# Database-Centric Programming for Wide-Area Sensor Systems

Shimin Chen[1], Phillip B. Gibbons[2], and Suman Nath[1,2]

[1] Carnegie Mellon University
{chensm,sknath}@cs.cmu.edu
[2] Intel Research Pittsburgh
phillip.b.gibbons@intel.com

**Abstract.** A wide-area sensor system is a complex, dynamic, resource-rich collection of Internet-connected sensing devices. In this paper, we propose *X-Tree Programming*, a novel database-centric programming model for wide-area sensor systems designed to achieve the seemingly conflicting goals of expressiveness, ease of programming, and efficient distributed execution. To demonstrate the effectiveness of X-Tree Programming in achieving these goals, we have incorporated the model into IrisNet, a shared infrastructure for wide-area sensing, and developed several widely different applications, including a distributed infrastructure monitor running on 473 machines worldwide.

## 1 Introduction

A wide-area sensor system [2, 12, 15, 16] is a complex, dynamic, resource-rich collection of Internet-connected *sensing devices*. These devices are capable of collecting high bit-rate data from powerful sensors such as cameras, microphones, infrared detectors, RFID readers, and vibration sensors, and performing collaborative computation on the data. A sensor system can be programmed to provide useful *sensing services* that combine traditional data sources with tens to millions of live sensor feeds. An example of such a service is a Person Finder, which uses cameras or smart badges to track people and supports queries for a person's current location. A desirable approach for developing such a service is to program the collection of sensors as a whole, rather than writing software to drive individual devices. This provides a high level abstraction over the underlying complex system, thus facilitating the development of new sensing services.

Recent studies [6, 11, 14, 18, 19, 32] have shown that declarative programming via a query language provides an effective abstraction for accessing, filtering, and processing sensor data. While their query interface is valuable, these models are tailored to resource-constrained, local-area wireless sensor networks [6, 14, 18, 19, 32] or provide only limited support, if any, for installing user-defined functions on the fly [6, 11, 18, 19, 32]. As a result, the programming models are overly restrictive, inefficient, or cumbersome for developing services on resource-rich, wide-area sensor systems. For example, consider a wide-area Person Finder service that for update scalability, stores each person's location in a database nearby that location, for retrieval only on demand. To enable efficient search queries, the data can be organized into a location hierarchy with *filters* associated with each node of the hierarchy. These filters summarize the list of people currently within the node's subtree and are used to limit the scope of a search by checking the filters. Programming such filters, associating them with (all or parts of) a logical/semantic hierarchy, installing them on the fly, and using them efficiently within queries are not all supported by these previous models. Similarly, declarative programming models designed for wide-area, resource-rich distributed monitoring systems [12, 28, 31] do not support all these features.

In this paper, we present a novel database-centric approach to easily programming a large collection of sensing devices. The general idea is to augment the valuable declarative interface of traditional database-centric solutions with the ability to perform more general purpose computations on logical hierarchies. Specifically, application developers can write application-specific code, define on-demand (snapshot) and continuous (long-running) states derived from sensor data, associate the code and states with nodes in a logical hierarchy, and seamlessly combine the code and states with a standard database interface. Unlike all the above models (except for our earlier work [11]) that use a flat relational database model and SQL-like query languages, we use instead the XML hierarchical database model. Our experience in building wide-area sensing services shows that it is natural to organize the data hierarchically based on geographic/political boundaries (at least at higher levels of the hierarchy), because each sensing device takes readings from a particular physical location and queries tend to be scoped by such boundaries [13]. A hierarchy also provides a natural way to name the sensors and to efficiently aggregate sensor readings [11]. Moreover, we envision that sensing services will need a heterogeneous and evolving set of data types that are best captured using a more flexible data model, such as XML. This paper shows how to provide the above features within the XML data model.

We call our programming model *X-Tree Programming* (or *X-Tree* in short) because of its visual analogy to an Xmas tree: The tree represents the logical data hierarchy of a sensing service, and its ornaments and lights represent derived states and application-specific codes that are executed in different parts of the hierarchy. Sensor data of a sensing service is stored in a single XML document which is fragmented and distributed over a potentially large number of machines. Xpath (a standard query language for XML) is used to access the document as a single queriable unit. With X-Tree, user-provided code and derived states can be seamlessly incorporated into Xpath queries.

There are three main contributions of this paper. First, we propose X-Tree Programming, a novel database-centric programming model for wide-area sensor systems. Our X-Tree solution addresses the challenge of finding a sweet spot among three important, yet often conflicting, design goals: expressiveness, ease of programming, and efficient distributed execution. As we will show in Section 2, achieving these three goals in the same design is difficult. X-Tree's novelty comes from achieving a practical balance between these design goals, tailored to wide-area sensor systems. Second, we present important optimizations within the context of supporting X-Tree that reduce the computation and communication overheads of sensing services. Our caching technique, for example, provably achieves a total network cost no worse than twice the cost incurred by an optimal algorithm with perfect knowledge of the future. Third, we have implemented X-Tree within IrisNet [2, 11, 13], a shared infrastructure for wide-area sensing that we previously developed. We demonstrate the effectiveness of our solution through both controlled experiments and real-world applications on IrisNet, including a publicly available distributed infrastructure monitor application that runs on 473 machines worldwide. A rich collection of application tasks were implemented quickly and exe-

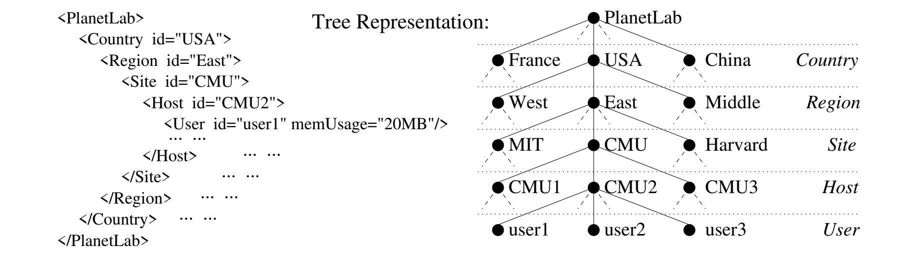



**Fig. 1.** An XML document representing IrisLog's logical hierarchy

cute efficiently, highlighting the expressibility, ease of programming, and efficient distributed execution that X-Tree provides.

The rest of the paper is organized as follows. Section 2 examines two application examples, describes the challenges and overviews our solution. After Section 3 provides background information, Section 4 illustrates the programming interface, Section 5 describes our system support for distributed execution, and Section 6 discusses optimizations. Our experimental evaluation is in Section 7. Section 8 discusses related work. Finally, Section 9 concludes the paper.

## 2 Example Applications, Challenges, and Our Solution

This section describes two representative wide-area sensing services (we use *service* and *application* interchangeably) that we aim to enable. (Additional examples can be found in [8]). We highlight the desirable properties of an enabling programming model, challenges in achieving them, and our solution.

### 2.1 Applications

**Person Finder.** A person finder application keeps track of the people in a campus-like environment (e.g., a university campus) and supports queries for the current location of a person or the current occupants of a room. The application uses sensors like cameras, microphones, smart-badges, etc. along with sophisticated software (e.g., for face or voice recognition) to detect the current location of a person. For scalability, it is important that the sensor data is stored near their sources (i.e., stored by location) and is retrieved only on-demand. One way to implement this application would be to maintain a distributed database of all the people currently at each location. A query for some person would then perform a brute force search of the entire database; such a query would suffer from both a slow response time and high network overhead. A far more efficient implementation would organize the distributed database as a location hierarchy (e.g., the root of the hierarchy is the university, and the subsequent levels are campus, building, floor, and room) and then prune searches by using approximate knowledge of people's current locations. Such pruning can be implemented by maintaining a Bloom filter (a compressed bit vector representation of a set—similar to [23]) at every intermediate node of the hierarchy, representing the people currently within that part of the location hierarchy.

**Infrastructure Monitor.** A distributed infrastructure monitor [1] uses *software sensors* [24] to collect useful statistics (e.g., CPU load, available network bandwidth) on the infrastructure's host machines and communication network, and supports queries on that data. One way to scale such an application to a large number of hosts is to hierarchically organize the data. Figure 1 (right) shows part of the hierarchy used by IrisLog, an infrastructure monitoring service deployed on 473 hosts in PlanetLab [3]. Infrastructure administrators would like to use such an application to support advanced database operations like continuous queries and distributed triggers. Moreover, they would like to dynamically extend the application by incorporating new sensors, new sensor feed processing, and new aggregation functions, as needs arise.

### 2.2 Design Goals and Challenges

A programming model suitable for the above applications should have the following properties. First, it should have *sufficient expressive power* so that application code can use arbitrary sensor data and perform a rich set of combinations and computations on that data. For example, applications may perform complex tasks (e.g., face recognition) on complex data types (e.g., images), and/or combine application-specific states (e.g., Bloom filters) with standard database queries. Second, the model should support *efficient distributed execution* of application code, executing a piece of computation close to the source of the relevant data items. This exploits concurrency in the distributed environment, and saves network bandwidth because intermediate results (e.g., the location of a person) tend to be much smaller than raw inputs (e.g., images of the person). Finally, the model should be *easy to use*, minimizing the effort of application developers. Ideally, a developer needs to write code only for the core functions of the application. For example, suppose she wants to periodically collect a histogram of the resource usage of different system components, but the infrastructure monitor currently does not support computing histograms. Then it is desirable that she needs to write only the histogram computing function, and have it easily incorporated within the monitor.

While achieving any one of the above goals is easy, it is challenging to achieve all three in a single design. For example, one way to provide sufficient expressive power is to enable collecting all relevant data items in order to perform centralized processing, and using application code to maintain states (e.g., Bloom filters) outside of the database. However, this approach not only rules out distributed execution, but it requires developers to integrate *outside* states into query processing—a difficult task. To understand the difficulty, consider the Bloom filters in the person finder application. To employ pruning of unnecessary searches, an application developer would have to write code to break a search query into three steps: selecting the roots of subtrees within the hierarchy for which Bloom filters are stored using the database, checking the search key against the Bloom filters outside of the database, and then recursively searching any qualified subtrees again using the database. This is an onerous task.

Similarly, consider the goal of efficient distributed execution. Distributed execution of aggregation functions (mainly with an SQL-style interface) has been studied in the literature [4, 14, 18]. The approach is to implement an aggregation function as a set of accessor functions (possibly along with user-defined global states) and to distribute them. However, it is not clear how to distribute application code for a large variety of possible application tasks that may access and combine *arbitrary* data items and application-specific states. For example, under the existing approaches, it is difficult to associate user-defined states (e.g. filters) with *subsets* of sensor readings. One could argue that application developers should implement all aspects of the distributed execution of their code. However, this approach requires developers to track the physical locations of stored sensor data and manage network communications, thus violating the goal of ease of programming.

### 2.3 Our Solution

We observe that although there are a large variety of possible application tasks, many tasks perform similar kinds of computations. For example, a common computation paradigm is to combine a list of sensor inputs of the same type (e.g., numeric values) to generate a single result (e.g., a histogram). Other common computation paradigms include (1) computing multiple aggregates from the same set of data sources, and (2) performing a *group-by* query, i.e., grouping data into classes and computing the same aggregate for each class (e.g., computing the total CPU usage on all the machines in a shared infrastructure for every user). Therefore, our strategy is to *provide a higher level of automation for common computation paradigms*. In this regard, we are similar to previous approaches [4, 14, 18].

As mentioned in Section 1, X-Tree employs XML to organize data into a logical hierarchy, and the Xpath query language for querying the (distributed) XML database, and hence requires techniques suitable for a hierarchical data model, unlike previous approaches. To enable user-defined computations with XML and Xpath, it provides two components. First, for common computation paradigms, X-Tree provides a *stored function* component with a simple Java programming interface and extends the Xpath function call syntax for implementing and invoking application code. Our implementation of X-Tree (denoted the X-Tree system) automatically distributes the execution of this application code. Second, X-Tree provides a *stored query* component that allows application developers to define derived states and to associate Xpath queries and application codes with XML elements. In this way, developers can guide the distribution of their code in the logical hierarchy of an XML document for *arbitrary* application tasks, without worrying about the physical locations of sensor data and or any needed network communication. Note that the physical locations of the sensor data can change over time (e.g., for the underlying system's load balancing, caching, etc.). Our implementation of X-Tree works regardless of these dynamics and hides them from developers.

## 3 Background: XML Model and Distributed Query Processing

An XML document defines a tree: Each XML element (tag-pair, e.g., `<PlanetLab>`, `</PlanetLab>`) is a tree node, and its nested structure specifies parent-child relationships in the tree. Every XML element has zero or more attributes, which are name-value pairs. Figure 1 illustrates the XML document representing the logical hierarchy in IrisLog. The root node is PlanetLab. It has multiple country elements as child nodes, which in turn are parents of multiple region elements, and so on. The leaf nodes represent user instances on every machine.

We can use Xpath path expressions to select XML elements (nodes) and attributes. In Xpath, "`/`" denotes a parent-child relationship, "`//`" an ancestor-descendant relationship, and "`@`" denotes an attribute name instead of an XML element name. For example, `/PlanetLab/Country[@id="USA"]` selects the USA subtree. An individual sensor reading, which is usually stored as a leaf attribute, can be selected with the whole path from root. `//User[@id="Bob"]/@memUsage` returns Bob's memory usage on every machine that he is using, as a list of string values. In order to compute the total memory usage of Bob, we can use the Xpath built-in function "sum": `sum(//User[@id="Bob"]/@memUsage)`. However, the handful of built-in functions hardly satisfy all application needs, and the original centralized execution mode suggested in the Xpath standard is not efficient for wide-area sensor systems.

IrisNet [2, 11, 13] supports distributed execution of Xpath queries, excluding functions. We highlight some of the features of IrisNet that are relevant to this paper; our description is simplified, omitting various IrisNet optimizations–see [11, 13] for further details. Sensor data of a service is conceptually organized into a single XML document, which is distributed across a number of host machines, with each host holding some fragment of the overall document. Sensing devices, which may also be hosts of XML fragments, process/filter sensor inputs to extract the desired data, and send update queries to hosts that own the data. Each fragment contains specially marked dummy elements, called *boundary elements*, which indicate that the true element (and typically its descendant elements) reside on a different host. IrisNet requires an XML element to have an `id` attribute unique among its siblings. Therefore, an element can be uniquely identified by the sequence of `id` attributes along the path from itself to the document root. This sequence is registered as a DNS domain entry, for routing queries.

To process an XML query, IrisNet extracts the longest query prefix with `id` attribute values specified and constructs an `id` sequence. Then, it performs a DNS lookup and sends the query to the host containing the element specified by the prefix; this host is called the *first-stop host*. The query is evaluated against the host's local fragment, taking into account boundary elements. In particular, if evaluating the query requires visiting an element $x$ corresponding to a boundary element, then a subquery is formed and sent to a host storing $x$. Each host receiving a subquery performs the same local query evaluation process (including recursively issuing further subqueries), and then returns the results back to the first-stop host. When all the subquery results have been incorporated into the host's answer, this answer is returned.

In the following, we describe X-Tree Programming within the context of IrisNet. However, we point out that our solution is applicable in any XML-based database-centric approach that supports in-network query processing.

## 4 X-Tree Programming

This section describes the two components of X-Tree, stored functions and stored queries, for efficiently programming wide-area sensing services.

### 4.1 Stored Functions

The stored function component incorporates application-specific code. Its programming interface is shown in Figure 2. A stored function can be invoked the same way as a built-in function in a user query, as shown in Figure 2(a). The colon separated function name specifies the Java class and the method major name of the application code. The semantics is that the *Input_XPATH* expression selects a list of values from the XML document, the values (of type String or Node, but not both) are passed to the stored function as inputs, and the function output is the result of the invocation. Optional arguments to a

(a) User query interface to invoke a stored function:

*myClass:mySF ( Input_XPATH, arg0, arg1, . . . )*

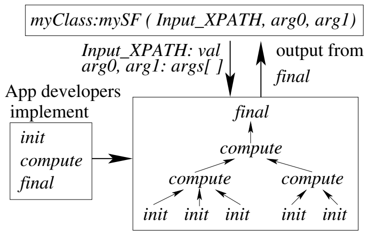
(b) Developers implement three methods for the stored function:

```
class myClass {
  String mySF_init (TYPE val, String[] args)
  // convert an XPATH output value into intermediate format.

  String mySF_compute (String[] midVals, String[] args)
  // perform computation on a set of intermediate values
  // merging them into a single intermediate value.

  String mySF_final (String midVal, String[] args)
   // generate the query result from an intermediate value.
}
// if Input_XPATH selects attributes, then TYPE is String
// if Input_XPATH selects nodes, then TYPE is Node
```

(c) Full picture:




The X−Tree system automatically distributes the execution of the three methods to relevant hosts

**Fig. 2.** Stored function programming interface

```
myClass:histogram ( Input_XPATH, "bucket boundary 0", "bucket boundary 1", ... )
// A set of numeric values is selected to compute the histogram.
class myClass { // args[ ] specifies the histogram bucket boundaries.
  String histogram_init (String val, String[] args)
  // determine bucket B for val, create an intermediate histogram with B's count set to 1 and all other counts set to 0.
  String histogram_compute (String[] midVals, String[] args)
  // merge multiple intermediate histograms given by midVals[ ] by summing up the counts of corresponding buckets.
  String histogram_final (String midVal, String[] args)
  // generate the query result from the final intermediate histogram.
}
```

**Fig. 3.** Implementation of a histogram aggregate

stored function are typically constant parameters, but can be any Xpath expressions returning a single string value.

As shown in Figure 2(b), application developers implement three Java methods for a stored function: *init*, *compute*, and *final*, which enable the decomposition of the stored function computation into a series of calls to the three methods. For each output value of the *Input_XPATH* expression, the *init* method is called to generate an intermediate value. Intermediate values are merged by the *compute* method until a single intermediate value is left, which is then converted to the query result by the *final* method. The *args* array contains the values of the arguments in the query. As shown in Figure 2(c), our X-Tree system automatically performs this decomposition and distributes the execution of the methods to relevant hosts where the data are located.[3]

Stored functions support the ability to perform computation on a single list of values selected by an Xpath query. Examples are numeric aggregation functions, such as sum, histogram, and variance, and more complex functions, such as stitching a set of camera images into a panoramic image [8]. Figure 3 illustrates the implementation of a histogram aggregate. Here, the *Input_XPATH* query selects *attributes* and thus the *init* method uses String as the type of its first parameter. However, because arbitrary data structures can be encoded as Strings, the interface is able to handle complex inputs, such as images.

[3] For stored functions (e.g. median) that are difficult to decompose, application developers can instead implement a *local* method which performs centralized computation.

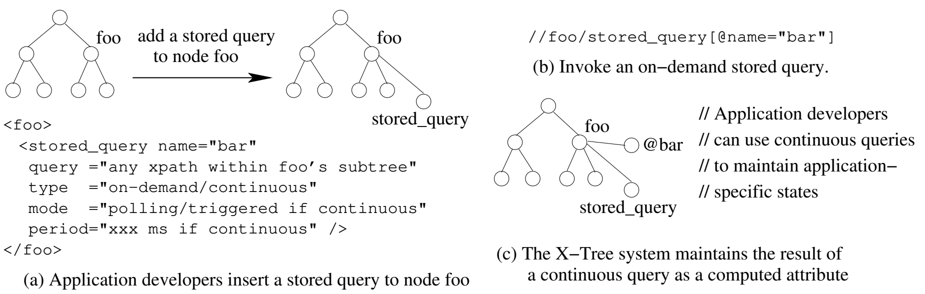



**Fig. 4.** Defining and using stored queries

Compared to previous approaches for decomposing aggregation functions [4, 18], our approach supports more complex inputs. For example, it allows computing functions not just on values but also on XML *nodes* (Node as input type), which may contain multiple types of sensor readings collected at the same location (e.g. all kinds of resource usage statistics for a user instance on a machine). This improves the expressiveness of the query language; for example, several common computation paradigms (e.g., computing multiple aggregates from the same set of data sources, and performing group-by operations, as described in Section 6) can be specified within a given Node context of the logical hierarchy.

### 4.2 Stored Queries

Stored queries allow application developers to associate derived states with XML elements in a logical hierarchy. Naturally, states derived from a subset of sensor readings can be associated with the root of the smallest subtree containing all the readings.

These derived states can be either maintained automatically by our system or computed on demand when used in queries. As shown in Figure 4(a), application developers define a stored query by inserting a stored query sub-node into an XML element. The stored query has a name unique within the XML element. The query string can be any Xpath query. In particular, it can be a stored function invocation, and therefore developers can associate application codes with logical XML elements.

Figure 4(b) shows how to invoke an on-demand stored query. For each foo element, our system retrieves the stored query string. Then it executes the specified query within the context of the subtree rooted at the parent XML element (e.g., the foo element).

For a continuous stored query, several additional attributes need to be specified, as shown in Figure 4(a). The query can be either in the polling mode or in the triggered mode. When the query is in the polling mode, the X-Tree system runs the query periodically regardless of whether or not there is a data update relevant to the query. When the query is in the triggered mode, the system recomputes the query result only when a relevant XML attribute is updated. As shown in Figure 4(c), the result of a continuous query is stored as a computed attribute in the database, whose name is the same as the stored query name. A computed attribute can be used in exactly the same way as a standard XML attribute. When adding a stored query, developers can also specify a duration argument. The X-Tree system automatically removes expired stored queries.

To support continuous stored queries, we implemented a continuous query scheme similar to those in Tapestry [27], NiagaraCQ [7], and Telegraph CACQ [20]. However, what is important for developers is that they can seamlessly use application-specific states in any queries, including stored function invocations and other stored query declarations, thus simplifying application programming.

### 4.3 Bottom-up Composition of Application Tasks

Combining stored functions and stored queries, our solution supports bottom-up composition of application tasks that may combine arbitrary data items and application-specific states. This is because:

- Application-specific states can be implemented as computed attributes and used in exactly the same way as standard XML attributes.
- The *Input_XPATH* of a stored function may be an on-demand stored query invocation; in other words, an on-demand stored query higher in the XML hierarchy may process results of other on-demand stored queries defined lower in the XML hierarchy. This gives application developers the power to express arbitrary bottom-up computations using any data items in an XML document.

X-Tree Programming saves developers considerable effort. Developers need not worry about unnecessary details, such as the physical locations of XML fragments, network communications, and standard database operations. They can simply write code as stored functions and invoke stored functions in any user query. They can also associate derived states with any logical XML elements without worrying about the computation and/or maintenance of the states.

## 5 Automatically Distributed Execution of Application Code

Stored functions and stored queries can be dynamically added into applications. Developers upload their compiled Java code to a well-known location, such as a web directory. When a stored function invocation (or subquery) is received at a host machine, the X-Tree system will load the code from the well-known location to the host.

Given a stored function invocation, the X-Tree system automatically distributes the execution of the *init*, *compute*, and *final* methods to the relevant hosts where the data are located. The idea is to call the accessor methods along with the evaluation of the *Input_XPATH* query, which selects the input data.

As shown in Figure 5(a), the stored function invocation is sent to the first-stop host of the *Input_XPATH* query (hosting the leftmost fragment in the figure). The system employs the standard query processing facility (in our case, provided by IrisNet) to evaluate the *Input_XPATH* query against the local XML fragment. As shown in Figure 5(b), the results of querying the local fragment mainly consist of two parts: i) local input data items (squares in the figure), and ii) boundary elements (triangles in the figure) representing remote fragments that may contain additional input data.

Next, the system composes a remote subquery for every boundary element, as shown in the shaded triangles in Figure 5(b). There are two differences between a remote subquery and the original stored function invocation. First, the function name is appended with a special suffix to indicate that an intermediate value should be returned. Second, a *subXPATH* query is used for the remote fragment. Note that the latter is

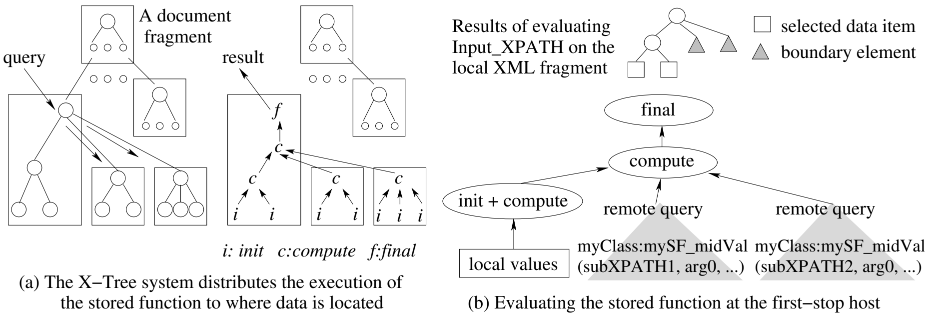



**Fig. 5.** Automatically distributed execution for *myClass:mySF(Input_XPATH, arg0, ...)*

obtained using the standard distributed query facility. The system then sends the subqueries to the remote fragments, which recursively perform the same operations. In the meantime, the system uses the *init* and *compute* methods to obtain a single intermediate value from the local data items.

Finally, when the intermediate results of all the remote subqueries are received, the system calls the *compute* method to merge the local result and all the remote results into a final intermediate value, and calls the *final* method to obtain the query result.

In summary, the X-Tree system automatically distributes the execution of stored functions to where the data are located for good performance. This scheme works regardless of the (dynamic) fragmentation of an XML document among host machines.

In addition to the above mechanism, application developers can use stored queries to guide the distributed execution of their code. They can define at arbitrary logical XML nodes stored queries that invoke stored functions. In this way, developers can specify the association of application-specific code to logical nodes in the XML hierarchy. Upon a reference of a stored query, the X-Tree system executes the stored function at the host where the associated logical XML node is located. Moreover, the stored function calls may in turn require results of other stored queries as input. In this way, developers can distribute the execution of their code in the logical XML hierarchy to support complex application tasks.

## 6 Optimizations

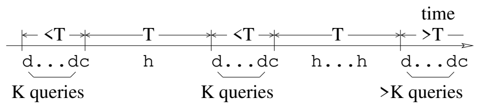
In this section, we first exploit the stored function Node interface to combine the computation of multiple aggregates efficiently and to support group-by. Then we describe a caching scheme that always achieves a total cost within twice the optimal cost.

### 6.1 Computing Multiple Aggregates Together and Supporting Group-By

In IrisLog, administrators often want to compute multiple aggregates of the same set of hosts at the same time. A naive approach would be to issue a separate stored function invocation for every aggregate. However, this approach uses the same set of XML nodes multiple times, performing many duplicate operations and network communications. Instead, like the usual mode of operation in any SQL-like language, the X-Tree system can compute all the aggregates together in a single query through a special stored function called *multi*. An example query, for the total CPU usage and maximum memory

```
if (tolerance time has passed) {
  Y = number of distributed executions
      in the previous T time
  if (Y < K)
       distributed execution (d)
  else centralized execution (c)
}
else  cache hit (h)
```

(a) Our algorithm assuming no future knowledge




(b) Example query patterns and choices of our algorithm

**Fig. 6.** Optimization for caching

usage of all users across all hosts, is as follows:

```
myOpt:multi(//User, "sum", "cpuUsage", "max", "memUsage")
```

The *init*, *compute*, and *final* methods for *multi* are wrappers of the corresponding methods of the respective aggregate functions. A *multi*'s intermediate value contains the intermediate values for the respective aggregate functions. Note that *multi* can be used directly in any application to compute any set of aggregates.

Using similar techniques, we have also implemented an efficient group-by mechanism, which provides automatic decomposition and distribution for grouping as well as aggregate computations for each group. Please see [8] for details.

### 6.2 Caching for Stored Functions

In IrisNet, XML elements selected by an Xpath query are cached at the first-stop host in hopes that subsequent queries can be answered directly from the cached data. Because stored queries may invoke the same stored function repeatedly within a short interval, the potential benefits of caching are large. To exploit these potential benefits, we slightly modify our previous scheme for executing stored functions. At the first-stop (or any subsequent) host, the system now has three strategies. The first strategy is the distributed execution scheme as before. Because IrisNet cannot exploit cached *intermediate* values, this strategy does not cache data. The second strategy is to execute the *Input_XPATH* query, cache all the selected XML elements locally, and execute the stored function in a centralized manner by invoking all the methods locally. The third strategy is to utilize existing cached data if it is not stale, and perform centralized execution without sending subqueries, thus saving network and computation costs. Cached data becomes stale because of updates. In IrisNet, a user query may specify a tolerance time T to limit the staleness of cached data. Associated with a piece of cached data is its creation time and the piece is used to answer the query only if this time is within the last T time units.

Our system has to choose one of the strategies for an incoming query. For simplicity, we shall focus on improving network cost. Assume for a given stored function invocation, the centralized strategy costs $K$ times as much as the distributed strategy, and the cost of a cache hit is $0$. Moreover, we assume all queries have the same tolerance time T. This defines an optimization problem: find an algorithm for choosing the strategy to evaluate each incoming stored function so that the total cost is minimized.

To solve this optimization problem, we propose the algorithm in Figure 6(a). This algorithm does not require any future knowledge. It only requires the X-Tree system to keep per-query statistics so that the $Y$ value can be determined. An example query pattern and the algorithm choices are shown in Figure 6(b). The algorithm performs distributed execution for the first $K$ queries, then centralized execution for query $K+1$, followed by a period of time T during which all queries are cache hits. Then the cached data is too stale, so the pattern repeats. An interesting, subtle variant on this pattern is when $> K$ consecutive queries use distributed execution (as shown in the rightmost part of the figure). This can arise because Y is calculated over a sliding time window. However, because the algorithm ensures that any $K+1$ consecutive distributed executions occur sparsely in a longer period of time than T, it is indeed better not to cache during these periods. We prove the following theorem in the full paper [8].

**Theorem 1.** *The algorithm in Figure 6(a) guarantees that the total cost is within twice the optimal cost.*

## 7 Evaluation

We have incorporated X-Tree Programming into IrisNet, and implemented the two applications discussed in Section 2. Although the Person Finder application is only a toy prototype, the Infrastructure Monitor application (IrisLog) has been deployed on 473 hosts in PlanetLab and has been publicly available (and in-use) since September 2003. The rich and diverse set of application-specific functions and states used by these applications (and others we studied [8]) supports the expressive power of X-Tree. The ease of programming using X-Tree is supported by the small amount of code for implementing these applications on our system: 439 lines of code for supporting Bloom filters in Person Finder and 84 lines of code for communicating with software sensors in IrisLog.

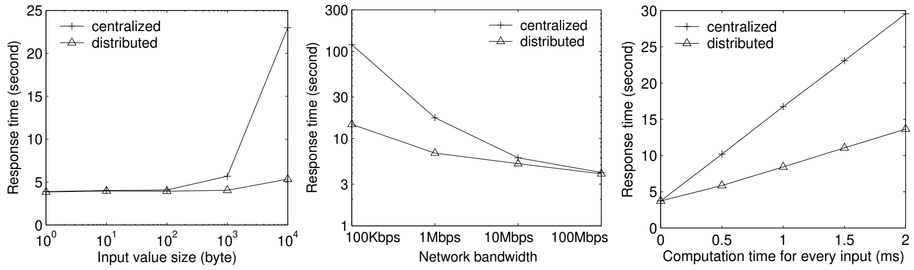
### 7.1 Controlled Experiments with Person Finder

We perform controlled experiments using the Person Finder application. For simplicity in understanding our results, we disabled IrisNet's caching features in all our experiments. We set up an XML hierarchy with 4 campuses in a university, 20 buildings per campus, 5 floors per building, 20 rooms per floor, and on average 2 people per room. Every room element contains a `name_list` attribute listing the names of the people in the room. We distribute the database across a homogeneous cluster of seven 2.66GHz Pentium 4 machines running Redhat Linux 8.0 connected by a 100Mbps local area network. The machines are organized into a three-level complete binary tree. The root machine owns the university element. Each of the two middle machines owns the campus and building elements for two campuses. Each of the four leaf machines owns the floor and room elements for a campus. In our experiments, we issue queries from a 550MHz Pentium III machine on our LAN and measure response times on this machine. Every result point reported is the average of 100 measurements.

**Stored Functions.** In order to quantify the improvements in response times arising from our scheme for distributed execution of stored functions, we compute an aggregate function using two different approaches. The first approach uses the *init/compute/final* programming interface, so that the computation is automatically executed in a distributed fashion. The second approach extracts all the relevant input values from the database and performs a centralized execution[4].

In order to show performance under various network conditions and application scenarios, we vary a number of parameters, including network bandwidth, input value size

[4] We actually implemented this approach using the alternative *local* method in our Java programming interface, which is equivalent to an implementation outside of the XML database system that runs on the root machine.




(a) Response time varying input value size

(b) Response time varying network bandwidth

(c) Response time varying computation time

**Fig. 7.** Distributed vs. centralized execution

to the stored function, and computation time of the function. The aggregation function we use models the common behavior of numeric aggregates (such as sum and avg), i.e. combining multiple input values into a single output value of similar size. For these experiments, every room element in the database contains a dummy attribute and the aggregations use this attribute. In order to make the size of input values a meaningful parameter to change, we choose to compute bit-by-bit binary OR on all the dummy attributes in the database and update every dummy attribute with a string of a given size before each experiment.

Figure 7(a) reports the response time of the two approaches while varying the length of every input value from 1 byte to 10,000 bytes. Centralized execution requires all input values to be transferred, while distributed execution only transfers intermediate results. As the input value size increases, the communication cost of the centralized approach increases dramatically, incurring large response time increases beyond 1000B. In contrast, distributed execution only suffers from minor performance degradations.

Figure 7(b) varies network bandwidth for the 100B points in Figure 7(a) in order to capture a large range of possible network bandwidth conditions in real use. The true (nominal) network bandwidth is 100Mbps. To emulate a 10Mbps network, we change the IrisNet network communication code to send a packet 10 times so that the effective bandwidth seen by the application is 1/10 of the true bandwidth. Similarly we send a packet 100 and 1000 times to emulate 1Mbps and 100Kbps networks. Admittedly, this emulation may not be 100% accurate since the TCP and IP layers still see 100Mbps bandwidth for protocol packets. Nevertheless, we expect the experimental results to reflect similar trends. As shown in Figure 7(b), when network bandwidth decreases, the performance gap between distributed and centralized execution increases dramatically. When network bandwidth is 1Mbps or lower, which is quite likely in a wide area network, distributed execution achieves over 2.5X speedups over the centralized approach.

Figure 7(c) studies the performance for computation-intensive aggregation functions. To model such a function, we insert a time-consuming loop into our aggregation function so that this loop is executed once for every input value in both the distributed and the centralized approaches. Then we vary the total number of loop iterations so that the whole loop takes 0, 0.5ms, 1ms, 1.5ms and 2ms, respectively, which models increasingly computationally intensive aggregation functions. As shown in Figure 7(c), distributed execution achieves over 1.7X speedups when the computation time is at

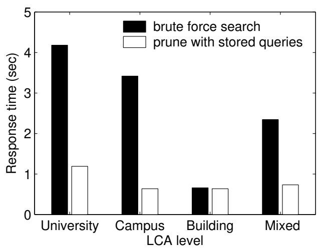
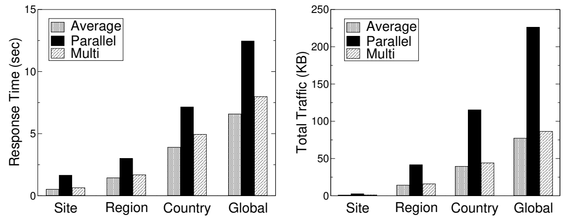



**Fig. 8.** Pruning vs. brute force




(a) Response time (b) Network traffic

**Fig. 9.** Calling multiple aggregates

least 0.5ms per input. This is because distributed execution exploits the concurrency in the distributed database and uses all seven machines, while the centralized approach performs all the computation on a single machine.

**Stored Queries.** To study the benefit of stored queries, we compare the performance of the brute force search approach and a pruning approach enabled by stored queries. For the latter, we use continuous stored queries to maintain Bloom filters at the building elements, and user queries check the Bloom filters using Xpath predicates. At a building element, (sub)queries that do not pass the Bloom filter check are pruned.

We use a 550MHz Pentium III machine for generating background update requests that model the movements of people. In the model, a person stays in a room for a period of time, which is uniformly distributed between 1 second and 30 minutes, then moves to another room. When making a move, the person will go to a room on the same floor, in the same building, in different buildings of the same campus, and in different campuses, with probabilities 0.5, 0.3, 0.1, and 0.1, respectively.

Figure 8 shows the performance comparison. We measure response times for queries that look for a person in the entire university, in a particular campus, or in a particular building. The mixed workload is composed of 20% university level queries, 30% campus level queries, and 50% building level queries. Because the scope (university, campus, or building) of a query is presumed to be an end user's good guess of the person's location, we set up the queries so that a query would succeed in finding a person within the given scope 80% of the time.

As shown in Figure 8, the Bloom filter approach achieves dramatically better performance than the brute force approach for queries involving campus or university level elements, demonstrating the importance of stored queries. The building level results are quite close because pruning is less effective in a smaller scope and additional stored procedure overhead almost offsets the limited benefit of pruning.

### 7.2 Real World Experiments with IrisLog

Our workload consists of queries with four different scopes. The *global* queries ask for information about all the PlanetLab hosts (total 473 hosts).[5] The *country* queries ask

[5] Although IrisLog is deployed on 473 PlanetLab hosts, only 373 of them were up during our experiments. The query latency reported here includes the timeout period IrisLog experiences while contacting currently down hosts.

about the hosts (total 290 hosts) within the USA. The *region* queries randomly pick one of the three USA regions, and refer to all the hosts (around 95 hosts per region) in that region. Finally, the *site* queries ask information about the hosts (around 4 hosts per site) within a randomly chosen USA site.

PlanetLab is a shared infrastructure; therefore all the experiments we report here were run together with other experiments sharing the infrastructure. We do not have any control over the types of machines, network connections, and loads of the hosts. Thus our experiments experienced all of the behaviors of the real Internet where the only thing predictable is unpredictability (latency, bandwidth, paths taken). To cope with this unpredictability, we ran each experiment once every hour of a day, and for each run we issued the same query 20 times. The response times reported here are the averages over all these measurements. We also report the aggregate network traffic which is the total traffic created by the queries, subqueries and the corresponding responses between all the hosts involved.

**Calling Multiple Aggregates Using Multi.** Figure 9 shows the performance of computing a simple aggregate (average), computing four different aggregates (average, sum, max, and min) using four parallel queries, and computing the same four aggregates using a single Multi query. For parallel queries, all the queries were issued at the same time, and we report the longest query response time.

From the figure, we see that the average query response time is small considering the number and the geographic distribution of the PlanetLab hosts. There exists a distributed tool based on Sophia [29] that can collect information about all PlanetLab hosts. Sophia takes minutes to query all the PlanetLab nodes [9]. In contrast, IrisLog executes the same query in less than 10 seconds.

Moreover, both the response time and the network overhead of the Multi operation are very close to those of a simple aggregation and are dramatically better than the parallel query approach. The Multi operation avoids the overhead of sending multiple (sub)queries, as well as the packet header and other common metadata in responses. It also avoids redundant selection of the same set of elements from the database.

We also studied the benefits on IrisLog of using our efficient group-by scheme. For a group-by query over all the nodes, our scheme achieves a 25% speedup in response time and an 81% savings in network bandwidth compared to the naive approach of extracting all the relevant data and computing group-by results in a centralized way [8].

## 8 Related Work

**Sensor Network Programming.** A number of programming models have been proposed for resource-constrained wireless sensor networks, including database-centric, functional, and economic models. The database-centric programming models (e.g., TinyDB [18, 19], Cougar [6, 32]) provide an SQL-style declarative interface. Like X-Tree, they require decomposing the target function into init/compute/final operators for efficient distributed execution. However, the resource constraints of the target domain have forced these models to emphasize simplicity and energy-efficiency. In contrast, X-Tree is a more heavy-weight approach, targeted at resource-rich Internet-connected sensing devices, where nodes have IP addresses, reliable communication, plenty of memory, etc.

The resource-rich target environment allows X-Tree, unlike the above systems, to sandbox query processing inside the Java virtual machine and to transparently propagate and dynamically load new aggregation operator code for query processing. Moreover, its XML data model allows posing queries in the context of a logical aggregation hierarchy. The functional programming models (e.g., programming with abstract regions [22, 30]) support useful primitives that arise in the context of wireless sensor network communication and deployment models. For example, the *abstract region* primitive captures the details of low-level radio communication and addressing. However, the requirements of wide-area sensing are different—generality is more important than providing efficient wireless communication primitives. Moreover, it is more natural to address wide-area sensors through logical hierarchy rather than physical regions. X-Tree aims to achieve these requirements. Proposals for programming sensor networks with economic models (e.g., market-based micro-programming's pricing [21]) are orthogonal to X-Tree. We believe that X-Tree can be used with such economic models, especially within a shared infrastructure (e.g., IrisNet [2, 13]) where multiple competing services can run concurrently.

**Distributed Databases.** Existing distributed XML query processing techniques [11, 26] support only standard XML queries. In contrast, X-Tree's query processing component supports user-defined operations. X-Tree leverages the accessor function approach for decomposing numeric aggregation functions [4, 18], and supports a novel scheme to automatically distribute the execution of stored functions. X-Tree's stored query construct has a similar spirit as the proposal for relational database fields to contain a collection of query commands [25]. The original proposal aims to support clean definitions of objects with unpredictable composition in a centralized environment. Because the logical XML hierarchy usually corresponds to real-life structures (such as geographical boundaries), X-Tree is able to support meaningful application-specific states computed from subsets of sensor readings. Moreover, stored queries can be seamlessly integrated into queries, and at the same time they can invoke application-specific code. This enables developers to compose arbitrary bottom-up computations, and to guide the distribution of application codes without knowing the physical layout of data.

**Distributed Hierarchical Monitoring Systems.** Astrolabe [28] allows users to use the SQL language to query dynamically changing attributes of a hierarchically-organized collection of machines. Moreover, user-defined aggregates can be installed on the fly. However, unlike X-Tree, it targets applications where the total aggregate information maintained by a single node is relatively small ($\approx$ 1 KB), and the aggregates must be written as SQL programs. Hi-Fi [12] translates a large number of raw data streams into useful aggregate information through a number of processing stages, defined in terms of SQL queries running on different levels of an explicitly-defined machine hierarchy. These processing stages can be installed on the fly. Both these systems target applications where aggregate data is continuously pushed toward the end users. Along with such *push-queries*, X-Tree targets *pull-queries* where relevant data is transferred over the network only when a query is posed. SDIMS [31] achieves a similar goal as Astrolabe by using a custom query language over aggregation trees built on top of a DHT. Moreover, it provides very flexible push vs. pull mechanisms. User-defined functions are more limited than with X-Tree, e.g., there does not appear to be an efficient means to perform bottom-up composition of distinct user-defined tasks. Finally, X-Tree differs from all three systems by using the XML data model and supporting a standard XML query language; thus it supports using a *logical* hierarchy that can be embedded on an arbitrary topology and a query language that incorporates the semantics of that hierarchy.

**Parallel Programming.** A number of programming models have been proposed to automatically parallelize computation within restricted target domains. For example, an associative function can be computed over all prefixes on an $n$ element array in $O(\log(n))$ time on $n/\log(n)$ processors using parallel prefix computations [5, 17]. In the context of LANs, the MapReduce model [10], like X-Tree, requires programmers to decompose the high level task into smaller functions. The MapReduce implementation then efficiently and robustly parallelizes the execution of those functions into thousands of machines in a single cluster. X-Tree can be considered as a simplification and distillation of some of these models based on our requirements. In particular, X-Tree provides efficient in-network aggregation (through the *compute* function, which MapReduce lacks), supports a standard query processing language, provides location transparency, and is targeted toward wide-area networks.

## 9 Conclusion

In this paper, we present *X-Tree Programming*, a novel database-centric approach to easily programming a large collection of Internet-connected sensing devices. Our solution augments the valuable declarative interface of traditional database-centric approaches with the ability to seamlessly incorporate user-provided code for accessing, filtering, and processing sensor data, all within the context of the hierachical XML database model. We demonstrate the effectiveness of our solution through both controlled experiments and real-world applications, including an infrastructure monitor application on a 473 machine worldwide deployment. Using X-Tree Programming, a rich collection of application-specific tasks were implemented quickly and execute efficiently, simultaneously achieving the goals of expressibility, ease of programming, and efficient distributed execution. We believe that X-Tree Programming will enable and stimulate a large number of wide-area sensing services.

## References


1. IrisLog: A Structured, Distributed Syslog. http://www.intel-iris.net/irislog.
2. IrisNet (Internet-scale Resource-Intensive Sensor Network Service). http://www.intel-iris.net/.
3. PlanetLab. http://www.planet-lab.org/.
4. F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. VLDB 1987*.
5. G. E. Blelloch. Scans as primitive parallel operations. *ACM Transaction on Computers*, C-38(11), 1989.
6. P. Bonnet, J. E. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proc. IEEE Mobile Data Management*, 2001.
7. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proc. SIGMOD 2000*.

8. S. Chen, P. B. Gibbons, and S. Nath. Database-centric programming for wide-area sensor systems. Technical Report IRP-TR-05-02, Intel Research Pittsburgh, April 2005.
9. B. Chun. PlanetLab researcher and administrator, http://berkeley.intel-research.net/bnc/. Personal communication, November, 2003.
10. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI 2004*.
11. A. Deshpande, S. K. Nath, P. B. Gibbons, and S. Seshan. Cache-and-query for wide area sensor databases. In *Proc. SIGMOD 2003*.
12. M. J. Franklin, S. R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design considerations for high fan-in systems: The HiFi approach. In *Proc. CIDR'05*.
13. P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Irisnet: An architecture for a worldwide sensor web. *IEEE Pervasive Computing*, 2(4), 2003.
14. J. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Toward sophisticated sensing with queries. In *Proc. IPSN 2003*.
15. P. R. Kumar. Information processing, architecture, and abstractions in sensor networks. Invited talk, *SenSys 2004*.
16. J. Kurose. Collaborative adaptive sensing of the atmosphere. Invited talk, *SenSys 2004*.
17. R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. of the ACM*, 27(4), 1980.
18. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proc. OSDI 2002*.
19. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. SIGMOD 2003*.
20. S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. SIGMOD 2002*.
21. G. Mainland, L. Kang, S. Lahaie, D. C. Parkes, and M. Welsh. Using virtual markets to program global behavior in sensor networks. In *Proc. ACM SIGOPS European Workshop*, 2004.
22. R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proc. ACM Workshop on Data Management for Sensor Networks*, 2004.
23. S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proc. INFOCOM 2002*.
24. T. Roscoe, L. Peterson, S. Karlin, and M. Wawrzoniak. A simple common sensor interface for PlanetLab. PlanetLab Design Notes PDN-03-010, 2003.
25. M. Stonebraker, J. Anton, and E. N. Hanson. Extending a database system with procedures. *ACM Transactions on Database Systems*, 12(3), 1987.
26. D. Suciu. Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems*, 27(1), 2002.
27. D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *Proc. SIGMOD 1992*.
28. R. van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2), 2003.
29. M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *Proc. Hotnets-II*, 2003.
30. M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proc. NSDI 2004*.
31. P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc. Sigcomm'04*.
32. Y. Yao and J. Gehrke. Query processing in sensor networks. In *Proc. CIDR 2003*.