Action-Oriented Query Processing for Pervasive Computing

Wenwei Xue   Qiong Luo

Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon
Hong Kong, China
{wwxue, luo}@cs.ust.hk

Abstract

Pervasive computing applications monitor physical-world phenomena and integrate data acquisition, communication and actions across small, heterogeneous devices (e.g., smart sensors, network cameras and handheld devices). To ease the development of these applications, we propose to perform their tasks by executing action-embedded queries, which are continuous queries with operations towards devices. We extend SQL to allow applications to specify actions and action-embedded queries. We treat actions as first-class citizens (operators) in query execution plans, and investigate adaptive, cost-based optimization techniques for a single query as well as for multiple queries. We evaluate our prototype query processor, *Aorta*, using a pervasive lab monitoring application. The initial experimental results show that Aorta ensures correct application semantics, improves query response time and balances device workload.

1. Introduction

In pervasive (or ubiquitous) computing [26], various kinds of networked computing devices are embedded or mobile in the physical world and interact with the world seamlessly. For instance, a pervasive video surveillance application automatically operates a number of remotely controllable cameras for security monitoring in a building. Involving data acquisition, communication, as well as operations on physical devices (*actions*), pervasive computing applications are usually difficult to develop and optimize. In this paper, we study how to utilize database-style query processing to facilitate the development and optimization of pervasive computing applications.

First, we propose to use SQL to develop pervasive computing applications, since the declarative nature of SQL queries eases application development and allows for performance optimization. Recent work including Cougar [3][4][27] and TinyDB [17] has pioneered this approach by treating devices and sensor nodes as virtual tables and data from these devices as relational tuples. Furthermore, user-defined functions and stored procedures are prevalent in pervasive computing applications because actions on devices often have to be programmed in a language other than SQL. Therefore, we extend SQL to specify continuous queries embedded with device actions, which we call *action-embedded queries*. Correspondingly, we call our action-oriented query processor *Aorta*.

Next, we explore opportunities for optimization of action-embedded queries in Aorta. This optimization is necessary because action-embedded queries are concerned with physical-world *events* (e.g., object movement), which may be transient. Therefore, the response time of an action-embedded query determines if an event is caught in time. Furthermore, there may be many devices available for executing a single action and the performance can be optimized by selecting a suitable device. This device selection is appropriate for the application semantics in pervasive computing, where it is sufficient for one or a few available devices (as opposed to all devices) of the same type to respond to one event. For instance, it is sufficient for one camera or a few cameras in a lab to take a photo of a location of interest upon an event; it is unnecessary to have all available cameras take photos of the location at one point in time.

To select devices for actions, we adapt the traditional cost-based query optimization [22] for actions. Apparently, suitable cost metrics may vary from application to application in pervasive computing, such as

Proceedings of the 2005 CIDR Conference

response time, power consumption, and quality of action effect. As a first step, we define the cost metric to be the response time, a relatively general one. To estimate the cost of an action, we define a set of atomic operations with estimated costs and abstract the composition of actions in terms of atomic operations.

There are two unique, intertwined challenges for cost-based optimization of actions. First, the current physical status of a device may affect the cost of an action on the device. For instance, some actions (e.g., pan, tilt and zoom) on a PTZ (pan/tilt/zoom) network camera involve the movement of the camera head. As a result, the starting head position of a camera affects the cost of moving the head of the camera to a target location. Moreover, the execution of an action takes more time on a busy camera than on a less busy one. In some cases, an extremely busy camera may malfunction and fail to finish the execution of an assigned action. Second, an action may have side effects that change the physical status of a device. Take the PTZ cameras as an example again. After an action is executed on a camera, the head position of the camera may change, which in turn changes the cost of the subsequent action execution on this device.

Given these two challenges, adaptive query processing [15] is essential for action-embedded queries in pervasive computing, where the environment is dynamic and delays and failures of devices are common. Consequently, we interleave the optimization and the execution of an action-embedded query in an adaptive, cost-based manner: Whenever an execution of a query is triggered, our query processor examines the current physical status of the candidate devices, estimates the cost, and selects the best device to execute the action. This adaptation occurs in every execution of an action-embedded query so that the cost estimation of the action is up-to-date and the cost of the actual execution is optimized.

Optimization becomes more complex when multiple action-embedded queries that have the same embedded action are running concurrently in the system. In this case, we can perform group optimization, in which multiple action requests from different queries are grouped and assigned to available candidate devices in a batch. We define an *action request* as the request from a query for the execution of an action with instantiated input parameter values for the action. Furthermore, cost estimation must be done dynamically in the process of group optimization due to the side effects of actions. We have developed two simple but effective algorithms for group optimization of action-embedded queries in Aorta.

There are many interesting systems issues in building an action-oriented query processor for pervasive computing, most prominently, data communication and device synchronization. In this paper, however, we focus on query processing in Aorta because this is the core technology of a database approach to the development and optimization of pervasive computing applications.

We have developed an action-enabled monitoring application with our Aorta prototype system for the pervasive lab in our department. The pervasive lab was established for accommodating cross-area research activities as well as undergraduate final year projects on pervasive computing. It has rack-mounted PC servers, desktops with removable hard disks, and various devices such as sensors, cameras, phones, and PDAs. Faculty, students and other personnel with access permission can enter the lab anytime using their university ID cards (smartcards). Due to the diversity and dynamic nature of the lab, the action-enabled monitoring application is highly desirable for safety, security and management purposes. We use this application as an illustrative example throughout the paper.

The remainder of this paper is organized as follows. We describe the overall Aorta system architecture in Section 2. We briefly introduce our SQL extension for actions and action-embedded queries in Section 3. In Section 4, we present our optimization and execution techniques for action-embedded queries in detail. In Section 5, we evaluate these techniques using the pervasive lab monitoring application. We discuss related work in Section 6 and conclude our paper in Section 7.

2. System Architecture

The Aorta system consists of three major layers, as illustrated in Figure 1.

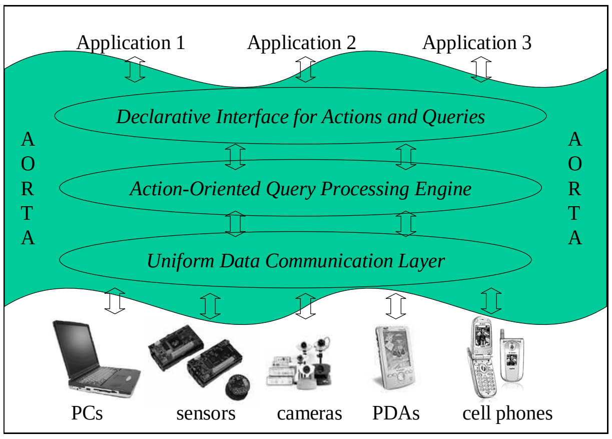
Application 1
Application 2
Application 3
Declarative Interface for Actions and Queries
Action-Oriented Query Processing Engine
Uniform Data Communication Layer
AORTA
AORTA
PCs
sensors
cameras
PDAs
cell phones

Figure 1: Three-layer illustration of Aorta

(1) A declarative interface that allows pervasive computing application programmers to specify actions towards heterogeneous devices through simple, declarative action-embedded queries. This interface alleviates the problem of programmers having to handle various programming APIs for specific types or models of devices. We extend the SQL language to capture action semantics and provide a library of system built-in methods (actions) for accessing and operating devices.

(2) A uniform data communication layer across heterogeneous devices. This layer ensures that the Aorta

system, not the individual applications, is responsible for monitoring and tuning the current network infrastructure and the physical settings of the devices. Similar to the Ingress and Caching modules in TelegraphCQ [7], this layer handles heterogeneous networking protocols and provides a dynamic, logical view of the networked devices for applications. This abstract view shields the lower-level implementation issues, such as data transmission loss, action failure and resource consumption on devices to enables the applications to focus on the real-world semantics of their tasks.

(3) An action-oriented query processing engine for queries involving actions. This engine is the core of our framework. Given the specifications of action-embedded queries received by the declarative interface and the facilities provided by the communication layer, the engine is responsible for generating, optimizing and executing action-embedded query execution plans. It interacts with the communication layer to adapt to device capacities and network loads. It also provides mechanisms to reduce device contention and to avoid malfunctioning devices.

In the remainder of this paper, we mainly present the top two layers in Figure 1.

3. The Declarative Interface

In this section, we describe the syntax and semantics of the query language provided by the Aorta declarative interface. We extend SQL to handle device actions and call our query language *AortaSQL*. The three main commands of AortaSQL are CREATE, DROP, and ALTER. They can be used for either actions (ACTION) or action-embedded queries (AQ).

Actions in Aorta are system-provided or user-defined methods/functions that operate devices. For a user-defined action, the user must pre-compile the code of the action into a dynamically linked library, and use the command CREATE ACTION to register the action along with an action profile to Aorta. An action profile is an XML text file that describes the high-level semantics of the action (more details will be discussed in Section 4). The following is an example of registering a user-defined action, which directs a programmable cell phone to ring.

```
CREATE  ACTION  ring(String phone_no)
AS  "admin/lib/ring.dll"
PROFILE  "admin/profiles/ring.xml"
```

An application can use the DROP command to drop an action that it has registered before:

```
DROP ACTION action_name
```

Both the executable and profile of the action will be removed.

The semantics of actions in pervasive computing are widely diversified. As the first step in investigating action-oriented query processing for pervasive computing, we currently consider only *single-device actions* (actions that involve a single device of some type). For *multi-device actions* (actions that involve multiple devices of various types), the coordination among the multiple devices involved in an action may be sophisticated, which makes the optimization of such actions more difficult. The extension of Aorta for handling multi-device actions is one direction of our ongoing work.

After an action is registered to the system, applications can then specify and register queries with the action embedded. Figure 2 shows the syntax of the CREATE AQ command, which defines an action-embedded query and registers the query to the system.

```
CREATE  AQ  aq_name  AS
          SELECT        select_list
          FROM          device_table_list
          [WHERE        where_condition]
          [GROUP BY     groupby_list]
          [HAVING       having_condition]
          [START        start_condition]
          [STOP         stop_condition]
          [LIFETIME     lifetime]
          [INVOCATION   num_invocations]
```

Figure 2: The CREATE AQ command in AortaSQL

The specification of an action-embedded query in AortaSQL appears to be an ordinary continuous query with a name and the optional START, STOP, LIFETIME and INVOCATION clauses. Considering that pervasive computing applications are often time-related, we designed a number of timer clauses. The START and STOP clauses specify when a query should start or stop. The LIFETIME clause describes how long the query will be kept in the system.

Aorta provides a set of system built-in variables and Boolean timer functions. Examples of system built-in variables include $now, $never, $forever, and $once. Examples of system-provided Boolean timer functions include every(interval_length, time_unit), inTimeInterval (start_time, end_time), and atTime(hour, minute, second). These variables and timer functions can be used in the WHERE clause as well as in the START and STOP clauses. The default conditions of the START, STOP and LIFETIME clauses are $now, $never and $forever.

The main difference between an action-embedded query and an ordinary continuous query is in the select_list, where an action may appear. Whenever the query condition is satisfied, the action is executed and the number of times of the execution is determined by the optional INVOCATION clause. This clause specifies how many times the embedded action is invoked whenever the query condition is satisfied. The default value in the clause is one. Applications can set the value in this clause to be an arbitrary number N or a system-provided variable $all, which means that the action will be invoked up to N times or on all candidate devices whenever the query condition is satisfied. Since the execution of an action is neither restricted to a specific device nor required on all candidate devices, this application semantics increases the reliability of action execution, saves system resources and creates opportunities for query optimization.

Putting these language constructs together, Figure 3 shows an example query, night_surveillance, for the pervasive lab monitoring application in AortaSQL.

```
CREATE  AQ  night_surveillance  AS
        SELECT  sendphoto(p.no,  photo(c.ip, s.loc,
                                  "photos/admin"))
        FROM    sensor  s, camera  c, phone  p
        WHERE   s.accel_x > 500
        AND     coverage(c.id, s.loc)
        AND     p.owner = "admin"
        START   atTime (0, 0, 0)
        STOP    atTime (6, 0, 0)
```

Figure 3: The night_surveillance example in AortaSQL

In this example, the action *photo(camera_ip, location, directory_name)* operates the network camera with an IP address *camera_ip* to move its head to the direction pointing to *location* and take a medium-size photo; and then stores the photo to the directory *directory_name*. The file name will be dynamically assigned by the camera using the current system date and time. The action *sendphoto (phone_no, file_name)* first converts an image file named *file_name* into the format for phone display and then sends the converted image to the phone with a phone number *phone_no*. As illustrated in this example, action nesting is supported in Aorta to enable complex interactions between devices. The constraint *s.accel_x > 500* in the query condition monitors the physical-world events of interest (e.g., someone pushes the door). The candidate cameras and phones are specified by the constraints *coverage(c.id, s.loc)* and *p.owner = "admin"*.

Action-embedded queries are "backward-compatible" with ordinary continuous queries and snapshot queries. This compatibility is intuitive, since data acquisition is essentially a trivial type of action. For instance, if there is no action in the select_list, a CREATE AQ command creates a traditional continuous query. Furthermore, if the LIFETIME of an AQ is specified to be $once, this AQ is a snapshot query that executes only once during its lifetime.

Finally, we give examples of the ALTER AQ and DROP AQ commands, which modify and remove pre-defined action-embedded queries, respectively. The following command stops a query immediately:

ALTER AQ aq_name SET STOP $now

If the application that defined a query does not need the query any more, it can use the following command to remove it from the system:

DROP AQ aq_name

4. Action-Oriented Query Optimization and Execution

Based on the application semantics for action-embedded queries, in this section we present our action-oriented query processing techniques using a simple example (see Figure 4). This snapshot query is a simplification of the night_surveillance example in Figure 3. In spite of its simplicity, this query is sufficient for illustrating the main design approach of our query optimizer and the related issues. This example is also used for performance evaluation in our experiments.

```
CREATE  AQ  snapshot  AS
        SELECT  photo(c.ip, s.loc, "photos/admin")
        FROM    sensor  s, camera  c
        WHERE   s.accel_x > 500
        AND     coverage(c.id, s.loc)
```

Figure 4: The snapshot query

4.1 Query Plan Generation and Execution

We have implemented a preliminary query operator framework in Aorta. Most of the query operators are relational, e.g., selection, projection and join operators, since data in Aorta are all in the form of relational tuples even though the attributes can be of less traditional data types, such as images.

Different from a traditional query optimizer, Aorta makes actions operators in query execution plans. An action operator contains the name, the input parameters, and the code block of the method to be executed. Figure 5 illustrates the query plan of the snapshot query in Figure 4. For simplicity, projections are omitted.

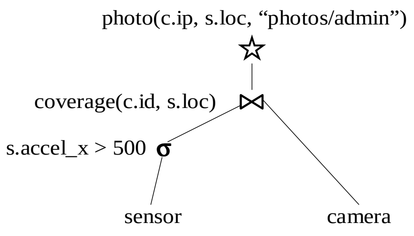

Figure 5: Query plan of the snapshot query

The execution flow of this query plan is as follows: First, the sensor virtual table is scanned and filtered to see if there are sensors that detect an x-axis acceleration rate larger than 500. When a sensor tuple is pushed from the sensor scan operator, it is used to join (probe) the camera virtual table to find cameras whose view ranges cover the sensor's location. Finally, the IP addresses of these cameras and the location of the sensor are passed to the action operator, which selects one suitable camera to take the photo.

Similar to the sensor and the camera scan operators in this query plan, for each type of device involved in Aorta, a scan operator is provided by the uniform communication layer for the query processor to access the corresponding virtual device table. The function coverage() is provided by the Aorta system. It shields applications from the change of the specific geographical location system adopted by Aorta.

In this query plan, the smart sensors are the *triggering devices*, which keep on sampling real-time values of the accel_x attribute with a system-tuned sampling period. Due to the asynchronous nature of event occurrence, a push queue is required to connect the sensor scan operator to its upstream operator. In comparison, the *triggered devices* on which the action is executed, i.e., the cameras in this query plan, need a traditional pull queue for the camera scan operator. We have implemented both types of queues in our query processing framework.

From this simple example we observe that, due to the event-driven nature, the execution flow of a single action-embedded query plan in Aorta is usually quite fixed. Consequently, there is little space for traditional plan enumeration [22], since the positions of operators in a query plan cannot be switched. Furthermore, different from traditional user-defined functions, it is generally unintuitive or impossible to push down an action operator in a query plan, because the action must know the values of all its input parameters before it can execute.

Even though the reordering of operators in an action-embedded query plan is unlikely, we can optimize individual actions for their efficient execution. More specifically, upon an event we can select the best device for an action operator, since there are usually multiple candidate devices for executing the action and the default application semantics is to execute the action only once. This optimization of action operators is similar to selection/projection/join method selection in traditional query optimization. The major difference is that in pervasive computing we need to consider the physical status of devices when optimizing actions.

In the following, we present the action-oriented query optimization and execution techniques we have designed and implemented in our Aorta query processor. For simplicity, we say “a type of device” in short for “a type or model of device”.

4.2 Cost Estimation Model for Actions

As we take a cost-based approach for the optimization of actions, an immediate problem is how to estimate the cost of an action to be executed on a specific device. We believe that the optimizer should be able to seek optimization opportunities for an action in a general way without knowing the implementation details of the action, no matter whether the action is system-provided or user-defined. Therefore, we propose a generic cost model for actions.

The cost model for an action includes the following components: (1) a set of atomic operations on the type of device that this action involves, (2) the estimated costs of the atomic operations, (3) a grammar for specifying the composition of the action, (4) the profile of the action, and (5) the formulas for estimating the cost of the action. All components are system-provided except for (4), which is provided by the application that registers the action. We describe these five components in order.

Atomic Operations. For each type of device involved in Aorta, we define a set of *atomic operations* that the devices can execute. Examples of such atomic operations include: “take a photo of a specific size (small, medium or large)”, “turn the head by one degree in a specific direction (up, down, left or right)”, and “zoom in (or out) one level” for cameras; “receive a photo (or a text message) of a specific size” for phones; “beep once” and “blink once” for sensors; and “establish a connection” for all types of devices. As the name suggests, the atomic operations are the basic, non-dividable units of actions.

Estimated Costs of Atomic Operations. We use a number of homegrown testing programs to measure the estimated cost of each atomic operation for each type of device by executing the operation repeatedly. Each estimated cost is the average of one hundred independent runs. These estimated costs of atomic operations are stored as part of the device profiles and managed by the uniform communication layer. Our tests show that an atomic operation has almost the same estimated cost on devices of the same type.

We currently measure the cost of an atomic operation in terms of *response time*: the time required to execute the operation. The cost of an action execution on a device is defined in the same way. Other cost metrics may be more meaningful for some atomic operations, such as power consumption for sensor beeps and price for phone calls. We choose response time because it is a general cost metric and is suitable for a large number of atomic operations on various types of devices. In addition, no matter what cost metric is chosen in our model, the methodology for cost estimation is similar. It is very easy to extend our cost model to adopt different cost metrics for different actions, or to use a weighted combination of multiple cost metrics for an action.

As an example, for the AXIS 2130(R) PTZ network cameras [2] we used, we have defined atomic operations and estimated their costs as listed in Table 1.

Table 1: Estimated costs of atomic operations for AXIS 2130(R) PTZ network cameras

Atomic Operation	Estimated Cost (milliseconds)
Pan (per degree)	13
Tilt (per degree)	14
Zoom (per level)	0.36
Connection	110
Take a Small-Size Photo	30
Take a Medium-Size Photo	40
Take a Large-Size Photo	50

Action Composition. Based on the atomic operations on devices, we define the composition of an action using the following grammar:

action := operationSequence
*operationSequence := operationUnit(& operationUnit)**
operationUnit := operationSequence | operationSet | operation
*operationSet := operationUnit (|| operationUnit)**
*operation := atomicOperation (& atomicOperation)**

In the grammar, the symbol "&" stands for sequential execution and "||" for parallel execution. An action in Aorta is an *operationSequence* and an *operationSequence* is a number of *operationUnit*s executed sequentially. An *operationUnit* in turn can be an *operationSequence*, an *operationSet,* or an *operation.* An *operationSet* consists of a number of *operationUnit*s that are executed in parallel.

Finally, an *operation* is defined as the sequential execution of a number of identical atomic operations. In other words, an *operation* is to execute the same atomic operation multiple times sequentially. Our consideration for the identical-atomic-operation constraint is to make the composition of an operation as simple as possible and to leave more complex relationships among operations to the nesting of *operationSequence* and *operationSet.*

Action Profiles. Instead of inquiring about the low-level implementation details, Aorta requires applications to provide the high-level semantics of an action via an action profile. An action profile is an XML file that contains the following information about an action: (1) the name, input parameters, and involved device of the action and (2) the composition of the action. The DTD (Document Type Definition) of an action profile is similar to that specified in the grammar for the composition of an action.

As an example, Figure 6 shows a fragment of the action profile of the system-provided action photo(). The composition of the action is illustrated in Figure 7. For simplicity, operationUnit is omitted in Figures 6 and 7.

This action profile specifies the following information to the Aorta query optimizer:

(1) The action name is *photo* and it is towards the AXIS 2130(R) PTZ network cameras.

(2) The action has three input parameters, denoted as *$camera_ip*, *$location*, and *$directory_name*, in order.

(3) The action consists of a sequence of operations and operationSets, which in turn consist of *connect*, *takeMediumSizePhoto*, and a set of *pan*, *tilt*, *zoom* atomic operations.

(4) The pan, tilt, zoom operations are executed in parallel. The system-provided functions deltaPan(), deltaTilt() and deltaZoom() (the latter two are not shown in Figure 6) are used to compute the numbers of pan, tilt, zoom atomic operations needed on a candidate camera based on the current head position of the camera, the value of the $location input parameter of the action, and the geographical location system that Aorta adopts.

```
<actionProfile>
   <name>photo </>
   <params>
      <1>$camera_ip</>
      <2>$location</>
      <3>$directory_name</>
   </params>
   <device>AXIS 2130(R) PTZ Network Camera</>
   <operationSequence>
     <operation>
        <atomicOperation>connect</>
        <number>1</>
     </operation>
     <operationSet>
        <operation>
           <atomicOperation>pan</>
           <number>deltaPan($location)</>

              ⋮
```

Figure 6: Action profile of the photo() action

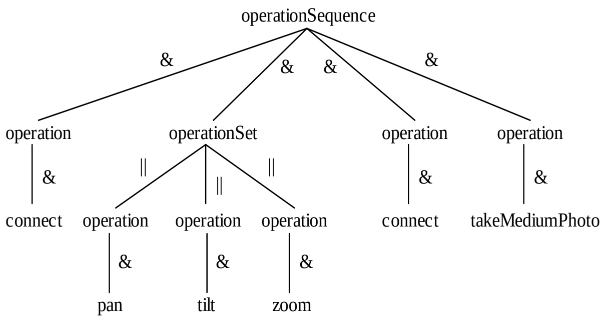

Figure 7: Composition of the photo() action

Cost Estimation Formulas for Actions. With the four pieces of information (the set of atomic operations, their estimated costs, the grammar for specifying action composition, and the action profile), the query optimizer is ready to estimate the cost of an action execution on a candidate device using the following cost estimation formulas ((1)-(4)):

$$C_{action} = C_{operationSequence} \tag{1}$$

$$C_{operationSequence} = \sum_{i=1}^{N} C_{operationUnit\_i} \tag{2}$$

$$C_{operationSet} = \underset{i=1}{\overset{N}{MAX}} \left(C_{operationUnit\_i} \right) \tag{3}$$

$$C_{operation} = C_{atomicOperation} * number \tag{4}$$

The calculation of these formulas is straightforward. The estimated cost of an action C_{action} is equal to the estimated cost of the top level operation sequence (Formula (1)). The estimated cost of an operation sequence $C_{operationSequence}$ is equal to the sum of the estimated costs of its operation units (Formula (2)). Since we use response time as the cost metric, the estimated cost of an operation set $C_{operationSet}$ is the maximum estimated cost of individual operation units in the set (Formula (3)).

Finally, the estimated cost of an operation $C_{operation}$ is equal to the total cost of the *number* of atomic operations it contains (Formula (4)). In Formula (4), $C_{atomicOperation}$ represents the estimated cost of an atomic operation on a type of device.

4.3 Cost-Based Optimization of a Single Query

Given the cost model for actions, we now describe how our optimizer works for the optimization of single action-embedded queries.

We continue to use the snapshot query in Figure 4 as the example. Suppose for an execution of this query, there are totally N candidate cameras for the action, and the optimizer is about to select one from them. The optimizer first connects to the candidate cameras *in parallel* and examines the current physical status of the devices, e.g., the current head position (the pan, tilt and zoom values) for the cameras. A TIMEOUT value is set to break connections to unresponsive devices. In our current implementation, the TIMEOUT value is set to be twice the estimated connection time for the type of device. We will see in Section 5 that this value worked well in practice. When the optimizer has received the responses from all of the cameras or the TIMEOUT limit is reached, this connection process ends.

From the composition of the photo() action (more specifically, the involved atomic operations) specified in its profile, the optimizer knows that the cost of a photo() action on a camera is mainly affected by the current head position and the network connection delay of the device. Note that for actions on different types of devices, the action cost may depend on different types of device physical status. As other examples, the current location of a robot affects the cost of moving it to a target location; and the depth of a sensor node in a multi-hop wireless network affects the cost of connecting the node.

We design the optimizer to poll the candidate devices for their current physical status due to the dynamic and unreliable nature of the device networks – physical devices in pervasive computing may join, move around or leave the network in a way unpredictable by the system. Cell phones, for example, may be turned on/off anytime and may become temporarily unavailable when moving into an area where no signals are received. Furthermore, it is possible that some devices also run applications that are not built on top of Aorta, which makes it difficult for our system to accurately keep track of and predict the current workload of the devices.

After getting sufficient information about the current physical status of the candidates, the optimizer begins to select one of them to execute the action. If all candidates seem to be unavailable (i.e., all connections are timed out), the optimizer will sleep for a while and then repeat the connection process to seek more candidate devices to execute the action. The optimizer gives up after repeating the connection process three times and an error message is returned to the application.

If things go well and M out of the N cameras are available to execute the action (i.e., their responses to the connection requests are received within the TIMEOUT limit), then for each camera i of these M cameras, the optimizer computes its estimated cost *Ci* to execute the action using the following formula:

$$\begin{aligned} C_i \;=\; & MAX\,(|\,P_{ti} - P_{ni}\,| * C_{pan},\ |\,T_{ti} - T_{ni}\,| * C_{tilt},\\ & |\,Z_{ti} - Z_{ni}\,| * C_{zoom}) + 2 * C_{connection\_i}\\ & + C_{takeMediumSizePhoto} \qquad (1 \le i \le M) \end{aligned}$$

Here P_{ni}, T_{ni}, Z_{ni} are the current pan, tilt and zoom values of candidate camera i. P_{ti}, T_{ti}, Z_{ti} are the target pan, tilt and zoom values specific to camera i corresponding to the sensor's location, which are computed in the deltaPan(), deltaTilt() and deltaZoom() functions, correspondingly. C_{pan}, C_{tilt} and C_{zoom} are the estimated costs of the pan, tilt, zoom atomic operations for this type of camera. The numbers of these atomic operations needed on camera i are $|\,P_{ti} - P_{ci}\,|$, $|\,T_{ti} - T_{ci}\,|$ and $|\,Z_{ti} - Z_{ci}\,|$ respectively, which are the output of the delta functions. $C_{connection\_i}$ is the connection cost of camera i recorded by the optimizer in the initial connection process. $C_{takeMediumSizePhoto}$ is the estimated cost of taking a medium-size photo. The optimizer will select the camera with the least C_i value to execute the action.

In the formula, the reason we use $C_{connection\_i}$ instead of the estimated cost $C_{connection}$ of the connection atomic operation for this type of camera is that cameras with a light workload are able to respond within the TIMEOUT limit but their connection time will be longer than those that are currently free. This is the way the optimizer estimates the current workload on candidate devices: a heavily-loaded device is very likely to require more time to respond to a connection request. As simple as it looks, this method is effective in practice.

The case that the query requires N executions of an action instead of only one can be optimized in a similar way. The optimizer sorts the candidate devices in the increasing order of the estimated cost for executing the action and picks the top N ones of them.

4.4 Grouping Multiple Action-Embedded Queries

Having considered selection of devices for an execution of a single query, we proceed to consider group optimization of multiple action-embedded queries. Group optimization in Aorta mainly focuses on balancing action workload on devices to improve system performance and to prevent device overloading.

In Aorta, we make queries that have the same embedded action (the functions are the same, but the input parameter values may be different) share a single action operator across their query execution plans. Each query is connected to the shared action operator by its own push-based output queue. We add the query ID to the output tuples of the query before they are passed to the shared action operator so that the operator knows which tuples are from which query. Sharing an action operator among multiple queries saves system resources and gives the optimizer a global view of the current system workload of an action. Instead of being optimized separately without any synchronization among them, multiple queries that have the same action are now grouped and optimized as a whole in the optimizer.

```
Algorithm 1: Action Workload Assignment
Input: A set of n action requests $R = (r_1, r_2, \dots, r_n)$ and a
       set of m devices $D = (d_1, d_2, \dots, d_m)$. Each request
       $r_i \in R$ has a set of candidate devices $D_i \subseteq D$. $(1 \le i \le n)$
Output: An assignment of R to D (each $r_i \in R$ is assigned
to a device $d \in D_i$) with the workload on D is balanced.
1.  For each device $d_i$ in D $(1 \le i \le m)$
2.    Initialize its assigned workload $W_i = 0$;
3.  $j = 1$;
4.  While there are unassigned requests {
5.      For each request r that j devices can service {
6.          For each of the j candidate devices for r
7.              Estimate the cost $C_{rk}$ to service r on
                device $d_k$, compute $E_k = W_k + C_{rk}$;
8.          Select device $d_l$ among the j candidate devices
            that has the least E value and assign r to $d_l$;
9.          Let $C_{rl}$ be the estimated cost to service r on $d_l$,
            $W_l$ += $C_{rl}$;
        }
10.     j++;
    }
```

Figure 8: The algorithm for action workload assignment

```
Algorithm 2: Prioritize and Service Multiple Action
             Requests on a Single Device
Input: A set of n action requests $R_d$ on a device d.
1.  While $R_d$ is non-empty {
2.      Record the current physical status $S_d$ of d;
3.      For each request $r \in R_d$
4.          Estimate the cost $C_r$ to service r in $S_d$;
5.      Select the request with the least estimated cost,
        service it (execute the action), and remove it from $R_d$;
    }
```

Figure 9: The algorithm for prioritizing and servicing multiple action requests on a single device

It is expected that Aorta, as a pervasive query processor, will always have a large number of queries running concurrently. In this scenario, multiple action requests from different queries may appear in a shared action operator at the same time or within a short time interval. In order to improve device utilization and query response time, we design and implement two algorithms to distribute multiple simultaneous action requests to available devices and to service these requests in an optimized order. These two algorithms are presented in Figure 8 and Figure 9.

The goal of the two algorithms is to minimize the maximum completion time of the set R of action requests (the interval between the time these requests appear in the shared action operator and the time all of them have been serviced) on the group of available devices D. This problem is similar to the classic makespan minimization problem [5] in scheduling theory that deals with unrelated parallel machines with sequence-dependent job setup time and machine eligibility restrictions. As the original problem is known to be NP-hard in general, we design and implement our own greedy algorithms.

Algorithm 1 assigns the action requests to the available devices with a goal of balancing workload among the devices. Note this algorithm is for request assignment only; a request that is assigned to a device is queued and is not serviced immediately. The heuristic for assignment is the number of candidate devices of each action request. More specifically, the algorithm starts with the request that has the least number of candidate devices, and assigns the request to the candidate device that will have the minimum estimated total workload if this request is serviced on the device. It then goes on to assign the request with the next least number of candidate devices until it finishes the assignment of all requests to devices. If two requests have the same number of candidate devices, the algorithm assigns the two requests in a random order.

After Algorithm 1 assigns a group of requests to a set of devices, for each device, Algorithm 2 prioritizes and services the requests that have been assigned to the device. The heuristic in Algorithm 2 is to service the request that has the least estimated cost in the current physical status of the device. As the execution of an action may change the physical status of a device, the physical status of the device is updated after selecting each request for the device to service.

The performance of these two algorithms is related to the system-defined grouping time interval T. The optimizer groups requests that fall into one grouping time interval. If T is large, it is likely that many action requests can be grouped, but the processing delay for individual requests may be large. In contrast, if T is too small, there may be very few action requests in each interval and the two algorithms are of little use. In our current implementation, we set T = 100 ms, which is about the average processing delay in one execution of the snapshot query in Figure 4 from the event being detected to the corresponding tuple(s) arriving at the input queue of the action operator.

In addition to enabling efficient group optimization for actions, sharing an action operator also gives the optimizer the opportunity of sharing the result of a single action execution among multiple requests from different queries. Such action result sharing is very important for load shedding when the system is heavily loaded with a large number of queries and their action requests. As a typical example, our optimizer can easily identify simultaneous requests with identical instantiated action input parameter values and invoke only one action on a device for all these requests. However, given the premise that the system should ensure the correct application semantics of the result of each action execution, a more complex result sharing mechanism seems to highly depend on the specific semantics of an action. We are currently investigating the feasibility and usefulness of designing general techniques in this regard.

5. Experiments

In this section, we use the pervasive lab monitoring application to validate the effectiveness of our proposed cost model and optimization techniques for actions. We mainly present our experimental results for action-embedded queries with the photo() action. We chose this action because it is representative and its effect is highly visible.

5.1 Experimental Setup

Our Aorta query processor was implemented using Java. The experiments involved one Pentium III PC, four Axis 2130(R) PTZ network cameras [2], and ten Berkeley MICA2 motes attached with MTS310CA sensor boards [10]. Two cameras were mounted on the ceiling of the pervasive lab and the other two were placed on desks. The ten sensor motes were put at ten different places in the pervasive lab. For instance, Mote 1 was attached to the front door of the lab and Mote 2 was put by the window. The view range of each camera covered the locations of a few motes. The location of each mote was in the view range of at least two cameras. We configured the cameras to tune their zoom level automatically based on the target location for taking photos. The purpose was to take photos with the same view size no matter how far a camera was from the target location, so that photos of the same location taken by two different cameras had almost the same visual quality.

5.2 Validation of the Cost Estimation Model

We first used the snapshot query in Figure 4 to verify our cost model for actions. In this experiment, all four cameras were started from their "home" positions: pan = 0, tilt = 0 and zoom = 1. We pushed the front door of the pervasive lab to let Mote 1 (attached to the door) get an x-axis acceleration rate larger than 500 so that an execution of the query was triggered. All cameras had no other workload.

Table 2: Estimated cost versus real cost of taking a photo on different cameras (milliseconds)

Camera ID	1	2	3	4
Estimated Cost	2993	3638	N/A	3347
Real Cost	3061	3682	N/A	3381

Table 2 shows the estimated cost and the real cost for executing the action on three cameras. The real cost was recorded by disabling the cost estimation module in the optimizer and letting the optimizer randomly pick one device for executing the action. All values in the table were the average of three independent runs.

In Table 2 we see that our cost model is reasonably accurate. The error rate of the estimated cost (defined as the ratio of the difference to the real cost) was around 1-2%. The order of the cameras by the estimated cost was the same as that by the real cost. The difference in absolute values between the estimated cost and the real cost was small. The estimated cost was consistently less than the real cost because the transition cost between operations in an action was omitted in the cost model.

Camera 3 had a long connection time due to some mechanical problem. We intentionally kept this malfunctioning camera in the experiment to see whether the optimizer could successfully identify a malfunctioning device. Its estimated cost was unavailable because the connection requests to it from the optimizer were always timed out. Its real cost was unavailable because it always failed to execute the action. This indicates that our optimizer could successfully distinguish devices that were having problems and avoided selecting them for executing actions.

We have run a set of other experiments with randomly generated initial head positions for each camera and with other sensor locations. In all experiments, the order of the cameras by the estimated cost was the same as that by the real cost, and the difference between the estimated cost and the real cost had an upper bound of 200 milliseconds. One set of these experimental results is shown in Table 3. The target location of the photo() action was the location of Mote 2, which was by the window of our pervasive lab. The reason that Camera 2 took a much longer time than the other two cameras was that the window was far from the camera and consequently the camera needed to enlarge its zoom level greatly, which took a lot of time.

Table 3: Cost of taking a photo with random initial camera head positions (milliseconds)

Camera ID	Initial Head Position	Estimated Cost	Real Cost
1	pan=21 tilt=-63 zoom=5001	3021	3077
2	pan=137 tilt=-33 zoom=1	5425	5595
4	pan=-75 tilt=-30 zoom=1	3770	3782

5.3 Optimization of a Single Action-Embedded Query

The main performance metric used in this study is the *response time* of one execution of an action-embedded query, which is defined as follows: the interval between the time an event is detected (i.e., an execution of the query is triggered) and the time the selected device finishes executing the action. This metric represents how fast a query can respond to an event occurrence.

For the AXIS 2130(R) PTZ network cameras used in our experiments, the pan, tilt and zoom ranges of them were [-169, 169], [-90, 10] and [1, 9999], respectively. Since these three types of operations are independent from each other and can be executed in parallel, in the extreme case the performance difference of two candidate cameras for executing a photo() action could be nearly 4.5 seconds. In the experiments that we did for verifying our cost model (see Section 5.2), the difference in response time between different candidate devices was 0.5 to 2 seconds in general.

One may wonder how pervasive computing applications can benefit from such a “slight” improvement in query response time. To illustrate this point, consider the snapshot query again. We set the head of Camera 1 to several different initial positions while making sure it always had the least estimated cost among the candidate cameras. As a result, the optimizer always selected Camera 1 to execute the action when Mote 1 attached on the front door detected a movement and triggered an execution of the query. We simulated the situation that someone was entering the pervasive lab at a normal speed (by pushing the front door from the outside) and ran tens of experiments. An execution of the query with a response time about 2.6 seconds always resulted in a photo similar to the one on the left of Figure 10, whereas an execution of the query with a response time about 3.2 seconds always resulted in a photo similar to the one on the right. This is simply because a physical-world event such as object movement may last only seconds or milliseconds.

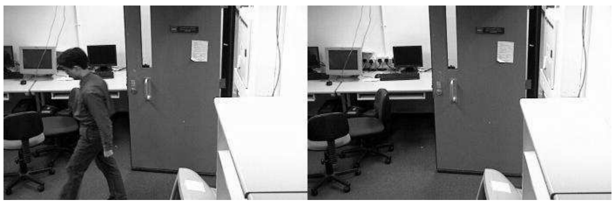
Figure 10: Photos taken by Camera 1 with different query response times

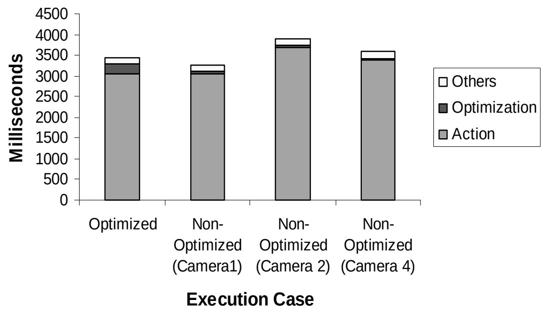

Figure 11: Time breakdown of four different executions of the snapshot query

Figure 11 shows the time breakdown of an execution of the snapshot query for both the optimized case and non-optimized cases. In the optimized case, the optimizer did action cost estimation for all candidate cameras and selected Camera 1 to execute the action. In the non-optimized cases, the cost estimation module was disabled and the optimizer randomly selected a camera, which could be any one of Cameras 1-4. The initial head positions of all cameras were set to their home positions.

In the figure, “Optimization” is the time the optimizer spent selecting a device from the candidates, “Action” is the time for the selected device to execute the action, and “Others” is the processing cost of the other operators in the query plan. The optimization time of a non-optimized case was smaller than that of the optimized case, since the optimizer was simply doing random selection. The optimization time of the optimized case shown in the figure was 220 milliseconds, due to the connection timeout of the malfunctioning Camera 3. We also ran a test with Camera 3 excluded and the optimization time was instead 70 milliseconds only. In comparison, the optimization time of a non-optimized case was 40 milliseconds.

In the figure, we also see that the cost of executing the action on the device dominated the query processing cost. Note that in a non-optimized case, when the optimizer happened to pick the device with the least cost for executing the action, the total query response time was slightly less than that of the optimized case. However, the optimization cost is tiny in comparison with the action cost. As a result, our optimization is beneficial since it trades a little larger processing cost at the server side for the possibly much smaller processing cost at the resource-constrained device side. Even when the total response time improvement is insignificant, our optimization techniques can prevent overloading of devices.

5.4 Optimization of Multiple Action-Embedded Queries

After examining the optimization issues of a single action-embedded query, we continue to investigate the impact of our group optimization techniques.

In the first experiment, we generated ten queries and registered them to the system. All queries were in the following format (1 ≤ i ≤ 10):

```
CREATE  AQ  test_query_i  AS
        SELECT    photo(c.ip, s.loc, “photos/test”))
        FROM      sensor  s, camera  c
        WHERE     s.id = i
        AND       coverage(c.id, s.loc)
        AND       every(1, minute)
```

In every minute, the i-th query requested a camera to take a photo of Mote i's location. In every grouping time interval, we found that our Algorithm 1 could distribute the ten action requests from these ten queries nearly uniformly to all three functioning cameras. Cameras 1, 2 and 4 were assigned 3, 4, and 3 requests, respectively.

Next, we examined the performance impact of prioritizing and servicing multiple action requests on a single device using Algorithm 2. We unplugged Cameras 2-4 so that all requests were directed to Camera 1. We created queries to periodically request the camera to take photos of the mote locations. The queries have the same format as the ten queries used in the previous experiment. We varied the hotness (the probability of being photoed) of the ten mote locations from uniformly distributed to highly skewed.

Since we considered multiple requests as a whole in this scenario, we recorded the total real cost of all requests. We compared the performance of the optimized case (using Algorithm 2) and the non-optimized case (Algorithm 2 was disabled in the optimizer and the requests were serviced in a random order on a device).

Figure 12 shows the total real cost for both the optimized and the non-optimized cases when the number of simultaneous action requests N on Camera 1 increased from 2 to 10. The hotness of the mote locations was skewed. The values for the non-optimized case were the average of ten runs. Taking an average was because in the non-optimized case Camera 1 serviced the requests randomly, so two runs might have a large variance in cost.

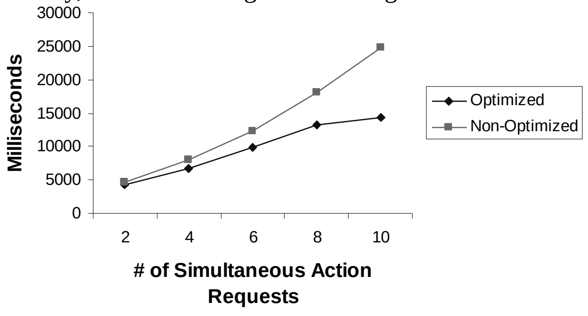

Figure 12: The total real cost of servicing multiple photo() action requests on Camera 1

The figure illustrates that in the non-optimized case, the total real cost for of all requests increased greatly with respect to N; whereas in the optimized case, such an increase was less significant and even became smooth when a threshold was reached (N = 8). This suggests that group optimization of multiple requests on a single device improves the overall response time and the device utilization. When the threshold was reached, the device was nearly fully loaded and utilized.

The results for the mote locations with uniformly distributed hotness were similar to those in Figure 12 and therefore were omitted from the paper.

6. Related Work

Recent work in pervasive computing focuses on networks of homogeneous devices, e.g. RFID (Radio Frequency Identification) tags [20] and cell phones [23]. In comparison, Aorta handles a network of heterogeneous devices. Work similar to ours is the Augmented Recording System [24], in which the authors proposed to actuate camera operations based on various sensor readings. However, this idea of actuation was not implemented in the paper and only sensory data collection methodology was presented. Moreover, Aorta takes the database query processing approach for the development and optimization of pervasive computing applications, which distinguishes itself from most existing work in pervasive computing.

In the database area, the design and implementation of Aorta has been influenced by a large body of recent research work on sensor databases and data stream management systems, including Aurora [6], Cougar [3][4][27], IrisNet [11], STREAM [19], Telegraph [7], and TinyDB [17]. These systems mainly deal with data flows, but have basic mechanisms for events and/or device actuation. In comparison, we have less emphasis on data flows but focus on optimizing and executing actions in networks of heterogeneous devices.

Actions are closely related to user-defined functions and stored procedures, which are widely supported in commercial DBMS products, such as IBM DB2 and Microsoft SQL Server. These functions and procedures seldom have the side effects on devices as actions in Aorta and usually run outside the core of the DBMS. Consequently, early work on optimization and execution techniques for expensive methods [16] and expensive predicates [8][14] has focused on caching the results or ordering the predicates in query execution plans. In comparison, we regard actions as first-class citizens in query processing and optimize them for the performance of their side effects on devices. In addition, our plan enumeration in query processing is on candidate device selection of individual action operators rather than ordering of operators.

Query optimization for minimizing response time has been previously studied in traditional parallel and distributed database systems [1][12]. Our optimization approach is specifically designed for pervasive computing, which mainly involves selecting the best candidate device for executing an action. Moreover, our proposed cost model is general and applicable to a wide range of cost metrics in addition to response time.

Finally, action-embedded queries are closely related to triggers [13][21] and continuous queries [9][18][25]. As a result, general group optimization techniques in these areas, such as predicate indexing or query indexing, are applicable to our system for testing the query conditions of a group of queries. Given the goal of our system on optimizing device actions for pervasive computing applications, we focus on considering the interplay between devices and actions in our work. This consideration makes our cost model different from others and our query optimization and execution process more dynamic. In addition, the adaptivity of our optimization of actions to the current physical status of the devices and our group optimization of actions share a similar spirit with CACQ [18] and NiagaraCQ [9].

7. Conclusion

We have presented the design and implementation of Aorta, an action-oriented query processor for pervasive computing. The goal of Aorta is to ease the development and the optimization of pervasive computing applications.

We have extended SQL for pervasive computing applications to specify their actions and action-embedded queries. We treat actions as first-class operators in query execution plans, and investigate adaptive, cost-based optimization techniques for them. We have proposed a cost model to estimate the cost of an action execution on candidate devices in terms of response time. We have also investigated group optimization techniques for multiple action-embedded queries that have the same action. Our experimental results with a pervasive lab monitoring application demonstrate that our cost model is reasonably accurate, and that our proposed single-query or multi-query optimization techniques ensure correct application semantics, improve query response time and balance device workload.

Future work includes extending our techniques for multi-device actions and actions towards new types of devices, studying more sophisticated group optimization techniques for action-embedded queries, and improving our query interface to be more general and expressive.

Acknowlegement

We thank Lionel M. Ni for his helpful input on this work. Funding for this work was from Grants HKUST6158/03E and HKUST6263/04E provided by the Hong Kong Research Grants Council (RGC).

References

[1] P.M.G Apers, A.R. Hevener, and S.B. Yao. Optimization Algorithms for Distributed Queries. IEEE Transaction on Software Engineering, Vol. 9, No. 1, 1983.

[2] Axis Communications. http://www.axis.com/.

[3] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Querying the Physical World. IEEE Personal Communications, Vol. 7, No. 5, October 2000.

[4] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. MDM 2001.

[5] Peter Brucker. Scheduling Algorithms. Third Edition, Springer Verlag, 2001.

[6] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring Streams – A New Class of Data Management Applications. VLDB 2002.

[7] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. CIDR 2003.

[8] Surajit Chaudhuri and Kyuseok Shim. Optimization of Queries with User-defined Predicates. VLDB 1996.

[9] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ – A Scalable Continuous Query System for Internet Databases. SIGMOD 2000.

[10] Crossbow Corp. http://www.xbow.com.

[11] Amol Deshpande, Suman Nath, Phillip B. Gibbons, and Srinivasan Seshan. Cache-and-Query for Wide Area Sensor Network. SIGMOD 2003.

[12] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query Optimization for Parallel Execution. SIGMOD 1992.

[13] Eric N. Hanson, Chris Carnes, Lan Huang, Mohan Konyala, Lloyd Noronha, Sashi Parthasarathy, J. B. Park, and Albert Vernon. Scalable Trigger Processing. ICDE 1999.

[14] Joseph M. Hellerstein. Optimization Techniques for Queries with Expensive Methods. TODS 1998.

[15] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Samuel Madden, Vijayshankar Raman, and Mehul Shah. Adaptive Query Processing: Technology in Evolution. IEEE Data Engineering Bulletin, Vol. 23, No. 2, June 2000.

[16] Joseph M. Hellerstein and Jeffrey F. Naughton. Query Execution Techniques for Caching Expensive Methods. SIGMOD 1996.

[17] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The Design of an Acquisitional Query Processor for Sensor Networks. SIGMOD 2003.

[18] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously Adaptive Continuous Queries over Streams. SIGMOD 2002.

[19] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. CIDR 2003.

[20] Kay Römer, Thomas Schoch, Friedemann Mattern, and Thomas Dübendorfer. Smart Identification Frameworks for Ubiquitous Computing Applications. PERCOM 2003.

[21] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, and C. Mohan. Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. VLDB 1991.

[22] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. SIGMOD 1979.

[23] Frank Stajano and Alan Jones. The Thinnest Of Clients: Controlling It All Via Cellphone. Mobile Computing and Communication Review, Vol. 2, No. 4, October 1998.

[24] Norman Makoto Su, Heemin Park, Eric Bostrom, Jeff Burke, Mani B. Srivastava, and Deborah Estrin. Augmenting Film and Video Footage with Sensor Data. PERCOM 2004.

[25] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous Queries over Append-Only Databases. SIGMOD 1992.

[26] Mark Weiser. The Computer for the 21st Century. Scientific American, September 1991.

[27] Yong Yao and Johannes Gehrke. Query Processing for Sensor Networks. CIDR 2003.