Abstract

In the next decade, millions of sensors and small-scale mobile devices will integrate processors, memory, and communication capabilities. Networks of devices will be widely deployed for monitoring applications. In these new applications, users need to query very large collections of devices in an *ad hoc* manner. Most existing systems rely on a centralized system for collecting device data. These systems lack flexibility because data is extracted in a predefined way. Also, they do not scale to a large number of devices because large volumes of raw data are transferred. In our new concept of a device database system, distributed query execution techniques are applied to leverage the computing capabilities of devices, and to reduce communication. In this article, we define an abstraction that allows us to represent a device network as a database and we describe how distributed query processing techniques are applied in this new context.

Querying the Physical World

Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri,
Cornell University

The widespread deployment of sensors, actuators, and mobile devices is transforming the physical world into a computing platform. We will soon find computing power, memory, and communication capabilities on temperature sensors and motion detectors, on door locks, light bulbs, and alarms, on every cellular phone, in every vehicle, and soon in every person's wallet or on their key ring. Emerging networking techniques ensure that devices are interconnected and accessible from local- or wide-area networks [1].

Using this new computing platform, users interact with portions of the physical world. In a large class of applications, users monitor phenomena in a given environment. Examples of monitoring applications include gathering information in a disaster area, supervising items in a factory warehouse, or controlling vehicle traffic in a large city [2, 3].

Let us take the concrete example of an existing flood detection system. For about twenty years now, the ALERT system has been deployed in several US states (http://www.alertsystems.org). A typical ALERT installation consists of several types of sensors in the field: rainfall sensors, water level sensors, weather sensors, etc. A predefined set of data is regularly extracted from each sensor, transferred to a central site and stored in a database system. Users query the database system through a graphical user interface. Here are some example queries that users can express: "For each rainfall sensor, display the average level of rainfall for 1999," Display the current level of rainfall for all sensors in Tompkins County, or "Every hour, display the location of the sensors where the level of rainfall is greater than 250 mm."

Query Processing over Device Networks

The example of the flood detection system emphasizes that monitoring is best described in a declarative manner: users submit queries concerning a device network regardless of its physical structure or its organization. In monitoring applications, users typically ask three kinds of queries:

- **Historical queries**: These are typically aggregate queries over historical data obtained from the device network, e.g., "For each rainfall sensor, display the average level of rainfall for 1999."
- **Snapshot queries**: These queries concern the device network at a given point in time, e.g., "Retrieve the current rainfall level for all sensors in Tompkins County."
- **Long-running queries**: These queries concern the device network over a time interval, e.g., "For the next five hours, retrieve every 30 seconds the rainfall level for all sensors in Tompkins County."

The existing ALERT system implements a *warehousing* approach, where data are extracted from the devices in a predefined way and stored in a centralized database system that is responsible for query processing. This warehousing approach is well suited for aggregate queries asked over historical data; however, it has two major limitations:

- **The warehousing approach dissociates access to devices from the query workload**. For instance, in an emergency situation, a fire department might require that specific data be accessed in a group of sites in order to decide on actions to take: "Every minute, display the rainfall level obtained from all sensors in Tompkins County." This long-running query cannot be answered in a traditional system if data is extracted from the sensors too infrequently. One solution would be to continuously extract all data from each device and transfer them to the database server. This solution is not feasible in practice because it might not be possible to extract all data from a sensor (e.g., a camera takes measurements in only one direction and it is not possible to measure data in all directions simultaneously) or because it is too expensive to transmit a continuous flow of raw data through the device network.
- **The warehousing approach uses valuable resources to transfer large amounts of raw data from devices to the database server**. Excessive resources are consumed at each device and on the network when transmitting large volumes of data. First, it is in general not necessary to extract data from the whole device network to answer a given query. In our example, the group of sensors sending data back to the database server should be reduced to sensors located in Tompkins County. Second, modern devices include processing capabilities that could be used to process data locally and thus reduce data transfer and energy consumption.

Our alternative to a warehousing approach is a *distributed* approach where the query workload determines the data that are extracted from remote sites, and where possibly portions

Praveen Seshradi is currently on leave at Microsoft.

 1070-9916/00/$10.00 © 2000 IEEE

of queries are executed on devices. This approach allows the database system to control the resources that are used; it is primarily targeted at snapshot and long-running queries. In addition, aggregate queries over historical data could be evaluated against materialized data stored on some devices instead of a centralized server. We call a database system that enables distributed query processing over a device network a *device database system*. We study such systems in the COUGAR Device Database Project at Cornell University.

The DataSpace project at Rutgers (http://www.cs.rutgers.edu/dataman/) recognized the advantages of the distributed approach over the warehousing approach for querying device networks [4]. In a DataSpace, devices encapsulating data can be queried, monitored, and controlled. Network primitives are developed to guarantee that only relevant devices are contacted when a query is evaluated.

Device Database Systems

In this article, we explain our new concept of a device database system, an area that we consider a very fruitful direction for new research. We will describe database abstractions for representing devices and we illustrate how queries are formulated in SQL with minimal additions to the language. Later, we use an example to show how distributed query processing techniques are applied in the new context of a device database system. We use an analytical model to illustrate the benefits of our approach.

We would like to point out that the methods described in this article represent the first generation of our system [5]. The core components of the first-generation COUGAR system are implemented and fully functional. We demonstrated the system at the Intel Computing Continuum Conference [6]. Note that in this article we do not address several of the specific research challenges that lie ahead, such as new query processing strategies to leverage computing capabilities on the devices, query processing strategies that adapt to changing conditions in the network, decentralized meta-data management, and administration. We overview these issues as we conclude.

Device Database Systems

We call a physical object with computing and communication capabilities a *device*. Some devices embed computing and communication capabilities (e.g., WINS sensor nodes [7], Smart Dust Motes [8], cell phones, or Smartcards) while others are composed of a physical object connected to an external computer (e.g., a door actuator connected to a desktop computer). Devices are interconnected and accessible from a local- or wide-area network. Some devices are stationary, others are mobile; some devices are always connected to the network, others intermittently. In this article we concentrate on stationary devices.

Database Abstractions for Representing Devices

In the warehousing approach, discussed earlier, devices are not part of the database system; they are accessed using a predefined extraction procedure that populates relations in the centralized database system. Our goal in a device database system is to access devices directly when processing queries. We thus need to represent devices in the database system.

Let us first refine our definition of devices. We consider that each device is a mini-server that supports a set of functions and can process portions of the queries directly at the device.[1] A function either (a) acquires, stores and processes data or (b) triggers an action in the physical world. Both kinds of functions return results (at least a status report or an error message). We distinguish between synchronous and asynchronous functions. Synchronous functions return results immediately, on-demand; they are used to monitor continuous phenomena, e.g., a function that returns the rainfall level. Asynchronous functions return results after an arbitrary period of time; they are used to monitor threshold events, e.g., a function that detects an abnormal rainfall level. Functions provided by an intermittently connected device can only return results when the device is connected; they are asynchronous functions. Stationary devices, e.g., rainfall sensors, may support both synchronous and asynchronous functions.

We need to represent the set of functions provided by devices at the database level. We distinguish two levels of representation:

- User representation — how are devices modeled in the database schema?
- Internal representation — how are devices represented internally?

User Representation — Today's object-relational and object-oriented databases support Abstract Data Type (ADT) objects that are single-attribute values encapsulating a collection of related data [9]. Note that there are natural parallels between devices and ADTs. Both ADTs and devices provide controlled access to encapsulated data through a well defined interface. We build upon this observation by modeling each type of device in the network as an ADT. The public interface of the ADT corresponds to the specific functions supported by the device. An actual ADT object in the database corresponds to a physical device in the real world.

Let us model the database schema corresponding to the flood detection example earlier. We consider a simplified schema that consists of the following relation:

RFSensors(Sensor, X, Y)

A record in the *RFSensors* relation has three attributes. The first attribute, called *Sensor*, is an ADT that represents the physical rainfall sensors. The actual sensor data is located on the rainfall sensor; the ADT *Sensor* provides functions for accessing the data. For example, *Sensor.getRainfallLevel()* returns the current level of rainfall measured in mm. The other two attributes denote the location of the sensor according to some coordinate system.

Internal Representation — Before discussing the internal representation of ADT functions, let us recall some background knowledge about query processing and the internals of a database system. Query processing classically proceeds as follows. The database system accepts a query, produces a query execution plan, executes this plan against the database, and produces the answer. The execution plan is the internal blueprint for evaluating a query. It combines algebraic operators (e.g., selection, projection, and join operators in the relational algebra), which serve as the basic building blocks for manipulating data (i.e., relations that are sets of records).

In object-relational database systems, ADT functions are either represented as expressions [9] or as joins involving virtual relations [10].[2] When an expression containing an ADT function is evaluated, a (local) function is called to obtain its

[1] Embedding a database server on a device is realistic. All major database vendors propose database servers for palm-sized PCs, which represent the processing capabilities that we can expect from all devices in a near future.

[2] Table functions defined in IBM DB2 associate a user-defined function with a virtual relation.

return value. It is assumed that this return value is readily available on-demand. This assumption does not hold in a device database system for two reasons. First, functions corresponding to device ADT functions may incur high latency due to their distant location from the database server. Second, some device functions are asynchronous and thus a call to such a function may incur an arbitrary delay.

A *virtual relation* is a tabular representation of a function. A record in a virtual relation (called a virtual record) contains the input arguments and the output argument of the function with which it is associated.[3] Such relations are called virtual because they are not actually defined in the database schema, as opposed to *base relations*. In COUGAR, we use virtual relations for the internal representation of device functions.

If a device function M takes m arguments, then the schema of its associated virtual relation Attrs(VR) has $m+3$ attributes, where the first attribute corresponds to the unique identifier of a device (i.e., the identifier of an actual device ADT object), attributes 2 to $m+1$ correspond to the input arguments of M, attribute $m+2$ corresponds to the output value of M, and attribute $m+3$ is a time stamp corresponding to the point in time at which the output value is obtained.[4] We assume global time. Each time stamp thus determines a position in an ordered domain shared across all devices. As a consequence, each virtual relation could be considered as a sequence [11].

In our example, the database schema consists of one base relation (*RFSensors*) and of a virtual relation *VRFSensorsGetRainfallLevel* for the function *getRainfallLevel()*. Since this function takes no input arguments, the virtual relation has three attributes: *Sensor*, *Level*, and *TimeStamp*, i.e., the identifier of the Sensor device, the *Level* of rainfall measured, and the associated *TimeStamp*.

Note that a virtual relation has specific properties:

- A virtual relation is append-only; new records are inserted in a virtual relation when the associated device function returns a result. Records in a virtual relation are never updated or deleted.
- A virtual relation is naturally partitioned across all devices represented by the same device ADT. Each device function contributes to a portion of the virtual relation to which it is associated.

The latter observation has an interesting consequence: **a collection of devices is internally represented as a distributed database**. Virtual relations are partitioned across a set of devices. Base relations are either stored on a central database server or partitioned across devices.[5]

The Cougar System consists of a front-end server connected to a set of devices. The front-end includes a full-fledged database server. Devices include a lightweight query execution engine that is responsible for accessing virtual relations and for processing query fragments that involve these virtual relations.

Queries over a Device Database

Recall that we consider historical queries, snapshot queries, and long-running queries over a device network. Historical and snapshot queries are naturally formulated as declarative queries in SQL. Long-running queries are also formulated in SQL with little modifications to the language. We add clauses for specifying the duration of a long-running query; the choice of syntax is arbitrary.

Because of space limitation, we do not describe the complete query semantics here; the interested reader is referred to Bonnet *et al*. [12] for details. Note that long-running queries involving time windows (in particular aggregates over time windows) are best expressed using temporal extensions to the relational model [13, 14] or using a sequence model [11].

We now provide an example of a long-running query based on the flood detection application presented earlier.

Query Q: "Retrieve every 30 seconds the rainfall level if it is greater than 50 mm."

```
SELECT R.Sensor.getRainfallLevel()
FROM RFSensors R
WHERE R.Sensor.getRainfallLevel() > 50
 AND $every(30);
```

The function *$every(30)* specifies that a new record is inserted every 30 seconds into the append-only virtual relation corresponding to the function *RFSensor.getRainfallLevel()*. This record is propagated within the query execution plan chosen for the long-running query, and possibly a new answer is generated. Note that a long-running query is not evaluated by repeatedly executing the declarative query over the new records inserted in the virtual relations. (This would be a form of polling and it would introduce an arbitrary delay into the processing of device data.)

Query Processing in a Device Database System

In this section, we concentrate on a simple example to give an overview of query processing and to show the benefits of the distributed query processing approach versus a warehousing approach. Because of space limitation, we do not cover here all the issues related to query processing in a device database system. We first define new performance metrics and then discuss our example.

Performance Metrics

When processing a query, a database system first constructs an execution plan. The query optimizer is responsible for generating the execution plan that minimizes a given cost function.

The traditional performance metrics in a database system are throughput and response time. Throughput is the average number of queries processed per unit of time; it depends on the total work performed in the system to evaluate a query. Response time is the time needed by the system to produce all answer records to a query.

For long-running queries in a device database system, the traditional performance metric of query response time becomes obsolete: the query will always run for a given time interval, with varying resource usage.

We define two new metrics that correspond to the performance goals of a device database system:

- **Resource usage**: The total amount of energy consumed by the devices when executing a query. Resource usage is expressed in Joules.
- **Reaction time**: The interval between the time a function, called on a device, returns a value, and the time the corresponding answer is produced on the front-end. Reaction time is expressed in seconds.

The problem now is twofold:

- To define cost models for resource usage and reaction time.

[3] *We assume without loss of generality that a device function has exactly one return value; an extension to the general case is straightforward.*

[4] *Note that for mobile devices, we might integrate the location of the device as an additional attribute in the virtual relation.*

[5] *It is particularly interesting to partition a base relation that references a device ADT in a system where devices frequently join or leave the network; partitioning the base relation thus avoids maintaining centralized information concerning the devices currently in the system.*

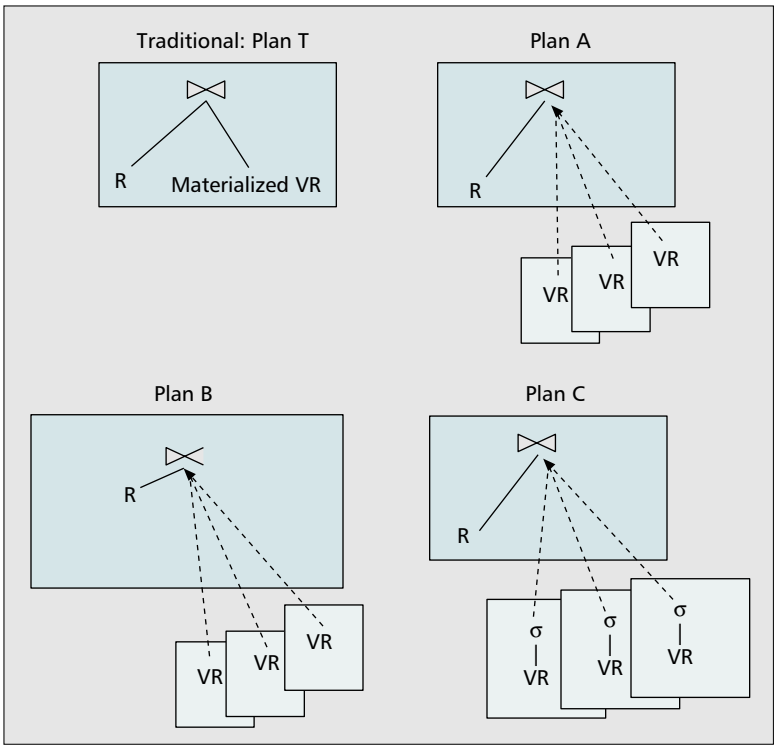

Figure 1. *Execution plans for Query Q1.*

- To obtain and maintain correct settings for the system parameters from the cost model, i.e., settings that actually reflect the status of a given device database system over time.

Example

Our goal in this section is not to cover all issues related to query processing in a device database system, but rather to illustrate how existing distributed database techniques can be applied in this new context [15, 16]. We discuss the characteristics of device database systems with respect to existing distributed database systems and use an analytical model to illustrate the benefits of our approach.

Query Q1: "Retrieve every 30 seconds the rainfall level if it is greater than 50 mm."

```
SELECT VR.value
FROM VRFSensorsGetRainfallLevel VR, RFSensors R
WHERE VR.Sensor = R.Sensor AND VR.value > 50
      AND $every(30);
```

We use as our example the query Q1, which is the result of rewriting query Q using the virtual relation *VRFSensorsGetRainfallLevel*. This query could be used to monitor the evolution of rainfall in flooded areas. We consider a system with 200 devices; the cardinality of relation *R* is therefore 200 records. Query Q1 is run as a long-running query with a duration of four hours. The rainfall level is measured every 30 seconds; as a result, up to 480 virtual records are inserted into each partition of the virtual relation.

Distributed Query Execution Plans — SQL queries usually have a large space of possible execution plans. These are obtained by considering various shapes for the tree of relational operators, by permuting the position of relational operators in this tree, by choosing various implementations for a relational operator (in particular, each database system implements a set of join methods, e.g., nested loop, sort-merge, hash-join, semi-join), and by permuting the relative position of sub-trees [17]. In a distributed context, the execution plan reflects the distributed nature of the database: it is composed of query fragments, i.e., sub-trees of relational operators, assigned to execution sites. Three more dimensions are thus added to the space of possible execution plans: What are the candidate execution sites? How are query fragments associated to execution sites? What is the strategy for transferring data from one site to another?

Figure 1 presents four execution plans for Q1; each plan is a tree of relational operators that manipulate base and virtual relations. Plan T represents the execution plan that would be generated for Query Q1 in a traditional system such as ALERT. Data extracted from the devices are materialized in the relation VR that is located on the front-end (represented as a darker shaded rectangle). The execution plan is a simple tree composed of one join operator between relation R and relation VR (using joining condition *R.Sensor = VR.Sensor AND VR.value > 50)*. This join is executed on the front end.

The other execution plans illustrate the use of distributed database techniques in a device database system. Plan A is also a simple tree where R is joined on the front end with relation VR partitioned across a set of devices (represented as lighter shaded rectangles). This execution plan is evaluated as follows. The front end asks each device to measure rainfall level and to transfer the resulting virtual records back to the front end. (Virtual records are produced once on each site for a snapshot query, and repetitively for a long-running query). Each virtual record arriving on the front end is then joined with relation R.

Intuitively, this execution plan is not optimal: all devices with rainfall sensors transmit data to the front end while the query only concerns the sensors that measure a rainfall level greater than 50. An alternative execution plan pushes the join to the devices, thus trading increased processing on devices for reduced network traffic. Instead of pushing the join between R and VR to each device, Plan B defines a semi-join between relation R and the partitions of the virtual relation VR located on the devices [16]. The semi-join projects out the joining attribute from relation R (here the device id *Sensor*) and sends the resulting relation to all devices; a semi-join avoids transferring the complete relation R to all devices. On the devices, whenever the rainfall level is measured, a virtual record is generated and it is joined with the portion of relation R sent by the front end (using joining condition *R.Sensor = VR.Sensor and VR.value > 50*). If the joining condition is verified, then the virtual record is sent back to the front end, where it is joined with complete records from relation R (not only the joining attribute). Only the sensors whose rainfall level is greater than 50 send data back to the front end.

A third execution plan only pushes the selection (*VR.value > 50*) onto the devices; only records that verify this condition are sent back to the front end, where they are joined with relation R. Plan C represents this execution plan. Compared to Plan B, there is no subset of relation R transmitted to the devices. We compare the resource usage of these three execution plans in the next section.

Analytical Model — We use a simple analytical model to compare the costs of the three execution plans identified in the previous section. We assume a multi-cluster, single-hop WINS network [7]. There are 20 clusters each containing 10 devices. We consider the total energy consumed per sensor node as the linear combination of CPU costs, the cost of a memory access, the cost of sending a message, and the cost of sending *N* bytes on the network:

Cost in Joules = Wcpu * CPU + Wram * RAM
+ Wsend * NbMsgs + Wbdw * SizeMsgs

The weight factors are used to transfer all components of the cost into Joules. Table 1 lists the weight factors we used for our experiments. The factors were obtained by W. Kaiser and G. Pottie, through measurements in a WINS network composed of sensor nodes from Sensoria Corp. [7]. The energy needed by the processor to operate dominates the energy needed by the RAM, so we set Wram = 0. The cost per record of a join or a selection is NbInstPerComp instructions. We do not model the cost of invoking the device function. The cost per message is due to synchronization between the sending and receiving nodes. We consider that nodes are 30 meters from each other. In this case the cost of sending 1000 bytes is 0.23J. (Note that the capacity of a battery on a WINS sensor node is 3.5E4 Joules.) We further assume that the size of each virtual record is 50 bytes.

Wcpu	0.000001 J/instruction
Wram	0
Wsend	0.059 J/msg
Wbdw	0.23 J/Kbytes
NbInstPerComp	5000

Table 1. *Parameters and settings for modeling resource usage.*

We study resource usage on sensor nodes directly involved in the query (i.e., the nodes on which a partition of the virtual relation is located); we do not consider resource usage on the nodes that are traversed for communication purposes. Each sensor node satisfies the condition in query Q1 (*Vr.value > 50*) with a certain probability. We trace the resource usage in the two extreme cases, i.e., for sensor nodes that are always located outside a flood area and whose rainfall level is thus never greater than 50, and for sensor nodes that are always located inside a flood area.

Figure 2 traces the resource usage expressed in Joules as a function of time (given that the rainfall level is measured every 30 seconds) for nodes always located outside a flood area. With Plan A, data is sent back to the front end whenever it is generated. With Plan B and, respectively, Plan C, a join and, respectively, a selection, are pushed to the device. As a result, the condition on the rainfall level is checked on the devices and none of the devices located outside a flood area sends data back to the front end. Plan B pays the initial cost of transferring a fragment of relation R to the devices. This initial cost is amortized (compared to Plan A) during the lifespan of a long-running query.

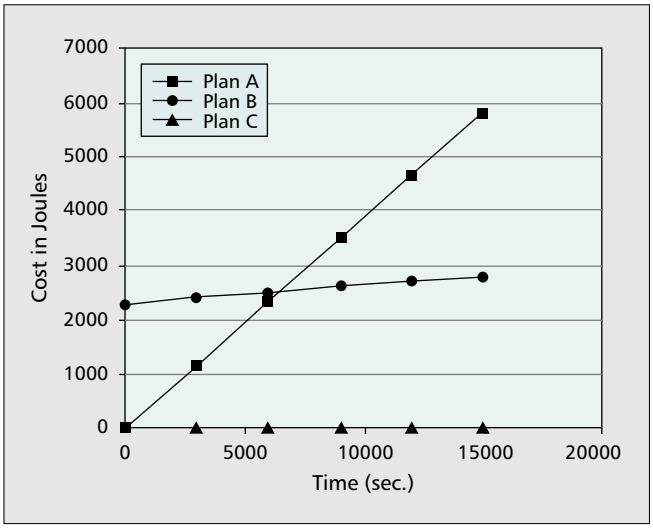

Figure 2. *Resource usage for sensors located outside a flood area.*

Figure 3 traces the resource usage expressed in Joules as a function of time (given that the rainfall level is measured every 30 seconds) for nodes always located inside a flood area. With all plans, data is always sent back to the front end. The initial cost of Plan B is here never amortized. Plan C and Plan A have almost similar curves; this illustrates that the cost of performing a selection is low compared to the cost of sending data.

In this example, pushing a selection as in Plan C is the optimal choice. This is intuitive since the query filters out uninteresting events generated on the devices. Pushing the selection allows the device database system to trade efficiently increased processing on the devices for reduced communication.

Conclusions

In the near future, devices with processing and communication capabilities will be deployed in the physical world, providing a powerful computing platform. The first generation of the Cornell Cougar systems demonstrates that the application of database technology to this new computing platform shows much promise for providing flexible and scalable access to large collections of devices. Our work has introduced a set of research problems, and we now provide a brief overview of some of the questions that our ongoing research is addressing:

Meta-data management: Current distributed database optimizers assume global knowledge, i.e., the optimizer has access to exact meta-information about the complete system. In a device database system, we cannot assume that a single site maintains global knowledge about the system because of the large scale and dynamic nature of a device network, and because it would incur a significant administration overhead. How can we maintain meta-data in a decentralized way and how can we utilize this information to devise good query plans?

Query processing: Query processing should take advantage of the computing capabilities at the devices in order to minimize the total amount of resources consumed in the device network while minimizing reaction time. In addition, conditions in a device network change over time. Devices fail, move, or disconnect, the network topology may evolve, and batteries are used and recharged. Thus query plans must adapt dynamically

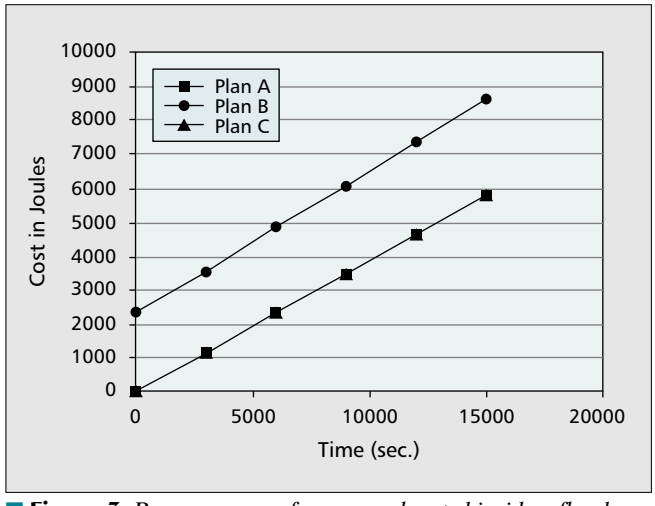

Figure 3. *Resource usage for sensors located inside a flood area.*

to changing network conditions and they must show a certain degree of robustness against device failures. In addition, for long-running queries the conditions in the device network might change significantly while the query runs.

Acknowledgments

We thank Stephane Bressan and Tobias Mayr, who helped debug earlier versions of this article, as well as the reviewers for helpful comments. This article benefited from interactions with the SensIT community. In particular, Bill Kaiser provided valuable information concerning the Sensoria WINS network. This work is sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F-30602-99-0528, by the National Science Foundation under Grant No. EIA 97-03470, by NSF Grant IIS-9812020, and by a grant from Microsoft Research to Philippe Bonnet.

References

[1] D. Estrin, R. Govindan, and J. Heidemann (Eds.), "Embedding the Internet," *Communications of the ACM*, vol. 43, no. 5. May 2000.
[2] DARPA: SenseIT Project, http://www.darpa.mil/ito/research/sensit/background.html
[3] D. Estrin *et al.*, "Next Century Challenges: Scalable Coordination in Sensor Networks," *Mobicom '99*, Seattle, Washington, pp. 263–70.
[4] T. Imielinski and S. Goel, "DataSpace: Querying and Monitoring Deeply Networked Collections in Physical Space," MobiDE 1999, pp. 44–51.
[5] P. Bonnet and P. Seshadri, "Device Database Systems, poster paper," *Proc. Int'l. Conf. Data Engineering ICDE'99*, San Diego, CA, Mar. 2000.
[6] *Intel Computing Continuum Conf.*, http://www.intel.com/intel/cccon/
[7] J. M. Kahn, R. H. Katz, and K. S. J. Pister, "Mobile Networking for Smart Dust," *ACM/IEEE Intl. Conf. Mobile Computing and Networking (MobiCom '99)*, Seattle, WA, Aug. 17–19, 1999
[8] G. Pottie and W. Kaiser, "Wireless Integrated Network Sensors (WINS): Principles and Approach," *Communications of the ACM*, vol. 43, no. 5, May 2000.
[9] P. Seshadri, "Enhanced Abstract Data Types in Object-Relational Databases," *VLDB Journal*, vol. 7, no. 3, 1998, pp. 130–40.
[10] U. Schreierf *et al.*, "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS," *VLDB*, 1991, pp. 469–78.
[11] P. Seshadri, M. Livny, and R. Ramakrishnan, "SEQ: A Model for Sequence Databases," *ICDE* 1995.
[12] P. Bonnet *et al.*, "Query Processing in a Device Database System," Cornell Technical Report TR99-1775, Oct. 1999.
[13] A. Tansel *et al.*, *Temporal Database: Theory, Design and Implementation*, Benjamin/Cummings, 1993.
[14] A.Dekhtyar, R. Ross, and V. S. Subrahmanian, "Probabilistic Temporal Databases: Algebra," Jan. 1999, University of Maryland technical report CS-TR-3987, submitted to *ACM Trans. Database Systems*.
[15] L. F. Mackert and G. M.Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries," *Proc. Int'l. VLDB Conf.*, pp. 149–59, Kyoto, Japan, Aug. 1986, Management Systems. ICOD 1980: pp. 204–15.
[16] C. T. Yu, "Distributed Database Query Processing," *Query Processing in Database Systems*, 1985: pp. 48–61.
[17] R. Ramakrishnan and J. Gehrke, *Database Management Systems, Second Edition*, McGraw Hill, 1999.

Biographies

PHILIPPE BONNET (bonnet@cs.cornell.edu) received a Ph.D. from the Universite de Savoie in 1999. He is currently a research associate in the Department of Computer Science at Cornell University. His current research interests involve device database systems, database tuning, and next-generation database systems. He is a member of the ACM and the IEEE Computer Society.

JOHANNES GEHRKE (johannes@cs.cornell.edu) received his Ph.D. degree from the University of Wisconsin in 1999. He is currently an assistant professor in the Department of Computer Science at Cornell University. His research is in the areas of data mining and database systems. He is the recipient of an IBM Faculty Award and he serves on the editorial board of *Knowledge and Information Systems*. He is the co-author of the textbook *Database Management Systems (Second Edition)* published by McGraw Hill in 1999. He is a member of the ACM and the IEEE Computer Society.

PRAVEEN SESHADRI (praveen@cs.cornell.edu) received a Ph.D. from the University of Wisconsin-Madison in 1996. He is an assistant professor in the Department of Computer Science at Cornell University, currently on leave at Microsoft. His research is in the area of next-generation database systems and data management for personal devices. He received an IBM Faculty Award and an NSF Career Award. He is a member of the ACM and the IEEE Computer Society.