Adaptive Strategies for Persistent Queries in Dynamic Environments

Christine Julien
Jamie Payton
Gruia-Catalin Roman

TR-UTEDGE-2007-013

© Copyright 2007
The University of Texas at Austin
Adaptive Strategies for Persistent Queries in Dynamic Environments

Christine Julien
Department of Electrical and Computer Engineering
University of Texas at Austin
c.julien@mail.utexas.edu

Jamie Payton
Department of Computer Science
University of North Carolina at Charlotte
payton@uncc.edu

Gruia-Catalin Roman
Department of Computer Science and Engineering
Washington University in Saint Louis
roman@wustl.edu

ABSTRACT
This paper is concerned with continuous and adaptive monitoring of highly-dynamic environments such as mobile, ad hoc, and sensor networks. At the logical level, the application issues a query that persists over time and returns a continuously updated view of relevant aspects of the system configuration—we call this a persistent query. We examine the semantic relation between the sequence of observable results and the actual configuration changes taking place in the system. A distinctive feature of our approach is that it considers temporal and spatial dimensions of the query evaluation strategy, possible dynamic adaptations of this strategy, and the data integration constructs being used. The result is a framework that allows an application to exercise a new degree of semantic control over the results generated by a persistent query, thus achieving an accurate interpretation of the data being examined.

1. INTRODUCTION
The world of computing and communications has undergone a dramatic change with the introduction of wireless mobile devices and wireless sensor networks. New kinds of applications can be built, often characterized by tight embedding into the environment, the enterprise, or social structure. The dynamic and ad hoc nature of such networks aligns with the needs of applications that exhibit a great deal of fluidity and must respond to rapid and frequent changes. Sensors can be added to a manufacturing environment without changing the application; as portions of a sensor network designed for habitat monitoring are washed out by heavy rains, new sensors may be deployed without intervention; a researcher may join an exploration team simply by being in proximity of PDAs carried by the team’s biologists; it is becoming feasible to replace RFID tags used to track cargo with mote-level devices storing detailed information that can be interrogated by robots managing the shipping process. In all these examples, the untethered nature of the devices allows dynamic changes to the network structure and the physical distribution of the application components.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXX-XX-X/XX/XX ...$5.00.
for dynamically selecting an inquiry strategy.

To illustrate these concepts, consider a simple example that involves a persistent query that specifies a construction site worker’s interest in hazardous materials within a two-hop network neighborhood. As the worker moves through the construction site, nearby hazardous materials sensors should continuously report concentration measurements as they enter or leave this neighborhood. The inquiry might periodically flood the area with queries that die after they traverse two hops; the integration might be as simple as removing duplicate responses and updating a list of hosts in the region and their concentration readings; the adaptation strategy might be one in which the query frequency adjusts automatically to the rate of change in the neighborhood; finally, semantic accuracy may be accomplished by communicating to the worker the level of uncertainty about the relation between the data being reported and the actual system state in the worker’s vicinity, e.g., 1.00 may be assigned to the case when the rate of change is at the level of seconds while the rate at which queries are issued is in the microsecond range.

The remainder of the paper is organized as follows. Section 2 gives an overview of a framework that supports adaptive persistent query execution in dynamic pervasive computing environments. Section 3 details the inquiry and adaptation strategies that may be used to issue one-time queries as part of a persistent query. We demonstrate our framework through an application example in Section 4. Section 5 presents related work. Conclusions appear in Section 6.

2. A PERSISTENT QUERY FRAMEWORK

In this section, we build a formal framework that supports reasoning about persistent queries in dynamic pervasive computing environments. This persistent query framework shares characteristics of our model of query execution described in [11] that was used to reason about the semantics of one-time queries in dynamic environments. This new model, however, provides a foundation for reasoning about long-lived queries in dynamic networks. In this section, we formalize our new perspective on persistent queries, first by defining configurations and queries and then by defining how persistent queries are evaluated within that model.

Although we provide a general persistent query framework that can be applied to support a wide range of applications, we use a single example application domain throughout the remainder of this paper to highlight the key points of our approach. We focus, in particular, on the problem of asset and operations management for construction projects. Careful management of physical assets and personnel on a construction site can be essential to the success of a project. We view a dynamic pervasive computing network as a system of hosts, where the number of hosts may be large, but the universe of hosts is bounded. We assume each host has a location and a single data value (though a single data value may represent a collection of values). At any moment in time, a host is represented as a tuple \( h = (\iota, \nu, \lambda, \tau) \), where \( \iota \) is a unique host identifier, \( \nu \) is the host’s data value, \( \lambda \) is the host’s location, and \( \tau \) is a timestamp associated with this host tuple. The value \( \tau \) represents a host’s local clock. We assume hosts’ local clocks are loosely synchronized. A total ordering over all result tuples can be achieved by setting the least significant bits of \( \tau \) to be equivalent to the host’s unique identifier, \( \iota \). The global abstract state of a network, which we call a configuration, is simply a set of these host tuples, specifically the union of all host tuples.

To collect information in a dynamic network, a host issues queries. Given a specific query issuer, \( h \), (the reference host), queries operate over only the projection of the configuration that includes hosts that are reachable from \( h \).

Typically, reachability is defined in terms of physical network connectivity, captured by a relation that conveys the existence of a (possibly multi-hop) network path between two nodes. A host’s tuple can change over time, because the host’s data value changed, the host’s location changed, or the host’s clock ticked. Any single such change may result in a configuration change. A reference host issues a query over a set of configurations, whose endpoints are roughly defined by the configuration in which the query is issued and the configuration in which the query’s result is returned.

We informally define a persistent query to be the evaluation over time of a function over the history of all of the relevant, reachable results. A persistent query is effectively two functions: a projection and an integration. A persistent query’s projection defines the subset of relevant results in a single configuration. For example, on a construction site, a persistent query may desire to operate only over results that represent palettes of bricks on the site. This projection is effectively a filter the query defines over data and is a well-understood aspect of query systems. In our model, a persistent query’s integration is a function that generates the query’s result, given a history of relevant projections. For example, given a history of the available palettes of bricks, a construction site supervisor’s persistent query may calculate the rate of consumption of bricks over a day. Ideally, any changes that occur within the relevant projection should be observable to this integration function. This integration function is fundamental to the manner in which we define persistent queries, and it takes much of the focus of the remainder of this paper.

In the highly dynamic environments we target, generating a perfect picture of the relevant history over which to evaluate a persistent query’s integration function is prohibitively expensive. For this reason, we approximate the above idealized notion of a persistent query using a sequence of one-
time queries, or queries that appear to be issued over a single configuration and return the set of relevant results from that configuration. We require only that the queries comprising a persistent query are non-overlapping. Each of this one-time queries provides a piece of the projection portion of the persistent query defined above. We define the periodic results of a persistent query in terms of integration over the series of one-time query results (i.e., over a series of historical projections). In constructing such an implementation of a persistent query, many questions must be resolved. One must determine how often to issue one-time queries, what type of query processing algorithm (query protocol) to use, and how to parameterize that protocol to best resolve the persistent query. In the remainder of this paper, we use the term inquiry strategy to refer to the combination of query protocol, parameters, and frequency used to generate a one-time query result that supports a long-lived persistent query.

2.2 Formalizing Persistent Queries

As defined above, the result of a persistent query is an integration of the results of a non-overlapping series of queries. The goal of a persistent query is to provide a good reflection of the “ground truth,” or the actual state of the world. This ground truth is equivalent to a complete picture of the actual configurations \( (C_0, \ldots, C_j) \) over which the persistent query lives. A persistent query’s integration, then, should be a function \( f \) that is evaluated (and reevaluated) over this series of configurations: \( f(C_0, \ldots, C_j) \). Directly measuring the configuration for an application is prohibitively expensive at best and impossible in most cases. Therefore, we use a sequence of one-time queries to provide a reflection of the underlying configurations. The result of a one-time query is a set, \( \rho \), of host tuples. When reasoning about the results of a one-time query, we naturally want \( \rho \) to contain only one result from any single host (it should, after all, appear that \( \rho \) is analogous to a single configuration). For example, the construction project manager may want to know the number of steel beams that are currently available for use on site. A particular steel beam should only be reported once within the associated one-time query result. For a persistent query, on the other hand, we may be interested in trends in the results over time, or even in trends in results for a single host. For example, the construction manager may want to know the maximum temperature of each container of flammable material over the course of the day to keep the construction site safe; this involves monitoring trends over the history of one-time query results provided by the containers.

We denote the results of the sequence of one-time queries that comprise a persistent query as \( \rho_0, \ldots, \rho_i \). There is not a one-to-one correspondence between the set of configurations a persistent query executes over \( (C_0, \ldots, C_j) \) and the sequence of query results that provides a view of those configurations \( (\rho_0, \ldots, \rho_i) \). Measures of how well a single query result reflects the configuration(s) over which it executes are discussed in more detail in [11]. We instead focus on how persistent queries can provide views reflective of the configuration and its changes over time. The result of a persistent query usually requires some kind of analysis or integration over the sequence of query results. That is, the result of a persistent query at stage \( i \) (i.e., after the results of the \( i \)-th component query have been incorporated) is \( \pi_i = f(\rho_0, \ldots, \rho_i) \) where \( f \) defined in more detail in the next section, is an integration function. An example integration function that could be used to provide our construction project manager with fire safety information would, for all host tuples \( (h \equiv (i, \nu, \lambda, \tau)) \) in a query result \( (\rho_i) \), select the host tuple with the maximum temperature value \( (\max \nu) \) that was present at some time \( (\tau) \) for each container \( (i) \), and include it in the integrated query result. In summary, the ideal persistent query result is \( f(C_0, \ldots, C_j) \). However, because configurations \( (C_j) \) are not directly measurable, evaluating \( f(\rho_0, \ldots, \rho_i) \) gives us an approximation of the desired ground truth used as a way to achieve persistence.

Because a persistent query comprises a series of one-time queries, it makes sense to question what type or types of queries are used to generate the results. For example, flooding the network for results may be expensive but may achieve strongly consistent results, while randomly sampling a few nodes in the network provides much weaker consistency, but is much less expensive. In the next section, we look at some specific examples of such query protocols; for now, we state simply that there are different types of query protocols, that these different types provide different qualities of query results, and that the selection of a particular query protocol depends on several factors including the nature of the environment and the application’s requirements. We refer to the particular manner in which we query the environment at stage \( i \) of the persistent query as the inquiry strategy, which includes not only the one-time query protocol and any parameters it might have but also the frequency with which one-time queries are issued. We refer to the manner in which this inquiry strategy is determined as the adaptation strategy. Because the persistent query’s only point of interaction with the environment is through the query itself, the strategy selected at stage \( i \) depends on the results of the persistent query’s previous one-time queries. The information that may impact the selection of the inquiry strategy \( i+1 \) is \( \pi_i \). Formally, the inquiry strategy at stage \( i+1 \), \( q_{i+1} \), is a function, \( g \), applied to this meta-information of the persistent query: \( q_{i+1} = g(\pi_i) \). Section 3.3 gives some concrete examples of definitions of both \( \pi_i \) and the function \( g \). Notice, however, that the information used in the adaptation function, \( g \), is effectively the result of a second persistent query, with semantics identical to the application-level persistent queries defined previously.

This section has provided an overview of our persistent query framework. The next sections elaborate on this framework, building on the model of a persistent query we have elucidated here. The next section examines integration strategies assuming a static inquiry strategy (i.e., that the query type does not change over the course of the persistent query). We then look specifically at the mechanics of adapting the inquiry strategy based on previous results \( (\pi_i) \) and properties of those results \( (\pi_i) \).

3. Achieving Persistence

A persistent query comprises a sequence of one-time queries and can be defined by two components. First, a persistent query’s integration strategy determines how component query results are combined to generate the persistent query’s result. Second, a persistent query’s inquiry strategy determines the algorithms and protocols by which contributory one-time query results are obtained. In this section, we first assume an available history of projections to support a persistent query’s integration and examine some example integration functions in detail. We then look more closely at
how inquiry strategies can be defined to create the necessary history of projections using one-time query results.

3.1 Integration Strategies

In this section, we define a set of integration strategies. While we do not claim that this set is exhaustive, we demonstrate its ability to address the needs of a variety of queries. This also provides a framework within which new integration strategies can be defined in the future. For each of the integration strategies we define, we describe it formally, from an omniscient perspective. We also define each integration strategy in terms of previous integration results where possible to enable an implementation that avoids having to maintain an entire history. Finally, for each strategy, we provide an application instance from the asset and operations management application introduced in the previous section.

3.1.1 Cumulative Integration

An obvious thing one might desire to do with a persistent query is generate a picture of all results available over the query’s lifetime. For example, in asset and operations management on a construction site, the supervisor may want to monitor the identities of all workers and visitors to the site during a given day. This cumulative integration gives the query issuer a picture of all results that have been available since the start of the query. In terms of previous persistent query results, a cumulative integration can be achieved by simply merging a component query’s results with the persistent query result from the previous stage:

\[
\pi_i = \pi_{i-1} \cup \rho_i
\]

This statement demonstrates that the value of the persistent query result at stage \(i\) depends only on the value of the persistent query result at the previous stage and the result of the one-time query issued in stage \(i\). The cumulative integration result is depicted pictorially in Figure 1.

![Figure 1: Cumulative Integration. The persistent query result is exactly the union of the results of the component queries. In the figure, the query issuer has a darker border and is white in color. Other nodes’ colors indicate their data values. Between the first and second component queries, two nodes were added (at the upper left) and two nodes departed (at the right). Between the second and third queries, the values at two nodes changed (the two nodes in black in the third component query).](image)

3.1.2 Stable Integration

As another example, a query issuer may want a picture of the available results that did not change during the query’s execution. For example a construction site supervisor may want to know which materials are not commonly used and thus potentially available for reallocation. In this case, the stable integration gives the query issuer a view of all of the available results that have been static since the beginning of the query. In terms of previous persistent query results, a stable integration can be achieved by comparing the values present in the new component query’s result and the persistent query results from the previous step:

\[
\pi_i = \pi_{i-1} \cap \rho_i
\]

As with the cumulative integration, a stable integration result depends only on the stable integration result at the previous stage and the result of the one-time query. Figure 2 shows stable integration pictorially.

![Figure 2: Stable Integration. This example uses the same component queries as in Figure 1.](image)

3.1.3 Additive and Departure Integration

Two additional simple integrations collect results that have been added or have departed since the start of the query. The former allows the construction site supervisor to keep track of materials that have been delivered since the start of the query, while the departure query allows him to keep track of assets that have been consumed since the start of the query. Simply stated, an additive integration is the difference between the current result and the very first result:

\[
\pi_i = \rho_i - \rho_0
\]

Additive integration is demonstrated in Figure 3.

![Figure 3: Additive Integration. This example uses the same component queries as in Figure 1.](image)

Likewise, a departure integration (Figure 4) is the difference between the very first result and the current result:

\[
\pi_i = \rho_0 - \rho_i
\]

Because these statements of the simple additive and departure queries only compare results for two instances in
time, they cannot collect transient changes. That is, results that are added but subsequently removed are not captured in an additive query; results that depart but return are not captured in a departure query. More sophisticated (and therefore potentially more expensive to implement) transient integrations can capture these semantics.

### 3.1.4 Transient Additive Integration

The first of the transient semantics, transient additive integration, allows the site supervisor to get a complete view of all assets that were added to the site today, even if they subsequently departed or were consumed. Recursively stated in terms of the previous persistent query result, this is:

$$\pi_i = (\pi_{i-1} \cup \rho_i) - \rho_0$$

This integration semantic is depicted in Figure 5.

![Figure 5: Transient Additive Integration](image)

Figure 5: Transient Additive Integration. Between the first and second component queries, two nodes departed, one node was added, and two nodes changed data values. Between the second and third component queries, the two departed nodes came back, the added node left, and one of the two changed values returned to its original value.

### 3.1.5 Transient Departure Integration

In a similar fashion, the transient departure integration monitors all of the results that departed over the lifetime of the query, even if they subsequently returned. On a construction site, the supervisor may want to keep track of tool usage since more frequently used equipment may require more frequent maintenance. Again, since we wish to avoid maintaining all of the component queries’ results, we state the transient departure integration recursively:

$$\pi_i = \pi_{i-1} \cup (\rho_0 \cap (\rho_{i-1} - \rho_i))$$

A straightforward extension would count the number of times a particular result departed. In this case, the result $\pi_i$ is a set of pairs, each comprising the result and a count of the number of departures for that result.

### 3.1.6 Returns Integration

In a similar vein to the above transient departure integration, a returns integration gives a picture of exactly those results that did depart, but have since returned. As with the transient departure integration, the returns integration could give a construction site supervisor a picture of all of the tools that were used today. The negation of the returns integration (omitted for brevity) could be used to monitor the tools that went missing over the course of a persistent query. Given all of the component query results, returns integration can be stated as:

$$\pi_i = (\text{set } p, i, j, k : i > j > k \land p \in \rho_i \land p \notin \rho_j \land p \in \rho_k :: p)^1$$

The returns integration is a bit more difficult to state in terms of previous persistent query results. However, the result is directly related to the result of a transient departure integration. By first performing a transient departure integration, a returns integration simply adds a check for whether a tool that has departed is present in the current component query result:

$$\pi_i = \pi_{i-1} \cup (\text{set } p : p \in \rho_i \land p \notin \pi_{i-1}^{\text{departed}} :: p)$$

This is depicted in Figure 7.

This example demonstrates the power of our integration framework; by defining a small set of fundamental integration strategies, new strategies can be defined as needed.

### 3.1.7 Windowed Rate Integration

In addition to the set style operations described previously, our framework for specifying integration strategies also allows integration strategies that are more temporal in nature. The windowed rate integration provides one example. The type of the result $\pi$ in this example is a rate

$$\text{range :: expression}$$

In the three-part notation: (op quantified_variables : range :: expression), the variables from quantified_variables take on all possible values permitted by range. Each instantiation of the variables is substituted in expression, producing a multiset of values to which op is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies range, then the value of the three-part expression is the identity element for op, e.g., true if op is $\lor$ or $\emptyset$ when op is set.
instead of a set of results. Such an integration gives a picture of the rate at which results have arrived (or departed or changed) since some point in the past. On a construction site, measuring the rate of consumption of some material (e.g., bricks) can give our site supervisor an idea of the pace of the project’s work and can also allow him to predict when additional supplies will be necessary.

Here we define the additive windowed rate integration.

\[
\pi_i = \frac{1}{k} \sum_{j=i-k+1}^{i} (\rho_j - \rho_{j-1})
\]

When \( k = i \), this integration measures the largest window possible; it measures the rate of change since the beginning of the persistent query. (This is analogous to the way stock market trends are reported on a daily basis.) On the other hand, the smallest window size, \( k = 2 \) gives the instantaneous rate of change (it is effectively the first derivative).

This windowed rate of change can be performed for departures and changes to results as well. In measuring the rate of departure, the terms inside the summation are swapped. In measuring the rate of change, we instead sum the count of the data values that have changed.

### 3.2 Inquiry Strategies

To generate one-time query results as input to the integration strategies, the query processing system generates a series of one-time queries. The persistent query’s inquiry strategy comprises three components: 1) the frequency of one-time queries; 2) the query protocol used to implement the one-time queries; 3) parameters that influence the behavior of the one-time query protocol. Therefore, in addition to associating a sequence of one-time query results to the persistent queries, we also associate a sequence of the corresponding inquiry strategies used to generate those results.

The next section discusses in detail how both environmental and application-level characteristics can help the system to adapt inquiry strategy within a persistent query in response to changes. In this section, we first look at some potential inquiry strategies, organized based on the type of query protocol used to collect the query’s results. The goal of this paper is not to identify an exhaustive list of such query protocols but to instead create a framework for constructing and evaluating persistent queries using such protocols to implement the component queries.

**Flooding queries:** the most obvious and easily implementable query type is based on flooding a distributed network. Flooding based queries are the most common in mobile and pervasive computing protocols [4, 5, 12]. Approaches also exist that constrain the flooding query to some region of the network [13]. In a flooding query, the sending node broadcasts the query to all of its one-hop neighbors. Every time a node receives a new query, it rebroadcasts the query to its one-hop neighbors. In this way, a query is very likely to reach every node in the network or in the targeted region of the network. However, such a flooding approach can be very expensive in terms of the message overhead required to reach all of the nodes [10]. Figure 8(b) shows the messages sent by and the nodes responding to a flooding inquiry constrained to nodes within two hops of the query issuer.

Because a flooding protocol is deterministic, it cannot be parameterized, so no protocol parameters are included in the inquiry strategy. When successive flooding queries are combined to form a persistent query, one expects the variability among successive queries to stem from changes in the nodes’ positions and to changes in the data values located in the network. Because the flooding approach should reach all nodes in the targeted region of the network, if the nodes’ positions and data values are unchanged, successive flooding queries should give exactly the same results.

**Probabilistic queries:** because flooding based approaches can be overly consuming of bandwidth and power with diminishing returns in terms of usefulness, several more recent approaches explore parameterizing flooding-based protocols using probabilities in various fashions. Figure 8(c) depicts the behavior of a sample probabilistic flood. In this example, each node receiving a query retransmits it to two randomly selected neighbors. No query is retransmitted more than three times (i.e., the query’s time-to-live is three hops). This is just one example of varying protocol parameters in a probabilistic protocol. Other probabilistic protocols may vary additional parameters. For example, probabilities can determine how likely the node is to rebroadcast a received message at all [10]. Goosip based protocols extend this approach, with additional parameters to determine, for example, how many times to retransmit a message in a dynamic network [2, 6]. A probabilistic query effectively exhibits a spreading behavior controlled by some probability distribution and, unlike the flooding query above, does not guarantee that all nodes are reached by the query.

When a persistent query’s inquiry strategy employs a probabilistic query protocol, parameters in the inquiry strategy dictate the probabilistic nature of the protocol in the manners described above. When a persistent query is built out of a sequence of probabilistic flooding queries, variations among the response to the foundational queries result not only from the dynamics that impact flooding queries, but also from the randomness introduced by the use of the various probabilistic parameters. Even in a static network, successive queries may return different results.

**Location based sampling:** if location information is available, it can direct queries to particular regions of the environment. For example, a construction site supervisor may monitor the identities of workers and visitors on the site by localizing his queries on the nodes nearby entries and exits to the site. Figure 8(d) demonstrates a location
3.3 Adapting the Inquiry Strategy

Clearly, different inquiry strategies are appropriate in different situations and to meet different application requirements. For example, in a construction site application that is monitoring the potential diffusion of hazardous materials on the site, a periodic random sampling with a long period between samples may be sufficient for continuous monitoring. However, if an anomalous condition is detected, the appropriate inquiry strategy will definitely query more frequently and may also switch from a purely random sampling to a targeted location-based query that directs the query to the area of the anomaly. In the remainder of this section, we describe a framework to support an adaptive persistent query through adaptation of the inquiry strategy.

Adaptation is governed by adaptation rules, which specify how and when the inquiry strategy is adapted. The approach to determining how to adapt the inquiry strategy is relatively straightforward; the inquiry strategy is adapted by altering one or more of its components. Therefore, an adaptation strategy varies some combination of the particular protocol that issues the query, the parameters associated with that protocol, and the frequency with which the query is issued. In general, adaptation of the inquiry strategy may be triggered by the current state of the environment, the results of the query, or the needs of the application. Any adaptation trigger can be combined with any adaptation strategy to define an adaptation rule. Ultimately, this decision lies with the application designer.

Below, we focus our efforts on identifying these adaptation triggers, which define when the inquiry strategy should be adapted during the execution of a persistent query. The set of adaptation triggers presented here is not exhaustive; rather, the set is intended to illustrate the needs of applications and to provide a framework for identifying and expressing other adaptation triggers. These adaptation triggers are defined over the metadata $\pi_i$, associated with results for the query $q_i$. This metadata may simply be the result itself (i.e., $\pi_i$) or the result of an associated metadata persistent query $q_i$ that monitors properties in the environment.

3.3.1 Data Capture Quality

The selection of an inquiry strategy is often related to a tradeoff between the desired quality of the result and the acceptable overhead incurred by the inquiry strategy. Variability in the environment, the location of the query issuer, and the randomness introduced by probabilistic methods can impact the quality of results delivered by a particular query strategy. An adaptation trigger can be defined based on a measurement of the changes in quality of the captured measurement of the changes in quality of the captured

Based sampling query targeting region A in the figure.

Parameters for a location based protocol include not only the location(s) to be sampled, but they may also dictate patterns of inquiry, for example, that successive one-time queries look in different directions (e.g., north, then east, then south, then west). Results for successive one-time queries that make up a persistent query may vary due to node movement or because data values changed. If only a fraction of nodes in the targeted region are sampled, variations in this sample set may also impact the query results.

Random sampling: at the far end of the spectrum, a random sampling algorithm may just randomly select $k$ hosts to send the query, as depicted in Figure 8(e). The network paths used to communicate with the selected nodes depend on the network’s connectivity. This type of querying provides the least deterministic results, but at a significantly reduced cost. As with probabilistic flooding, the variance among successive queries stems from nodes’ movements, changes in data values, and the randomness introduced by the randomness of the query algorithm. Parameters that can be tuned within the inquiry strategy could dictate how many nodes to query or what percentage of the nodes to query.

While, in theory, our framework makes it possible to combine any inquiry strategy with any integration strategy, in practice some combinations may not be reasonable. For example, combining stable integration with a purely random sampling inquiry strategy is not likely to be useful since the likelihood of randomly sampling the same results in successive one-time queries is low. By defining the above integration and inquiry strategies, we have laid the groundwork for a reasoning framework that can provide directed guidance to persistent query developers with respect to intelligently selecting combinations of inquiry and integration strategies that provide a meaningful representation of the “ground truth.” In addition, we expect to be able to quantify what we mean by “meaningful” by defining formal consistency semantics for persistent queries using this framework.
data that occur due to this variability.

One way to express this kind of adaptation trigger is to use a logical distance function \( d \) to quantify the changes in quality between previously collected results and the most recent result set. Adaptation occurs when the evaluation of the distance function reaches a threshold:

\[
d(\tilde{\pi}_i, \tilde{\pi}_j) \leq \delta
\]

where \( \delta \) is the threshold, \( j \) is the current stage of the query, and \( i < j \). For each distance function, the threshold \( \delta \) at which adaptation is triggered depends on the needs of the application.

**Set Difference.** One way to measure the quality of results is to measure their variability between collection. Using the set difference operator, it is possible to quantify the percentage of current items returned at stage \( j \) that are newly available, have been modified, or are no longer reachable by the query issuer in comparison to the data values previously collected at stage \( i \). A construction supervisor can use this kind of distance function to determine when deliveries of new materials are being made so that his persistent query can begin to monitor supplies in the delivery area of the construction site with greater precision. Formally, we can express this distance function as:

\[
d = |\{ \tilde{\pi}_j \} - |\{ \tilde{\pi}_i \} |
\]

Applied to the monitoring of construction supply deliveries, the metadata in this distance function is simply the result of the query for new materials, so \( \tilde{\pi} = \pi \). Again, the threshold at which the evaluation of this distance metric triggers adaptation depends on the needs of the application. Here, the construction supervisor may recognize that a 20% increase in results over the execution of the persistent query indicates that a delivery has occurred.

The set difference operator can also be used to describe a distance function that measures how many result elements have departed over the execution of the persistent query. The construction site manager may describe this kind of adaptation trigger to alter his inquiry strategy in order to get a better picture of the tools that are actually available on the construction site when a certain percentage of tools are no longer available in the query result. This departure distance function can be expressed as:

\[
d = |\tilde{\pi}_i - \tilde{\pi}_0|
\]

Variations of these distance functions that allow a construction supervisor to define an adaptation trigger based on a cumulative view of the entire sequence of returned results are possible as well. These variations on set difference distance functions are analogous to the transitive versions of additive and departure integration strategies presented in Section 3.1. For instance, a construction supervisor may want to alter his inquiry strategy after some number of new supplies were delivered to the site, even if they departed later. This can be captured as:

\[
d = \frac{|\tilde{\pi}_j|}{\left| \bigcup_{i=1}^{j} \tilde{\pi}_i - \tilde{\pi}_0 \right|}
\]

**Aggregate distances.** Another way to describe the quality of a persistent query’s result over time is to measure the distance between the aggregated numerical values of result sets. In other words, aggregate distance measures can be used to define an adaptation trigger based on trends in the reported query results. For instance, a construction supervisor may use a query to monitor the total level of hazardous chemicals on the construction site. If the hazardous chemical level rises at a rate that implies a dangerous accident or leak may have occurred, the construction supervisor may need to alter his query to focus on the area where the leak occurred and to report results more frequently. In this case, an aggregate distance metric can be used to compute the distance of the total hazardous material detected:

\[
d = |\langle \sum p : p \in \tilde{\pi}_{i+1} \rangle - \langle \sum p' : p' \in \tilde{\pi}_i \rangle |
\]

Once again, the metadata used for adaptation in this example is simply the result of the query itself, so \( \tilde{\pi}_i = \pi \). The threshold value associated with this distance function that triggers adaptation is application-dependent. Variations of this kind of distance metric can be applied to other aggregate measures such as count, minimum, maximum, and average.

In theory, a distance function strategy can be used to define an adaptation trigger in conjunction with any inquiry strategy; in practice, however, some combinations will not be useful. For example, when using a location-based random sampling inquiry strategy, it is impractical to define an adaptation trigger using a distance function based on trends of the average values of chemical sensors distributed across a construction site. The results of such a query may provide a poor representation of chemical sensor readings and so cannot be used to provide a reliable or reasonable estimate of the trends in values.

### 3.3.2 Result Density

Another useful adaptation trigger relates to the density of query results. In regions of the network where the data is dense, data values contributed by neighboring hosts often have similar values. For example, temperature values collected from sensors that are placed side by side would likely report the same temperature within some small error. Therefore, in a dense area, it may be desirable to adapt a query, for example, to collect random samples rather than all results. This would yield a query result that is representative of the data in the region while reducing the overhead associated with processing a query in a dense area.

An adaptation trigger can be specified based on the density of results from the previous stage of the persistent query. We can get a rough estimate of the density of results by finding the average number of neighboring results; a relatively high average suggests that results are sparse. To define such an adaptation trigger, we first define a binary connectivity relation \( K \) that defines the existence of communication links between hosts that contribute to the result \( \tilde{\pi}_i \). For example, a physical connectivity relation that represents a connectivity model with a circular, uniform communication range can be defined using the location field of host tuples:

\[(h_{i0}, h_{i1}) \in K \iff |h_{i0} \uparrow 3 - h_{i1} \uparrow 3| \leq b\]

where \( \uparrow 3 \) refers to the third field of a host tuple—in this case, the host’s location—and \( b \) refers to a bound on the distance between two hosts to consider them connected. We can roughly determine the density of the query results by averaging across the number of one-hop neighbors that are...
included in the result \( \pi \). This characterization of density can be expressed as:

\[
\frac{\sum h_{0i}, h_{1i} : h_{0i} \in \pi_i \land h_{1i} \in \pi_i :: 1}{\sum h : h \in \pi :: 1}
\]

which gives the average number of neighbors that reported results at stage \( i \). The application can dictate how this measure relates to density and define a threshold that is used to determine adaptation.

### 3.3.3 Semantic Discovery

In some cases, it may be desirable to adapt an inquiry strategy based on the presence of a particular value. For example, an adaptation trigger may be defined as the point at which a result with a particular value was found:

\[
(\exists p : p \in \pi :: p = v)
\]

where \( v \) is the desired value.

The value of interest for adaptation may not be related directly to a reported result of a query. Instead, causal relationships between different kinds of values in the networks may be the basis for adaptation. For example, the discovery of smoke on a construction site implies the presence of fire. If smoke is detected, then a query should be adapted to “look harder” for evidence of a fire. These causal relationships can be captured as an adaptation trigger by associating a metadata query with the original query and evaluating over metadata that incorporates both the original query result and the result of the metadata query.

Adaptation triggers based on the discovery of a particular value typically rely on some knowledge of the semantics of the data related to query results. Therefore, in developing a framework to support adaptive persistent query execution, it is important to acknowledge the need to include application-specific knowledge. It is also necessary to provide the application programmers with ability to dictate the type of reactive adaptation that should be performed in response to the discovery of data values.

### 4. APPLICATION EXAMPLE

In this section, we illustrate the use of the adaptive query framework through an example. Consider a scenario in which an accident has occurred on a construction site, resulting in the release of hazardous chemicals. The area of the incident may spread as liquid chemicals spill over the ground and as airborne droplets are released into the atmosphere and scattered by the wind. A construction supervisor wants to ensure that he and his employees are safe as they evacuate the site. The supervisor’s application issues a persistent query over sensors that have been scattered across the entire construction site in order to monitor the danger of encountering hazardous materials within the area as he walks through it.

A specification of the construction supervisor’s needs is compiled into an inquiry strategy and associated adaptation rules (i.e., a set of adaptation triggers, each associated with a strategy for adapting the inquiry strategy). Each stage of the supervisor’s persistent query results in a report of the current location of the chemical cloud, which is computed by the integration strategy. From these results, a “danger index” can also be computed. This danger index is used in defining the adaptation trigger of the adaptation rule. As it becomes more likely that the construction supervisor has entered a dangerous area, he wants a more complete picture of the location of the chemical cloud to aid in navigating his way through the site. Therefore, the goal of the adaptation strategy associated with the trigger is to improve the precision with which the persistent query results reflect the environment.

The specification of the construction supervisor’s persistent query dictates the following initial inquiry strategy and adaptation rules.

#### Initial inquiry strategy

The emergency response crew member has arrived on the scene. The crew member wants to quickly estimate where the hazardous material is located and how it may be spreading through the area. The inquiry strategy to be used, then, will perform a sampling over a broad area. The inquiry strategy can be specified as:

- **protocol**: location-based random sampling
- **parameters**: location is the entire construction site, 5 hosts are sampled at random
- **frequency**: sample every 1 sec

The associated integration strategy will provide the most recent location information for sensors that report the presence of a hazardous chemical.

This integration strategy will be utilized until adaptation is triggered. The emergency responder’s adaptation trigger is defined by the “danger index” associated with query results. The danger index measures how likely it is that there is an unsafe level of hazardous chemicals in the area. This danger index is calculated as:

\[
\frac{\sum p : p \in \pi_i \land p = \text{contaminated} :: 1}{\sum p : p \in \pi_i \land p = \text{contaminated} :: \text{avg}}
\]

where \( \text{avg} \) is an abuse of notation that indicates the location of the reference host. This characterization basically equates danger with the ratio of the number of dangerous chemical readings and their average distance to the supervisor. A high danger index value indicates a high level of danger.

**First adaptation rule: near the cloud.** As the construction supervisor walks near the chemical cloud, he would like the persistent query to report the location of the chemical cloud with greater precision. This adaptation trigger is defined using a threshold on the danger index associated with the result of the query at stage \( i \). A threshold that may indicate that the supervisor is near the chemical cloud would yield 40 readings within 100 meters, for example. Once this threshold is reached, the inquiry strategy is adapted to provide results with increased precision:

- **protocol**: location-based probabilistic flooding
- **parameters**: location is area surrounding the supervisor, one-half of the hosts receive the query
- **frequency**: sample every 0.5 sec

**Second adaptation rule: in the cloud.** As the construction supervisor enters into the chemical cloud, he would like to have the query give the most precise readings possible so that he can discover the safest evacuation route as quickly as possible. The same danger index is used to define the adaptation trigger; here, the threshold can be defined as finding 75 readings within 100 meters. The inquiry strategy is adapted to provide the strongest possible view of the chemical readings:
• protocol: location-targeted flooding

• parameters: location is area surrounding the supervisor, all the hosts receive the query

• frequency: sample every 0.1 sec

With an adaptive persistent query strategy such as this, a construction worker can use conserve resources when in a safe area of the construction site and can use those resources to acquire information that can be used to navigate safely through the construction site when in danger.

5. RELATED WORK

Over the years, the sensor network community has embraced the notion of viewing the sensor network as a database. A number of distributed query processors which have been developed for sensor networks, e.g., implementations of directed diffusion [3], TinyDB [8], TAG [7], and Cougar [14]. Many of these systems are designed to deliver streaming data to the application through the use of a persistent query construct that proactively delivers query results to the issuing application (e.g., periodically at a specified sampling rate).

More recent work in this area has begun to explore a model-driven approach to query processing in sensor networks [1] for the purposes of energy conservation. In this work, each node in the network constructs a local model of the data available across the entire network and estimates the error in the model. If the estimated error associated with the local data model, a node will process a query over the local data model to avoid consuming energy by communicating the query over the network. This approach can be viewed as a way of specifying an adaptation trigger (e.g., using a distance metric and a threshold) associated with an adaptation strategy increases the frequency with over the network. A related approach in [9] also queries a local data model when possible, and furthermore uses the model to tune parameters associated with the query protocol when the query must be sent out over the network. These model-driven approaches illustrate how adaptation is currently being used in issuing queries in dynamic pervasive computing environments. To our knowledge, however, there is no system that include general support for dynamically adapting a persistent query based on arbitrary strategies provided by an application for dictating when and how to adapt execution.

6. CONCLUSIONS

This paper considered the issue of approximating persistent queries by using the results generated by sequences of one-time queries. The challenge rests with the very dynamic nature of the class of applications and networks under consideration, e.g., mobile users who are interested in continuous and adaptive monitoring of highly dynamic environments. Several dimensions of this complex problem have been examined. First, we addressed the issue of correct interpretation of the results obtained over time and we identified several pragmatically relevant integration methods over the history of received samples. Second, we considered the issue of long-lived data acquisition processes that are responsive to the persistent query semantics and adaptive to changes in the environment; several classes of inquiry strategies were highlighted along with trigger mechanisms that enable runtime rule-based adaptation of the inquiry strategy.

7. REFERENCES


