus $|A(t)|\rho^+$, the maximum number of false positive filters. Since the maximum number of false negatives for k-NN query is given by $k\epsilon^-$, the sum of the two kinds of false negatives, $k\rho^-$ and $|A(t)|\rho^+$, must be less than $k\epsilon^-$. Equation 7 simplifies this to:

$$\rho^- \leq \frac{\rho^+}{\epsilon^+ - 1} + \epsilon^- \tag{15}$$

Guaranteeing correctness. To make sure that both false positives and false negatives are met, we combine Equations 14 and 15 so that the following is achieved:

$$\rho^- \leq \frac{\rho^+}{\epsilon^+ - 1} + \min((1-\epsilon^-)\epsilon^+, \epsilon^-) \tag{16}$$

Essentially, given the tolerance ϵ^+ and ϵ^-, the values of ρ^+ and ρ^- must be configured to satisfy Equation 16. To maximize the benefit, ρ^+ and ρ^- should set as:

$$\rho^- = \frac{\rho^+}{\epsilon^+ - 1} + \min((1-\epsilon^-)\epsilon^+, \epsilon^-) \tag{17}$$

B. Fraction-based Tolerant k-NN Query

Once the values of ρ^+ and ρ^- are correctly set, we can extend **FT-NRP** to exploit the fraction-based tolerance of k-NN queries. The corresponding protocol, called **FT-RP**, differs from **FT-NRP** in two aspects:

(1) Unlike a range query with a fixed bound $[l, u]$, the "range" of k-NN query is defined by R – the tightest bound that contains the k-th nearest neighbor. Thus, **FT-RP** first finds R before running the initialization phase of **FT-NRP**. Notice that the filter constraint R so calculated will not be changed even when R contains more or less than k objects – except when the conditions described next are met. Essentially, we use R only as an estimate of the k nearest neighbors.

(2) A requirement for the answer $A(t)$ of a rank-based query is that $k(1-\epsilon^-) \leq |A(t)| \leq \frac{k}{1-\epsilon^+}$ (Equations 7, 9). Initially $|A(t)| = k$, but as time goes by, the number of items in $A(t)$ increases (decreases) when an object enters (exits) R. Intuitively, when $|A(t)| > \frac{k}{1-\epsilon^+}$, there are too many objects in R i.e., R is "too loose". Similarly, when $|A(t)| \leq k(1-\epsilon^-)$, too few objects are in R, i.e., R is "too tight". In either case, R is no longer an appropriate bound. We need to find a bound that encloses the new k-nearest neighbors.

Thus, the advantage of **FT-RP** over **ZT-RP** is that it does not recompute and broadcast R when an object enters or leaves R, but only when $A(t)$ drops below $k(1-\epsilon^-)$ or exceeds $\frac{k}{1-\epsilon^+}$.

VI. PERFORMANCE EVALUATION

We now examine the performance of our protocols. Sections VI-A, VI-B and VI-C analyze the computation, communication and power consumption of ZT-NRP and FT-NRP, respectively. Section VI-D presents the experimental setup, and Section VI-E discusses the results.

A. Computational Complexity

Let us first consider the computation cost of the processing server in ZT-NRP. In the beginning, ZT-NRP just needs to send the $[l, u]$ bound to the stream sources. Any update received from the stream sources will be used to refresh the query answer. Thus, both initialization and maintenance need $O(1)$ time.

For FT-NRP, during initialization, Steps 1 to 3 require a complexity of $O(n)$. Steps 4, 5, 6 and 8 cost $O(1)$. If random selection is used, Steps 7 and 9 require a total of $O((1-\omega)(e^{max+} + e^{max-}))$. Thus, initialization requires $O(n)$. If nearest-boundary selection is used, an additional sorting cost of $O(n\log(n))$ is required, and initialization costs $O(n\log(n))$. For maintenance, notice that the complexity of `fixError` is $O(1)$ (Figure 7). Thus, the maintenance phase requires only $O(1)$ for every update received by the server.

B. Communication Cost Model

Our distributed communication model is similar to the one described in [26], where a server can broadcast its messages to the stream sources. These messages are classified into three categories:

- **Broadcast cost**(C_b) for broadcasting a message from the server to all stream sources.
- **Downlink cost**(C_d) for sending a downlink message from the server to a stream source.
- **Uplink cost**(C_u) for sending an uplink message from a stream source to the server.

This cost model can be used to characterize the type of resources studied. For example, the cost can be the network bandwidth or energy consumed by substituting appropriate values. In Section VI-C, we discuss how power costs can be considered based on this model.

For ZT-NRP, the initialization process needs a cost of C_b for broadcasting the $[l, u]$ bound to all stream sources. During maintenance, a cost of C_u is needed for a stream source to report its value. As for FT-NRP, during initialization, all stream sources send their initial values in response to the broadcast request from the server (Step 1). Therefore, a cost of $C_b + n\,C_u$ is needed. Then, the server broadcasts a message which consists of the $[l, u]$ bound to every stream source, with a cost of C_b. After the initial set of false positives and negatives are determined, the FT-NRP needs a cost of $(1-\omega)(e^{max+} + e^{max-})C_d$ to deploy the false postive and false negative filters to stream sources (Steps 6-9). Thus, the communication cost during initialization is

$$2C_b + nC_u + (1-\omega)(e^{max+} + e^{max-})C_d \qquad (18)$$

i.e., a complexity of $O(n)$.

During maintenance, each update from a stream source will incur a cost of C_u. If resolution takes place, it requires a cost of $C_d + C_u$ for requesting a value from a false positive stream source (S_y). If the protocol further requires a false negative stream source S_z be interrogated, a cost of $(C_d + C_u)$ is needed. Therefore, the cost of maintenance is at most

$$3C_u + 2C_d \qquad (19)$$

For immediate compensation, an extra message for updating the filter of S_i may be needed, and so in the worst case, its cost is

$$3(C_u + C_d) \qquad (20)$$

Thus, maintenance of FT-NRP requires a communication cost of $O(1)$. Although immediate compensation is more expensive, our experiments show that this extra cost is worth paying. If we substitute the values of C_b, C_d, and C_u by 1, then the above analysis will yield the total number of messages used. Next, we study the energy consumption costs.

C. Power Consumption of Sensor Networks

In many data stream applications, battery power is a precious resource. For example, in wireless sensor networks (WSN), sensors are only equipped with a limited power source (e.g. 0.5 Ah, 1.2V) [2]. Therefore, it is important to preserve energy resources for the sensors in these systems.

Let us now study the energy requirements of sensors in FT-NRP and ZT-NRP. Assume each sensor is a data stream source, each of which can be installed with a filter. According to [24], there are two types of energy consumption due to data communication:

- **Transmission power**(P_t) for sending an uplink message to the server.
- **Reception power**(P_r) for receiving a downlink message from the server at the stream source.

The amount of energy consumed due to communication can now be deduced by using the communication cost model described in the previous section. In particular, the broadcast cost C_b becomes nP_r, since all the n sensors will receive a message from the server. We also equate the downlink cost (C_d) to P_r, and the uplink cost (C_u) to P_t. By substituting these values into the results obtained in Section VI-B, we can see that ZT-NRP requires an initialization and maintenance cost of nP_r and P_t respectively. For FT-NRP, we have:

- Initialization: $nP_t + (2n + (1-\omega)(e^{max+} + e^{max-}))P_r$
- Maintenance (without immediate compensation): $3P_t + 2P_r$
- Maintenance (with immediate compensation): $3(P_t + P_r)$

In addition to communication, a sensor needs to collect information from the external environment. We use **sensing power** (P_s) to denote the amount of energy required for a sensor to acquire a data value (e.g., temperature, location). We assume that unless a sensor is installed with a false positive/negative filter, it collects data at a fixed periodicity (or duty cycle). At any instant of time during maintenance, for ZT-NRP, there are n stream sources installed with $[l, u]$ filters. Thus, the total amount of sensing power consumed int the maintenance phase is nP_s. On the other hand, FT-NRP only has $(|A(t)| - n^+) + (|S - A(t)| - n^-)$ stream sources installed with $[l, u]$ filters, and so the sensing power required is $((|A(t)| - n^+) + (|S - A(t)| - n^-)) \times P_s$. The higher the number of false positive filters (n^+) or false negative filters (n^-), the more sensing energy is saved.

In the rest of this section, we present the experimental results for the network bandwidth and power consumption performance of our protocols.

D. Experimental Setup

We use CSIM 19 [33] to simulate the environment illustrated in Figure 3. We test the performance based on both real and synthetic data.

For real data, we choose the TCP traces described in [23]. Our experiment models a remote network monitoring application, where a central console is used to monitor a network composed of 800 subnets. The dataset contains 30 days of wide-area traces of TCP connections, capturing 606,497 connections. Each subnet represents a stream source. The "number of bytes sent" field in each packet trace is used as a data value. We assume an agent software that implements our filters is installed at each subnet router. We consider two kinds of queries: a top-k query and a range query. These two queries are important to the network monitoring domain. Specifically, the *heavy hitter analysis* is often used to continuously discover an IP-source (called "heavy hitter") that delivers the largest number of packets to the monitored network, which is also likely involved in Denial-Of-Service (DOS) attacks [5], [15]. If the user only needs to know the top-k heavy hitters, then this problem can be modeled as a top-k query, which reports continuously the subnets with the k highest traffic volume [4]. The range query can be used to classify subnets with different ranges of traffic volume. It also captures IP sources that may participate in a DOS attack, if the amount of traffic is identified to be within a "dangerous" range. Here we assume $k = 60$, and a query range of $[200, 350]$.

We also perform testing on synthetic data, where 5000 stream sources are generated. The time between each data item is generated follows an exponential distribution with a mean of 20 time units. The values of a data stream item is uniformly distributed in $[0, 1000]$. When a new data value is generated, its difference from the previous value follows a normal distribution with a mean of zero and a variance of 60 units. For range queries, we assume the default query range is $[400, 600]$. For rank-based queries, we simulate the k-NN queries with k equal to 60. These parameter values are tuned in such a way that interesting trends can be observed and compared with the real data set. We also examine the effect of varying the value distribution of the synthetic dataset.

For the FT-NRP protocol, the default value of ω is zero, and immediate compensation is not used. These are the same parameter settings for the experiments used in our previous paper [7].

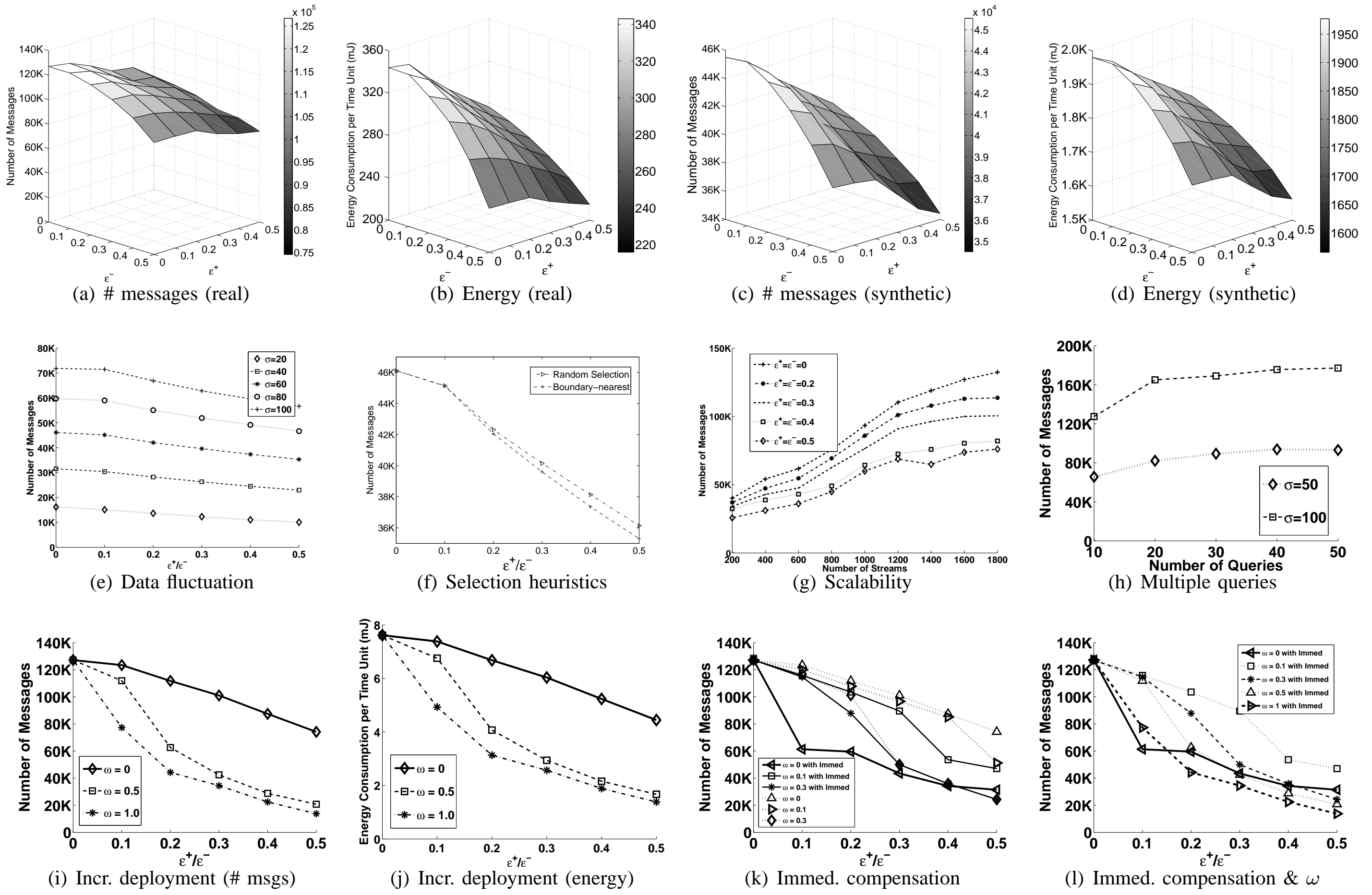
The two metrics used to measure the protocol performance is (1) the total number of messages generated and (2) the amount of energy consumed, based on the cost model described in Sections VI-B and VI-C. For the parameters of power consumption, we adopt the specifications used by a MICA2 mote [9]: the operation voltage of each sensor is $3.6V$, the transmission power P_t is $77.4mJ$ (at a distance of $+10dBm$), the reception power is $25.2mJ$ and the sensing power P_s is $2.52mJ$. The sensing frequency is once every 5000 time units.

E. Results

We describe the simulation results for range queries, a non-rank-based query, in Section VI-E1. Section VI-E2 presents the results for k-NN query, a rank-based query.

1) Non-Rank-Based Queries: We examine how well the **FT-NRP** protocol exploits the saving in messages and energy consumption for range queries. Figure 9(a) shows that the number of messages decreases as ϵ^+ and ϵ^- increase. For example, at $\epsilon^+ = \epsilon^- = 0.1$, the improvement over **ZT-NRP** (which does not exploit tolerance) is 2.65%. Hence, **FT-NRP** performs better than **ZT-NRP**. Figure 9(b) shows a similar behavior for energy consumption: 3.49% saving over **ZT-NRP** is achieved at $\epsilon^+ = \epsilon^- = 0.1$; and about 38% of power is saved at $\epsilon^+ = \epsilon^- = 0.5$.

For synthetic data, we examine **FT-NRP** under different values of ϵ^+ and ϵ^-. Figures 9(c) and 9(d) show that **FT-NRP** exploits tolerance on both messages and energy saving effectively similar to the real data. For example, at $\epsilon^+ = \epsilon^- = 0.1$, the message and energy savings are 2.25% and 2.22%, respectively. The relatively small improvement at small tolerance values (e.g., $\epsilon^+ = \epsilon^- = 0.1$) is due to the fact that the answer set is quite small (20% of stream sources), and so only a small number of false positive/negative filters can be



(a) # messages (real) (b) Energy (real) (c) # messages (synthetic) (d) Energy (synthetic)

(e) Data fluctuation (f) Selection heuristics (g) Scalability (h) Multiple queries

(i) Incr. deployment (# msgs) (j) Incr. deployment (energy) (k) Immed. compensation (l) Immed. compensation & ω

Fig. 9. Experiments on FT-NRP.

used. The overhead of maintaining these filters thus affects the performance of the protocol.

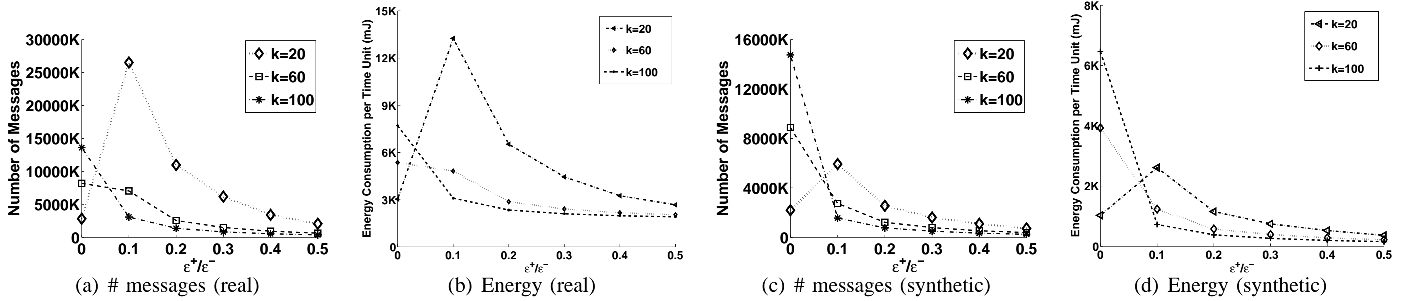
Data Fluctuation. Figure 9(e) illustrates the effect of data fluctuation (i.e., the amount of difference between two adjacent values in a stream) on **FT-NRP**, for synthetic data. We test the cases when $\epsilon^+ = \epsilon^-$, with different values of standard deviation (σ). For all values of σ tested, the performance improves with larger values of ϵ^+ and ϵ^-. Another observation is that as σ increases, **FT-NRP** yields more messages. When a data value changes more abruptly, it has a higher chance of violating the filter bound constraint and generating an update.

Selection Heuristics. We explore how **FT-NRP** is affected by the assignment of false positive/negative filters during initialization. Specifically, we compare the performance of two heuristics: *random* and *boundary-nearest*. Figure 9(f) shows that *boundary-nearest* is only slightly better than *random*. The difference is small because the assumption that values closer to the query boundary is more likely for its future values to cross the boundary again may not hold. For example, an object can be moving away from the boundary and so even if it is close to the boundary during initialization, it is not necessarily worth to assign a false positive/negative filter to its corresponding stream source. As we will illustrate, the use of incremental deployment – assign a fraction of $1 - \omega$ filters to the stream sources, and only allocate a filter to the stream source when it generates an update – often results in a better performance.

Let us now focus on real data.

Scalability with Data and Queries. We also test the effect of the number of stream sources and queries on **FT-NRP**. Figure 9(g) shows that **FT-NRP** scales well with the number of stream sources, under different values of ϵ^+ and ϵ^-. We further investigate the performance of our protocols in handling multiple queries, as discussed in Section IV-D. We consider a varying number of queries under different distributions. These queries are executed at the same time. The centers of the query ranges follow a normal distribution with a standard deviation (σ). As shown in Figure 9(h), the protocols in general scale well over a large range of number of queries. The number of messages does not increase linearly with the number of queries. This is because individual updates can be shared by more queries as the number of queries increases. Consider, for instance, two range queries that have a high degree of overlap. Then, the crossing of boundary for one query will likely co-occurs with the crossing of the other query's boundary. Hence, only one update is necessary for both queries. With a similar argument, we can see that the protocol performs better when the distribution of the queries is denser (i.e., decreases from $\sigma = 100$ to $\sigma = 50$), because each update can be shared by more queries.

Incremental Deployment. Figure 9(i) illustrates the effect



(a) # messages (real) (b) Energy (real) (c) # messages (synthetic) (d) Energy (synthetic)

Fig. 10. Experiments on FT-RP.

of ω on the number of messages. Recall that the larger the value of ω, the lower the number of stream sources is assigned with false positive/negative filters in the initialization phase. These filters are only assigned in the maintenance phase, when their values cross the query boundaries. We can see that the the performance improves with a larger value of ω. There are two reasons. First, before the initialization phase, we may not know exactly which stream sources are worthy to be installed with the false positive/negative filters. If a stream source is assigned with a false positive filter but its value actually has little chance of crossing the boundary, then this filter is wasted. By delaying the filter assignment until a stream's value has crossed the query boundary, the filter is allocated to a stream source which is more "active" (in crossing the query boundary), so that any other updates generated by this stream source are dropped. Secondly, observe that `fixError` will be executed in Step 2(III)(a) if one of the two conditions is violated. If n^+ and n^- are smaller than the maximum values allowed (correspondingly e^{max+} and e^{min-}), then `fixError` will not be executed. If ω is non-zero, then the initial values of n^+ and n^- will be less than the maximum values allowed, and so `fixError`, which is a costly operation, will be less likely to be executed upon the arrival of updates. We can observe similar results for energy saving, as shown in Figure 9(j).

Immediate Compensation. In this experiment, we examine the use of immediate compensation in `fixError`. Figure 9(k) shows the number of messages required, for three values of ω (0, 0.1 and 0.3). We can see that for all the values of ω shown, immediate compensation provides better performance than if it is not used. This is because the `fixError` in [7] replaces some false positive/negative filters with $[l, u]$ filters. Hence, fewer updates can be filtered after its execution. Immediate compensation, on the other hand, maintains the number of false positive/negative filters in most cases. Therefore, it provides a better performance than the original protocol.

We also note that the effect of immediate compensation is more profound at smaller values of ω. This is because at a large ω, more false positive/negative filters are preserved for future use, and it is less likely for `fixError` to be invoked (`fixError` is only executed if all the false positive/negative filters have been assigned). In fact, although not shown in the graph, when ω is larger than 0.5, immediate compensation almost yields no benefit. Thus, immediate compensation is more useful for small ω values.

In Figure 9(l) we show the effect of immediate compensation for different ω values. We can see that except for $\omega = 0$, as ω increases, the performance improves. For $\omega = 0$, all the false positive/negative filters have been assigned during initialization, and more stream sources can be allocated with filters in the beginning than other ω values. If immediate compensation is further used, with $\omega = 0$ the protocol could retain the largest number of filters after `fixError`, and so it performs well in this experiment.

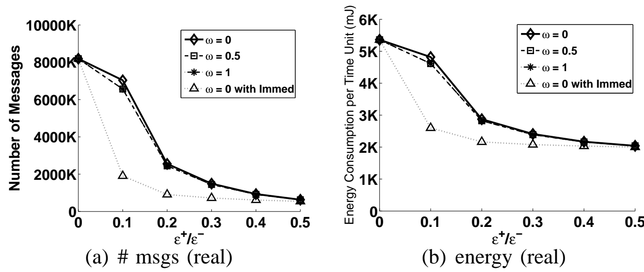
2) Rank-Based Queries: Next, we examine how FT-RP exploits the fraction-based tolerance for k-NN query, a rank-based query. Figure 10(a) shows the result under three values of k: 20, 60, and 100. As we can see, when k is 60 or 100, the number of messages drops significantly with a small increase in tolerance. This is because the bound R for enclosing the k nearest objects is not "tight", and so objects can cross R without requiring R to be recomputed and sent to the stream sources. With zero tolerance, however, R virtually changes every time an object crosses it. When $k = 20$ and $\epsilon^+ = \epsilon^- = 0.1$, the protocol yields a high message cost. This is because the number of false positive/negative filters assigned is limited. Therefore, the little benefit of tolerance cannot overcome the high maintenance cost. Thus, FT-RP is not suitable in this situation. For energy consumption, the radio energy incurred by message transmission dominates, thus as shown in Figure 10(b), the overall energy consumption follows the same behavior as message count.

Figure 10(c) and Figure 10(d) show the performance of the protocols for the synthetic data set. The result is similar to the case of real data. We note that the performance for $k = 20$ for synthetic data is better than the real data set. The reason is that the values of the synthetic data are relatively stable (i.e., they crossed the R bound less frequently). Thus, the number of updates generated, as well as the recomputation of R, is smaller.

Finally, we examine Incremental Deployment and Immediate Compensation for FT-RP using the real dataset. We assume $k = 60$. Figures 11(a) and 11(b) show the results of message count and energy consumption respectively. In contrast to FT-NRP, Incremental Deployment has a relatively small improvement. Recall from Equation 17 that the values of ρ^+ and ρ^- are usually smaller than the user-specified tolerance (i.e., ϵ^+ and ϵ^-). Thus, the number of false positive/negative filters computed is small, and the performance is relatively insensitive to the value of ω.

On the other hand, Immediate Compensation performs much better when the tolerance is at 0.1 – a message saving of 75% can be achieved. Due to the smaller number of false positive/negative filters available, `fixError` has to be executed more frequently. By using Immediate Compensation, the false-positive/negative filters is more likely to be retained after the evaluation of `fixError`. This in turn reduces the number of

recomputation and deployment of the new R bound, resulting in a better performance.



(a) # msgs (real) (b) energy (real)

Fig. 11. Incremental Deployment (FT-RP).

VII. Conclusions

In this paper we developed protocols to improve the performance of data stream management systems. The main idea of these protocols is to translate fraction-based query tolerance to filter bounds, and transmit these bounds to the stream sources. The stream sources can then conditionally drop their updates without affecting query correctness. We designed algorithms that initiate and maintain filter bounds for non-rank- and rank-based queries. We further presented two variations of these protocols, namely *incremental deployment* and *immediate compensation*. Through detailed testing on real and synthetic data, we verified the effectiveness of our protocols in reducing communication and energy costs.

For future work, we will enhance our algorithm to adapt to the addition of stream sources during query execution. We also want to examine how the value distribution across different data streams can be used to design better filter assignment policies. We will address how other data stream optimization algorithms (e.g., synopse construction and load shedding) can employ the notion of fraction-based tolerance. We will also study how this new notion can be used to design filters for other query types (e.g., joins). Another interesting study is to design a new correctness notion that combines the advantages of value-based and non-value-based tolerances, and develop filtering protocols accordingly. Such a tolerance could be useful when a user is not only concerned about the number of false positives, but also that the actual values of the false answers are not too far away from the query range. Another interesting question is how our protocols can be applied to sensor-networked environments, where each sensor can be viewed as a stream source. The challenge is to exploit the unique characteristics of sensor networks (e.g., sensors are arranged in a hierarchical manner) in order to further optimize the use of energy and bandwidth resources.

Acknowledgments

This work was supported by the Research Grants Council of Hong Kong (GRF Projects 513508 and 513307), the Germany/HK Joint Research Scheme (Project G_HK013/06), and the University of Hong Kong (Project 200808159002). We thank the reviewers for their insightful comments.

References

[1] D. Abadi et al. The Design of the Borealis Stream Processing Engine. *CIDR*, 2005.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey, *IEEE Communication*, Aug 2002.

[3] A. Arasu et al. Characterizing memory requirements for queries over continuous data streams. *TODS*, 29(1), 2004.

[4] B. Babcock and C. Olston. Distributed top-k monitoring. *SIGMOD03.*

[5] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312:3-15, 2004.

[6] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. *SIGMOD03.*

[7] R. Cheng, B. Kao, S. Prabhakar, A. Kwan, and Y. Tu. Adaptive stream filters for entity-based queries with non-value tolerance. *VLDB05.*

[8] Y. Chi and H. Wang and P. Yu and R. Muntz. Loadstar: A Load Shedding Scheme for Classifying Data Streams. *SIAM05.*

[9] Crossbow Inc. *MPR-Mote Processor Radio Board User's Manual*.

[10] B. Cui et al. Exploring bit-difference for approximate knn search in high-dimensional databases. *Australasian Database Conference*, 2005.

[11] A. Deligiannakis, Y. Kotidis, and N. Roussopoulos. Hierarchical in-network data aggregation with quality guarantees. *EDBT04.*

[12] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. *VLDB04.*

[13] L. Doherty, B.A. Warneke, B.E. Boser, and K.S.J. Peter. Energy and performance considerations for smart dust. *IJPDSN*, 4(3), 2001.

[14] D. Abadiand et al. Aurora: A data stream management system. *SIGMOD03.*

[15] S. Ganguly, M. Garofalakis, R. Rastogi, and K. Sabnani. Streaming algorithms for robust, real-time detection of DDoS attacks. *ICDCS07.*

[16] M. Greenwald and S. Khanna. Power-conserving computation of order-statistics over sensor networks. *PODS04.*

[17] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. *VLDB03.*

[18] G. Iwerks, H. Samet, and K. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. *VLDB03.*

[19] A. Jain, E. Chang, and Y. Wang. Adaptive stream resource management using Kalman filters. *SIGMOD04.*

[20] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. *ASPLOS-X 2002.*

[21] S. Khanna and W. Tan. On computing functions with uncertainty. *PODS01.*

[22] N. Koudas, B. Ooi, K. Tan, and R. Zhang. Approximate NN queries on streams with guaranteed error/performance bounds. *VLDB04.*

[23] Lawrance Berkeley National Laboratory. The Internet Traffic Archive, USA. URL http://ita.ee.lbl.gov.

[24] O. Landsiedel, K. Wehrle, and S. Gotz. Accurate Prediction of Power Consumption in Sensor Networks. *EmNetS-II, 2005.*

[25] Z. Liu, K.C. Sia, and J. Cho. Cost efficient processing of min/max queries over distributed sensors with uncertainty. *ACM SAC05.*

[26] K. Mouratidis et al. A Threshold-Based Algorithm for Continuous Monitoring of k Nearest Neighbors. *TKDE,17(11), Nov 05.*

[27] J. Ni and C. V. Ravishankar. Probabilistic spatial database operations. *SSTD03.*

[28] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. *SIGMOD03.*

[29] V. Poosala and V. Ganti. Fast approximate query answering using precomputed statistics. *ICDE99.*

[30] A. Silberstein, R. Braynard, and J. Yang. Constraint chaining: on energy-efficient continuous monitoring in sensor networks. *SIGMOD06.*

[31] N. Tatbul et al. Load Shedding in a Data Stream Manager. *VLDB03.*

[32] Y.-C. Tu and S. Liu and S. Prabhakar and B. Yao. Load Shedding in Stream Databases: A Control-Based Approach. *VLDB06.*

[33] Mesquite Software. CSIM 19. URL http://www.mesquite.com.

[34] S. Vrbsky and J. Liu. Producing approximate answers to set- and single-valued queries. *Journal of Systems and Software*, 27(3), 1994.

[35] O. Wolfson, P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *DPD*, 7(3), 1999.