Query Processing in a Device Database System

Philippe Bonnet, Johannes Gehrke, Tobias Mayr and Praveen Seshadri[1]

Cornell University

Upson Hall 4122

Ithaca, NY 14853

{bonnet,johannes,mayr,praveen}@cs.cornell.edu

In the next decade, networks of devices will be widely deployed for measurement, detection and surveillance applications. Millions of sensors and small-scale mobile devices will integrate processors, memory and communication capabilities. Large collections of devices need to be controlled and accessed in an ad-hoc manner. This paper shows that database technology can be adapted to meet the challenges of this new computing environment. In our new concept of a device database system, individual devices are modeled as database objects, which allows us to access collections of devices with declarative queries. We present a semantics for queries over device database systems and novel query processing techniques for evaluating such queries. We describe our implementation of these techniques in the Cornell COUGAR system and present a experimental evaluation that illustrates the performance characteristics of query processing in a device database system.

1 Introduction

1.1 Motivation

The nature of computing is changing dramatically with the widespread deployment of sensors, actuators and mobile devices that integrate processors, memory and communication capabilities. Computing power and memory will be integrated 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

[1] Praveen Seshadri is currently on leave at Microsoft: 3/1102 Microsoft, One Microsoft Way, Redmond, WA. pravse@microsoft.com.

wallet or key ring. Emerging networking techniques ensure that devices are interconnected, and provide gateways between device networks and local- or even wide-area networks.

Device networks are primarily used for measurement, detection or surveillance. Application scenarios described in the literature involve monitoring items in a factory warehouse, gathering information in a disaster area, detecting threats in a battlefield or organizing vehicle traffic in a large city [S99, E+99]. We detail below the warehouse example that we use throughout the paper.

Example 1*: Consider a warehouse that stores a large number of items. Each item has a smart tag attached to it, i.e., a form of smartcard that stores tracking information. Contactless terminal readers placed on the storage racks provide access to these smart tags. Temperature sensors and alarms are installed on the walls and on the ceilings. A wireless network interconnects the sensors and the terminal readers and provides a gateway to a local area network. The warehouse supervisor uses the device network to manage the inventory and to monitor activities as follows. She typically accesses the device network in an ad-hoc manner: Where are the items originating in Florida? Whenever a new item from Florida is added to the warehouse, notify me about its location. What are the items stored in a particular area of the warehouse? How many items of a certain type are stored? How long have items of a certain type been stored? What is the temperature in the back of the warehouse? Raise an alarm if temperature sensors detect an abnormal temperature.*

Detection, surveillance and measurement applications access large collections of devices. A large warehouse may contain a hundred thousand items and thousands of sensors. Organizing the vehicle traffic in a large city may involve millions of devices. Further, device networks with mobile devices have a dynamic topology, making it difficult to predict the location of individual devices. Clearly, procedural programming mechanisms over device networks would be too complex and limited in flexibility and scalability. Monitoring and control of a device network are best described in a declarative manner, that is, queries should be formulated independent of the physical structure and the organization of the device network. Database systems have proven that they can provide declarative yet efficient access to large volumes of data. We show in this paper that database technology can be adapted to provide declarative, efficient, and scalable access to device networks. Our solutions involve a blend of techniques from many areas including object-relational databases [S98] and continuous queries [E+99, T+92, S+91].

Once one has made the observation that database technology can be applied to access device networks, the following questions arise:

1. Can we represent devices in a uniform way? All devices provide some interface to a computer system – however there is no shared standard way to access them[2]; different categories of devices are accessed with different protocols, they represent data in different ways.
2. Can declarative queries/commands capture the various patterns of access to a device network? In our example, the device network is used (a) to obtain information at a given point in time (how many items of a certain type are stored in the warehouse?), (b) to continuously gather information (whenever an item from Florida is added to the warehouse then give me its location in the warehouse), (c) obtain data asynchronously (tell me the location of sensors that detect an abnormal event) and (d) to trigger actions (raise an alarm if two temperature sensors implanted in the same area detect abnormal temperatures). What are suitable semantics for such queries?
3. Can these queries be processed using a traditional database system? Can they be executed using traditional relational operators? Can they be optimized using a traditional cost model?
4. Can a database system cope with the large number of devices, and the volume of information they may produce?

We believe that all these questions have positive answers. We show how to model devices using abstract data types, a feature readily available in today's object-relational and object-oriented database systems. We then describe how we can query a device network declaratively and how to process such queries efficiently. It is our vision that the idea of applying database technology to the domain of device networks has opened the door to a new area of research problems, and we see this paper as an important first step. Our view of the problem introduces a plethora of other research questions (e.g., what level of transactional support is needed, how to deal with the limited resources on the devices) that are topics for future research.

1.2 Summary of Contributions

Our paper makes the following contributions:

- We show that we can model devices as abstract data types (ADT) in an object-relational or object-oriented data model, and we argue that this abstraction is powerful enough for the purpose of accessing the different kinds of devices. In particular, a device ADT may support synchronous or asynchronous methods.

- We introduce well-defined semantics of long-running queries over device ADT objects.
- We show that conventional query execution techniques are not suitable in a device network. The usual tuple-at-a-time execution strategy does not work for asynchronous methods, because the system may block for a long time executing a method on one object while asynchronous methods on other objects could produce data. We present set-oriented execution strategies adapted to the distributed context of a device database system.
- We describe the implementation of the Cornell COUGAR device database system based on PREDATOR.
- We study the scalability of the COUGAR device database system and we illustrate the performance characteristics of the different query execution strategies.

The outline of this paper is as follows. Section 2 details our new concept of a device database system. We describe a data model for representing devices and we precise the semantics of declarative queries over collections of devices. Section 3 focuses on the processing of these queries. We first introduce novel query execution techniques and then describe a cost model adapted for this new context. Section 4 presents our implementation of the COUGAR device database system. We analyze the performances of this system in Section 5. Section 6 discusses related work.

2 A Database Abstraction for Device Networks

In this paper, we study the problem of querying a device network. A query over a device network involves data from the device network (obtained as the result of function invocations on collections of devices) with possible external data from the accessing database. Representative example queries are given in the introduction.

Note that, at this point, we do not have the foundations for a formal problem definition. We consider one of the contributions of this paper the embedding of the problem into a sound framework based on ideas from the database community.

2.1 Device Networks

A physical object with computing and communication capabilities is called a *device*. There are smart devices that embed computing and communication capabilities (e.g., smartcards [GS98], JavaRings [JR98], and I-button temperature sensors [IW98]) and less sophisticated devices that are composed of a

[2] This is a goal of Jini [E99] to standardize the interfaces and the protocols for accessing devices.

physical object connected to an external computer (e.g., a seismic or infrared sensor connected to a Windows CE palm-sized PC or a door actuator connected to a desktop computer).

Each device is conceptually a mini-server. It supports a set of functions and allows a certain amount of processing to be done directly at the device. 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 in a timely fashion, at any point in time, whenever. *Asynchronous functions* return results at arbitrary points in time. Asynchronous methods are inherent to the nature of some devices. Smartcards, for instance, are only available when they are connected to a terminal reader. As a consequence, every method invocation on a smartcard is asynchronous: The method will return a result only when a smartcard actually connects. Event detection using sensors can also be asynchronous. For example, assume that we program a temperature sensor to return a notification if a dangerously high temperature is detected. We do not know when such a temperature will be detected, thus the detection function is asynchronous. Note that a device might provide both synchronous and asynchronous functions. For example, a temperature sensor might provide the functionality of obtaining the current temperature plus the functionality of programming the sensor as above.

A network of devices (or short, device network) consists of several, very likely a large number of devices that are connected through networks of various forms, ranging from a classical wired local area network to ad-hoc wireless networks. The routing mechanisms in such a network can range from conventional routing to newer protocols that are based on publish/subscribe mechanisms [E+99]. In such a protocol, the recipient of a message is not described by an individual address (e.g., an IP address) but by meta-information that describe its properties (e.g., the location of the device, or its type). From the view of a user of such a device network, it is important that we can assume that the device network exports an abstract interface that allows to route messages to individual devices and groups of devices.

2.2 A Uniform Way of Modeling Devices

In order to formalize the problem of querying a device network, we propose to model devices as objects in a database schema.

2.2.1 Devices as Abstract Data Types

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. The critical feature of an ADT that makes it suitable for representing devices is *encapsulation*.

Note that there are natural parallels between devices and ADTs. Both ADTs and devices provide controlled access to encapsulated data and functionality through a well-defined interface. We build upon this observation by modeling each type of device in the network as an ADT in the database – an actual ADT object in the database corresponds then to a physical device in the real world. The public interface of the ADT corresponds to the functionality supported by the device. Methods on the device are executed by sending requests to the device, which evaluates the methods and answers with return values. Differences between the different device types are reflected by differences between the abstract data types that represent them. Note that due to the flexibility in the design of the interface of the ADT, we can incorporate a wide range of security models for devices that provide certain security and integrity levels for the data they contain.

Let us model the database schema corresponding to the warehouse example from the introduction. We consider a simple schema that consists of three relations

- *Items(ArrivalDate, Location, Tag)*
- *Sensor(Location, TemperatureSensor)*
- *Control(Location, Alarm)*

These relations are stored on the central database server. When a new item is received, a new tuple is inserted in the *Items* relation. Each tuple contains the date of arrival *ArrivalDate* and a *Tag*. The *Tag* attribute of this tuple is a device ADT value, that contains a unique identifier for the represented tag. The actual *Tag* data is located on the smart tag attached to the item (and is obtained through the contactless terminal associated to the storage rack where the item is stored). The *Tag* device ADT provides, for example, methods encapsulating tracking information, e.g. *getType*() and *getOrigin*() and methods for locating the items, e.g. *getLocation*().

The *Sensor* relation contains a tuple for every temperature sensor in the warehouse. Similarly, the *Control* relation contains a tuple for every alarm. These relations associate a location attribute with a device ADT. The *TemperatureSensor* ADT provides a synchronous method *getTemperature*() that returns the current temperature and an asynchronous method *detectGreaterThan*(*int Temperature*) that is used to detect abnormal temperatures, i.e., temperatures greater than the given argument value. The *Alarm* ADT supports the method *raise*() which is used to trigger the alarm.

Here is the formulation in SQL of three queries presented in the introduction:

- Query 1: Where are the items that originate from Florida?

```
SELECT  I.Tag.TagID, I.Tag.location()
FROM    Items I
```

WHERE I.Tag.getOrigin() = "Florida";

- Query 2: What are the dates of arrival for items of a certain type?

SELECT I.ArrivalDate

FROM Items I

WHERE I.Tag.getType() = "Gadget";

- Query 3: Raise an alarm whenever a temperature sensor detects an abnormal temperature in warehouse hall 5.

SELECT C.Alarm.raise()

FROM Control C, Sensor S

WHERE S.location = C.location

AND S.location = "Hall 5"

AND S.TemperatureSensor.detectGreaterThan(100);

Although these SQL queries are quite simple, their semantic is not yet perfectly clear. Are the methods on devices applied once or are they applied repeatedly? For instance, in Query 2, we want to access the current state of the device network, whereas Query 3 is a long running query that should, when necessary, raise several alarms over a period of time.

We precise the semantics of these queries in the next section.

2.3 Query Semantics

Given that we query the device network using SQL, why is the semantics of such queries not simply straightforward and seems possibly ambiguous? First, users will want to ask long-running queries that persist over a specified time-interval. Since methods are applied repeatedly during this time interval, we have to specify what the result of a long-running query contains. Second, queries combine data that is the result of method invocations on devices with external data that come from a traditional database. If the method is asynchronous, the result of the method might be available only after an arbitrary amount of time. What does it mean to ask queries involving such methods? We address these two problems in this section. In this paper we assume for simplicity that the external data is static, and we refer to relations that contain external data as base relations. Our query semantics can be extended to the case that the external data is not static, but we do not address that case in this paper, since we do not discuss the related issue of concurrency control.

Before we address the two problems from the previous paragraph, let us introduce a modeling assumption that we make throughout the paper. (The following discussion assumes without loss of generality that a method on a device has exactly one return value; an extension to the general case is straightforward.) In a query that involves method invocation M on an ADT X, we model the method as a *virtual relation VR.* If M takes m arguments, then the schema of VR has m+2 attributes, where the first attribute corresponds to a unique identifier of an ADT object instance, attributes 2 to m+1 correspond to the input arguments of M, and attribute m+2 corresponds to the output value of M. We call such a relation virtual, since it does not exist materialized in the database schema.

As an example, consider Query 2. The method getType() of the ADT Tag is translated into a virtual relation, let us call it VTTag, with schema (TagID, type). Using VTTag, Query 2 is rewritten as follows:

SELECT I.ArrivalDate

FROM Items I, VTTag V

WHERE I.TagID = V.TagID and V.type = "Gadget";

Let us now address the first of the two problems, the issue of long-running queries. We incorporate the notion of time inherent in such queries by assuming that we are given a totally ordered domain of discrete time stamps T={t0, t1, …} with t0<t1<… To specify the semantics of queries precisely, we replace each relation R with its timestamped relation RT as follows: If R is a relation with schema (a1, …, an), then RT has schema (a1, …, an, t). For a base relation R, the corresponding timestamped relation RT is RT={(a1, …, an, t)|(a1, …, an) \in R and t \in T}. (Note that this definition relies on our simplifying assumption that base relations do not change while a query is running.)

To address the second problem, we need to extend the definition of timestamped relations to virtual relations. For a virtual relation VR, there are two cases. If the virtual relation corresponds to a synchronous method M whose result is readily available on demand, then the timestamped virtual relation VTR contains for each object O with identifier o and timestamp t in T, a record (o, a1, …, am, y, t) such that method M returns value y for arguments (a1, …, am) at time t. If the virtual relation corresponds to an asynchronous method M that returns a result only in a subset S subset T, then the timestamped virtual relation VTR contains for each object O with identifier o and timestamp t in S a record (o, a1, …, am, y, t) such that method M returns value y for arguments (a1, …, am) at time t. Note that for asynchronous methods in general S ≠T, since asynchronous methods return output values at arbitrary points in time; as a result, there is a tuple in the virtual relation only for some values in the time domain. In contrast, base relations have

identical contents for each time stamp, and virtual relations that correspond to synchronous methods have a tuple for each time stamp, but the contents might be different for different timestamps.

Having introduced the notion of a timestamped relation, we can now formulate queries precisely by replacing relations with their corresponding timestamped relations and introducing suitable join conditions on the time stamps. The semantics of the queries involving timestamped relations can be defined as in regular SQL; we omit the details here and explain by example. Consider Query 2, which is formulated to capture the current state of the device network; it can now be formulated precisely as follows:

```
SELECT   I.ArrivalDate
FROM     Items I, VTTag T
WHERE    I.TagID = T.TagID  and T.type = "TYPE1"
         AND I.ts = T.ts
         AND I.ts = $currentTimeStamp; // captures the current state
```

As another example, Query 3 is formulated as a long-running query and can be expressed as below. The two method invocations have been reformulated into two virtual tables that result in two joins. Note that this query only produces an output tuple for those timestamps where the asynchronous method *detectGreaterThan(100)* returns a value, i.e., the alarm is raised exactly at those times when the temperature is greater than 100. This is reflected in the virtual relation, which only contains tuples for the timestamps at which an alarm is raised:

```
SELECT   A..raise
FROM     Control C, Sensor S, VTTemperaturSensor TS, VTAlarm A
WHERE    S.location = C.location
         and C.AlarmID =A.AlarmID        // first method modeled as join
         and S.sensorID =TS.sensorID     // second method modeled as join
         and C.ts = S.ts and S.ts = TS.ts and TS.ts = A.ts
         and $currentTimeStamp <= C.ts   // long-running query semantics
         and C.ts <= $currentTimeStamp+3600;    // duration of the query
```

3 Query processing

In the previous section, we described the semantics of device databases where devices are represented as ADT objects and SQL queries are used to access them. Now the problem is to define an architecture for a device database system which allows the processing of queries over device databases with the given semantic.

In this paper, we consider that a device database system is composed of a database server connected to a set of devices using a device networking service.[3] Figure 1 represents the basic components of a device database system.

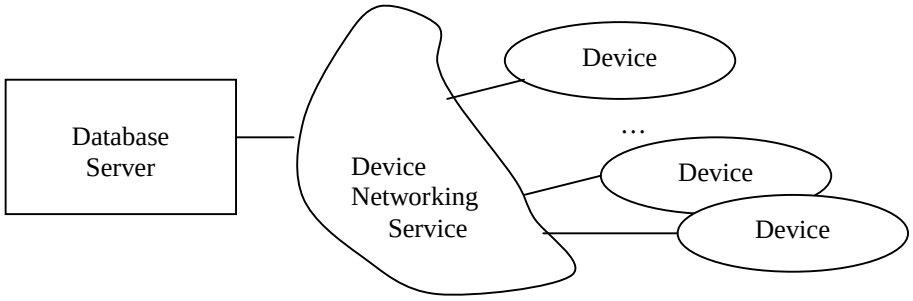


Figure 1

Based on this framework, we need to devise query evaluation techniques that work efficiently on device databases. Can the database server rely on traditional query execution techniques? Should the optimizer be extended? We discuss these issues in the next section.

3.1 Query Execution

In object-relational database systems, the return value of a method is treated at the same level as variables or constants in a condition or a projection list. Thus the physical operators assume that the return value of a method invocation is readily available at any time. This assumption does not hold for asynchronous methods.

To illustrate the problem, let us consider a specific physical plan used to evaluate Query 3 (introduced in section 2.1). This query raises an appropriate alarm whenever a given temperature sensor detects a temperature greater than 100F. One possible execution plan would be the following. A join is involved

[3] We discuss the connection between the database server and the devices in the description of the COUGAR implementation (Section 4).

between relations *Control* and *Sensor*. Let us assume that sensor is chosen as the outer relation and that a tuple nested loop is used to perform this join (these are critical choices, we will come back on this when discussing query optimization in section 3.2). For each temperature sensor in the Sensor relation, the method *detectGreaterThan(100)* is applied. When the output of this method is obtained, then the nested loop join looks for a tuple in *Control* satisfying the joining condition (*S.location = C.location*). If a matching tuple is found, then the method *raise()* is applied on the Alarm ADT value and an answer is produced.

There is a serious flaw in this execution! The method *detectGreaterThan(100)* is asynchronous, i.e. it returns its output after an arbitrary amount of time. While the system is trying to detect an abnormal temperature on the first sensor obtained from the *Sensor* relation, the other temperature sensors have not been yet been contacted. It may very well be that some temperature sensors could have detected temperatures greather than 100, while the system is blocked waiting for the output of one particular method. Figure 2 illustrates this problem by showing the timeline of this query execution. The system blocks and waits for temperature sensor TS# 1 to provide an answer. In the meantime, TS# 2 and TS#3 would have detected abnormal temperatures but have not yet been contacted. (The dashed arrows represent the missed asynchronous method output). Moreover, with the default long-running query semantics, the method detectGreaterThan must be applied repeatedly on all devices. The execution described above only accounts for one method execution on each device.

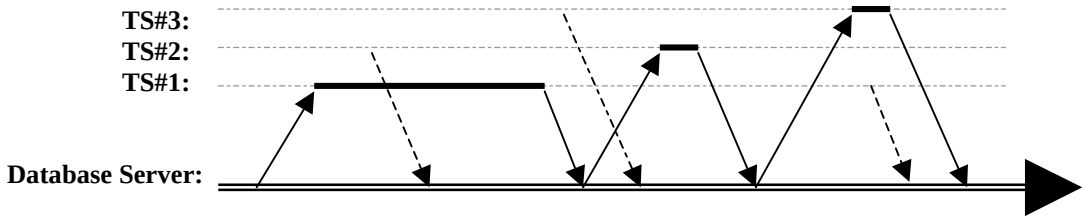


Figure 2

This example shows that the traditional *tuple-at-a-time* execution of ADT methods is inadequate for the execution of asynchronous methods on device ADTs, or for applying methods repeatedly with the long-running query semantics. Intuitively, we would like all asynchronous method executions to overlap, i.e. all temperature sensors should be asked to detect abnormal temperatures simultaneously and repeatedly. Figure 3 shows the timeline for a query execution where the execution of asynchronous methods on all devices is overlapped.

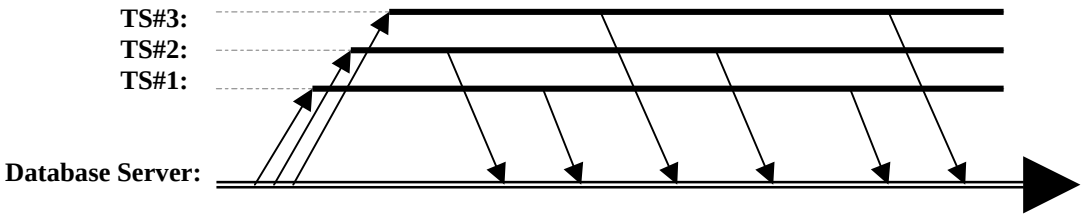


Figure 3

We introduce a new operator in the physical algebra to represent the execution of an asynchronous method on a device ADT. Indeed, representing methods in the condition list or in the projection list of relational operators implies a tuple-at-a-time evaluation. Representing a method invocation as an operator in the physical algebra naturally implies set-at-a-time evaluation.[4]

We call the new operator *virtual join*, because conceptually, it joins a base relation with the virtual relation representing a method. The base relation provides the input arguments for the method execution. In our example the virtual join takes the *Sensor* relation as input. Note that the virtual relation that originates on the devices corresponds to the virtual relation used to define the query's semantics in section 2.2.

We define four strategies for the execution of virtual joins. We first illustrate these strategies using our example query. In the execution plan, the virtual join operator is applied on the *Sensor* relation before its results are joined with *Control* on the location column:

- **Sequential strategy**: This strategy corresponds to the tuple-at-a-time execution described above. Conceptually, this virtual join is a tuple nested loop join that uses the base table *Sensor* as the outer table, while the inner, virtual table is probed in the order in which the device ids appear in *Sensor*. As we have seen, this strategy fails.
- **Parallel strategy**: For each temperature sensor in the *Sensor* relation, the parallel virtual join operator sends a message asking one particular temperature sensor, identified by its device id, to execute the asynchronous method detectGreaterThan(100). Each time one of the addressed devices detects heat and accordingly executes the method, it sends a message back to the operator on the server, which joins the result with the corresponding tuple from *Sensor*. Conceptually, the virtual join is executed as a semi-join between the server and the devices: The join column of *Sensor* is shipped to the devices, which return only the relevant tuples of the virtual table.

[4] It appears that representing method invocation as relational operators offers interesting opportunities in other environments as well[MS99].

- **Aggressive Strategy**: The aggressive virtual join operator sends a message to all temperature sensors (without considering which ones are actually referenced in *Sensor)* asking them to apply the method *detectGreaterThan(100)* repeatedly. All temperature sensors execute the asynchronous method whenever possible. When a temperature sensor obtains a result, it is returned to the database server, which joins the result with the corresponding *Sensor* tuple. Conceptually, this strategy corresponds to a nested loop join on the server with the virtual table as outer table.
- **Conservative Strategy**: The server sends a message to all sensor devices corresponding to the device type on which the method is to be applied (not only to the ones in *Sensor*), requesting them to send back their device identifier. When a device is available to apply a method, it sends its id back to the server, which looks for it in the *Sensor* relation. If found, the server contacts this device asking it to apply the function and to return the answer when available. A returned answer is joined with the corresponding tuple in *Sensor*. Conceptually, this strategy executes a semi-join between the devices and the server: The devices send their join column to the server which requests back only the relevant results. In contrast to a fully distributed semi-join, the final join happens on the server because the devices are not able to process it.

We can thus categorize the virtual join strategies in terms of join method and join order. The sequential strategy corresponds to a nested loop with the base relation as the outer. The parallel strategy corresponds to a semi-join with the base relation as the outer. The aggressive strategy corresponds to a nested loop with the virtual relation as the outer. And the conservative strategy corresponds to a semi-join with the base relation as the outer. Figure 4 summarizes this classification.

The aggressive strategy is only applicable if the asynchronous method does not take input arguments (other than the device id). The conservative strategy is particularly useful in the context of devices where a method is asynchronous because the device is available only intermittently. This is the case of smartcards. It thus makes sense to check what devices are involved in the query before actually applying a method.

We compare the performance of these virtual join strategies in Section 5.

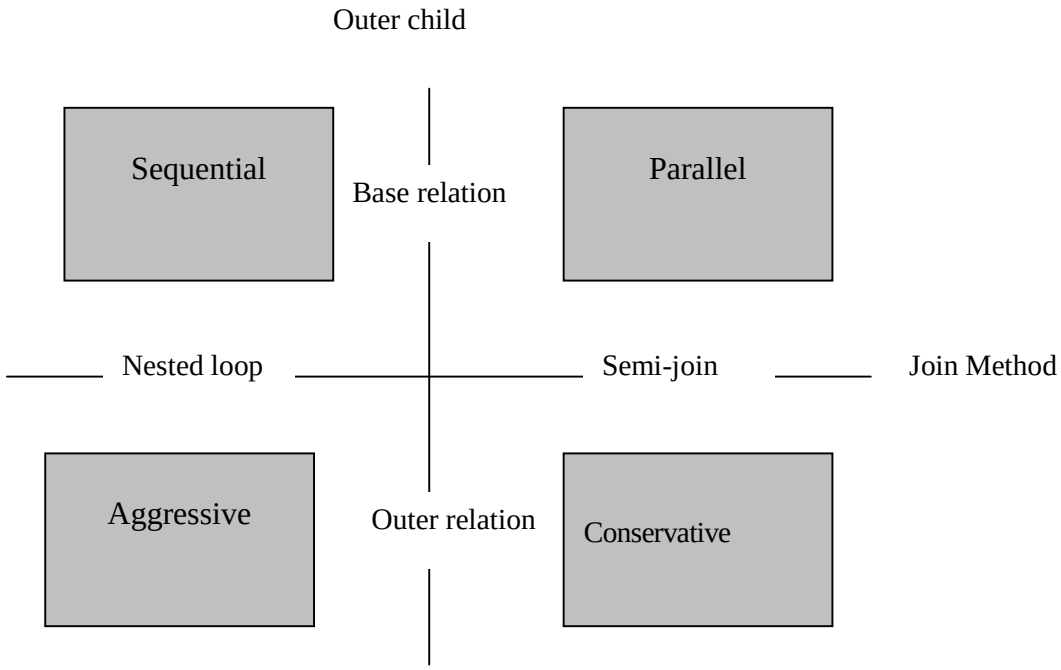


Figure 4

3.2 Query Optimization

A classical query optimizer can be modified in a straightforward way to generate query execution plans that include virtual join operators. A virtual join operator is generated for each method on a device ADT. A virtual join operator is generated for each method on a device ADT. Classical optimizers place such operators based on their per-tuple cost and their selectivity. A virtual join operator's selectivity is given by estimates on the frequency of method invocation and its cost can be estimated as the incremental cost to react to a single answer returned by a device.

Virtual join operators that apply methods repeatedly are able to continuously produce tuples. With the iterator query execution model, these tuples should be pipelined to the root of the execution plan. For instance, a nested loop join can be used with one virtual join operator that applies its method repeatedly

as its outer child and a scan on a base relation as its inner child (this is the case in our example query in section 3.1).

The optimizer must choose between the different implementations of a virtual join operator. The cost model can be extended to integrate these operators. With the default semantics of long running queries, the traditional objective function of query response or execution time becomes obsolete: The query will always run for the whole time interval, with varying resource usage. The latter is important and can be reflected as utilization cost – the amount of resource that is used by a long running query during its execution.

A meaningful metric for long running queries that reflects latencies, esp. of device network communication, is the *reaction time*: the time elapsed between (a) an asynchronous method becomes applicable and (b) the corresponding result is produced. New metrics will have to be introduced to reflect the resource usage on the devices (e.g. the power consumption) and the traffic in a device network. Defining cost models adapted to these metrics is a topic for future work.

4 Implementation in Cougar

We have extended the Cornell PREDATOR DBMS to prototype the COUGAR device database system. PREDATOR is an object-relational database server. Details of the implementation of PREDATOR are provided in [P99]. In addition to standard SQL, it supports extensions defining new data types (ADTs) and new user-defined functions (UDFs). The support for ADTs is fortunate, since it allowed us to easily extend the system with device ADTs. We defined a generic device ADT and specific ADTs for the following categories of devices: smartcards, javarings, temperature sensors and door actuators.

The COUGAR system can be represented with a three-tier architecture. At the top level, the COUGAR device database server is the front-end to the device network. It is a database server that stores base relations and is able to optimize and execute long-running queries over a device network. The COUGAR database system is connected to one proxy server, which represents the middle-tier architecture. This proxy server provides the networking service abstraction, i.e. it is responsible for routing the messages addressed by Predator to groups of devices (all temperature sensors, all alarms). In our current implementation, the communication is based on TCP/IP. We are going to use other communication paradigms in the context of the SenseIT project. On each device, a proxy is responsible for the interaction with the proxy server (applying methods once or repeatedly).

We added two operators to the PREDATOR query execution engine. These operators encode the parallel and aggressive strategies we described for the execution of virtual joins. The sequential strategy does not necessitate the addition of a new operator. Indeed, the evaluation of methods represented as expressions in the condition list or in the projection list of a tuple nested loop operator already follows this strategy. We modified the existing PREDATOR optimizer so that it recognizes methods on devices and generates an execution plan. In our experiments, we force the execution strategy chosen by the optimizer.

We have not yet implemented the conservative strategy. We observe that the conservative strategy can be modeled as a combination of the aggressive strategy for obtaining the device id in the first phase and of parallel strategy to apply the method in the second phase. In this paper, we simulate the execution of the conservative virtual join by executing a first dummy query using the aggressive strategy for obtaining the device id and a second dummy query using the parallel strategy for obtaining the answer to the method. We independently run these queries and then combine the cost of these operations.

With a handful of devices we could build a demonstrator to illustrate our ideas [B+99]. However, the number of sensors and actuators that we can actually use is not sufficient to study how our system behaves in a reasonably large-scale application. We thus designed a software component that simulates the execution of asynchronous methods on a large number of devices. We used this *device simulator* for our experiments.

The device simulator simulates a network of devices, containing *N* devices. Each of these devices supports a set of methods. The results of asynchronous method execution in the device network are provided in a series of bursts. The size of a burst, noted S, corresponds to S devices providing an answer at the same time. Two consecutive bursts are separated by a time interval *T*. These three parameters (the number of devices, the number of answers provided simultaneously and the frequency at which answers are provided) characterize the flow of data produced by a device network.

5 Performance Evaluation

5.1 Experimental Setup

In this section we focus on the performance of long running queries, evaluated using a virtual join operator. We consider a simple database schema containing the base relations *Sensor(location, TemperatureSensor)* and *Control(location, Alarm)* introduced in Section 2.1. In both relations, location is represented as an integer; *TemperatureSensor* and *Alarm* are two device ADTs. We assume there is

one Alarm bell for every Temperature Sensor. Alarm supports a synchronous method called *raise()* that does not take input arguments. *TemperatureSensor* supports an asynchronous method called *sample()* that returns a string. This string encodes a set of measurements obtained before an abnormal temperature is detected. We adjust the length of this string to influence the volume of data exchanged over the device network. Indexes are created on the *TemperatureSensor* attribute of the *Sensor* relation and on the *location* attribute of the *Control* relation.

We study the performance of the system using Query 3*, modified to integrate the sample method. We can vary the selectivity of the condition on the temperature sensor location.

Query 3*: *SELECT C.Alarm.raise(),S.TemperatureSensor.sample()*
FROM Control C, Sensor S
WHERE S.location = C.location
AND S.location LIKE "Basement";*

For now, we do not model the networking infrastructure and thus the latency of the device network is the latency of our local area network. This latency is part of the reaction time being measured. We believe that this latency is a lower bound on the latency of any full-fledged device network. We believe that for our warehouse example, an interconnection network with performance characteristics similar to a local area network will be used. Thus our experiments are representative of some real life scenario.

We described our asynchronous device simulator in the previous section. Using this simulator we can vary the arrival rate of answers returned by devices. The arrival rate is defined as the number of answers produced by unit of time (the arrival rate combines the size of a burst and the frequency at which these bursts are produced). Another parameter of the simulator is the number of devices it represents. Finally, the alarm proxy simply records the point in time the *raise*() method is applied on a particular instance of the Alarm ADT.

The execution plan chosen for this query is the following. A virtual join operator is introduced to execute the asynchronous method *sample()* on the TemperatureSensor ADT. In our experiments, we force the choice of the parallel, aggressive or conservative strategy (as we discussed in Section 3 the sequential strategy does not apply to asynchronous methods). A tuple nested loop join operator selects those Alarms whose location match the location of the temperature sensor that produced a virtual tuple. The virtual join operator is the outer child of the tuple nested loop so that the tuples it produces are pipelined. Indexes are defined on the *TemperatureSensor* attribute of the Sensor relation and on the *location* attribute of the Control relation.

All the graphs we present in the following section trace the reaction time, i.e., the time elapsed between (a) the method sample returns an output on a temperature sensor and (b) the time the method *raise*() is

called on the corresponding alarm as a function of one parameter in the system. A more thorough performance evaluation will compare the various execution strategies based on a set of different metrics including the traffic in the device network and the resource usage on the devices (e.g., power consumption). This is future work[5].

5.2 Results

Our first experiment is a sanity check. Given a scenario where the number of devices should not be a factor on the performance of the system, we wanted to see whether our way of modeling devices as ADT objects inside a database relation would preserve this property. In this experiment, we increased the size of the base relation *Sensor* from 10 to 100000 tuples while holding the overall arrival rate constant. We measured the reaction time for Query 3. The graph in Figure 5 depicts the average reaction time as a function of the number of devices. The average reaction time remains constant. The indexes on the TemperatureSensor attribute in the *Sensor* relation and on the location attribute in the *Control* relation guarantee that the reaction time remains constant. The average reaction time is 600 ms for the aggressive and parallel policies; and the reaction time is 900 ms for the conservative strategy. This experiment shows also that, in terms of performance, the virtual join operator integrates well with the existing infrastructure of the query execution engine.

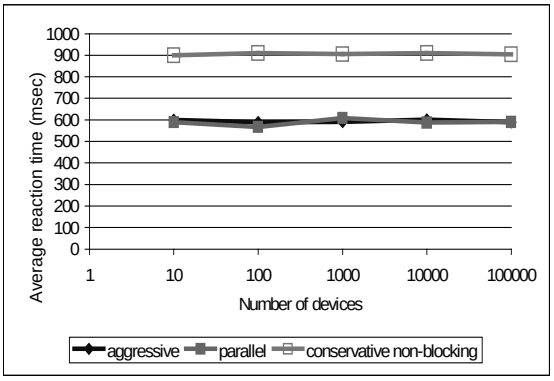


Figure 5

This experiment assumes that the average rate at which individual devices return an answer decreases in proportion to the number of devices. In practice though, the arrival rate, i.e. the rate at which the database server receives answers from the device network will increase with the number of devices. If an answer is received while the virtual join is busy, this answer is buffered in the database server.

[5] We are extending the simulator to model different networking infrastructures and we will include more performance numbers in the final version of this paper.

The reaction time for a particular answer depends on the number of answer that is waiting to be processed when the database server receives it. The number of answers waiting to be processed depends on the ratio between (a) the time it takes for the virtual join to process one answer and (b) the arrival rate.

To illustrate this point, we first increased the number of answers returned simultaneously (from 1 to 1000) in order to control the number of answers being buffered. We measured the response time for Query 3*. The graph in Figure 6 traces the average reaction time as a function of the number of answers returned simultaneously.

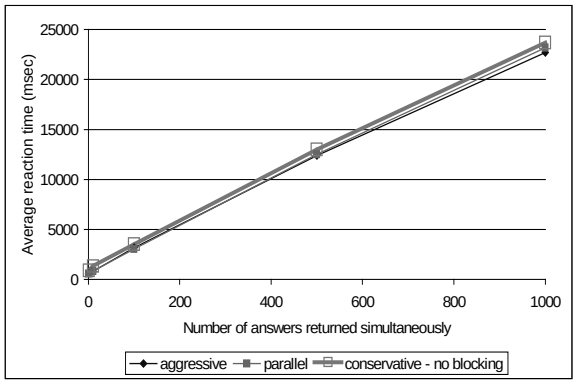
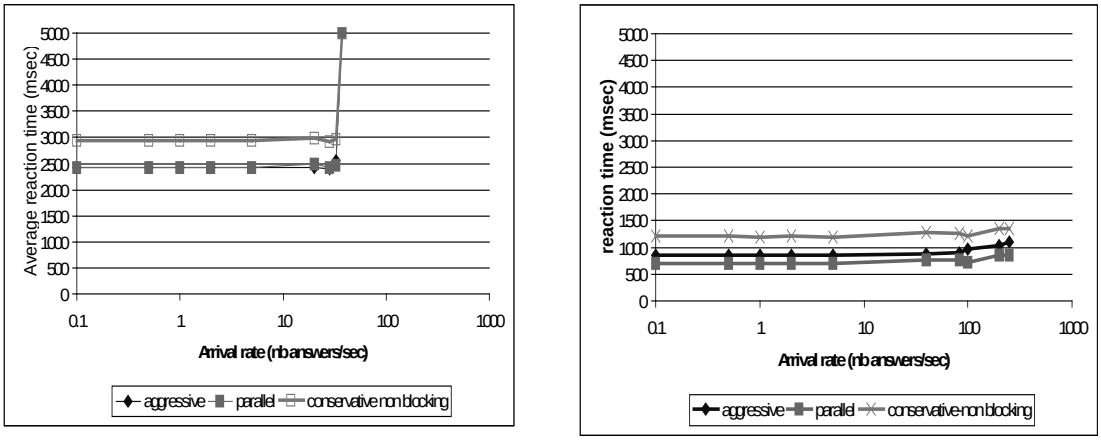


Figure 6.

Now, we illustrate the influence of the arrival rate and selectivity on the reaction time. Figures 7 a) and b) show the reaction time as a function of the arrival rate for Query 3 with two different selectivity predicates: 100% (the condition on *location* is verified for all tuples in the Sensor relation) and 10% (only 10% of the tuples in Sensor verify the condition) respectively. The arrival rate is modeled as bursts of 100 answers arriving at various frequencies.

These graphs illustrate the fact that the average reaction time is constant, when the arrival rate increases homogeneously. The system displays a thrashing behavior when a given arrival rate is reached (the thrashing point is represented by a vertical line). With 100% selectivity, all strategies reach their thrashing point simulteneously at about 40 answers per second. This was to be expected, since, in this experiment, we consider bursts of 100 tuples sent simultaneously and that Figure 6 shows that 100 tuples are processed in approximately 2.5 seconds.



a) **b)**

Figure 7

With a selectivity of 10%, the reaction time decreases and the thrashing point is reached with a higher arrival rate (it is not reached with Query 3*, we did not experiment with arrival rates greater than 250 tuples/sec). The parallel strategy profits more of a lower selectivity than the aggressive strategy (the reaction time is lower). This is because with the parallel strategy, the condition on the location of the temperature sensors is applied early and only those devices that satisfy this condition are contacted. Thus, the number of answers sent by devices decreases with respect to the aggressive strategy.

With the aggressive strategy, all devices send their answer back to the database server regardless of whether the condition on their location is verified or not. Still, the aggressive strategy also profits from a lower selectivity. This is because, with all strategies, the virtual join generates fewer tuples (only the ones that satisfy the condition on location). As a result, with a selectivity of 10%, the time to process 90% of answers is reduced to the time needed to find a matching tuple in the *Sensor* relation plus the time needed to apply the selection condition on the location attribute. For the 10% of the answers that satisfy this condition (and for all answers for a selectivity of 100%), processing time also includes the time spent in the nested loop to find a matching tuple in the Control relation and sending a message to the Alarm device. Consequently, the time to process each answer decreases and so does reaction time.

The conservative strategy can be regarded as the combination of an aggressive strategy to obtain the device identifiers and a parallel strategy to obtain the answer to the actual method. Generally, the time to obtain the answers dominates the time to obtain the device identifier and as a result the non-blocking conservative strategy behaves like the parallel strategy.

These experiments illustrate the behavior of our system and the influence of various parameters on performances. In conclusion, it appears that the parallel strategy is the most suited when optimizing for reaction time. In a more thorough performance evaluation we will need to take into account the resources used on individual devices. We will then be able to study the influence of the various virtual join execution techniques on metrics such as power usage on the devices.

6 Related Work

Device networks are already reality today [E+99, KKP99]. Example systems include the WINS system [K+, active badges [W+92], the ORL location system [WJH97], the PinPoint Positioning System [WL98], the Ubiquitous Computing project at Xerox Parc [W91], the MIT smart room project [P95] and the Forest of Sensors project [G+98]. At the networking level, work on self-organizing networks of mobile wireless nodes provides the infrastructure for ad-hoc distribution of such devices [G93, PB94, B+98, HP98].

The environment of a device network with computing power at each node resembles a mobile computing environment [IK96, BI94, IB93]. Devices in our scenario differ from mobile hosts in that devices are inherently passive, insofar they only serve external requests but do not initiate requests themselves. In this paper, we disregarded issues of transaction management [LHY99, FCL97].

We model the devices as ADTs in a database schema [S86, S98]. Our ADT approach is suitable for integration of a large number of homogeneous devices.

Alternative to the integration of devices as types in the schema, devices could be seen as external data sources that have to be integrated into query processing. Federated architectures [RS97, C+94, TRV98] allow the integration of heterogeneous data sources while permitting localized processing of partial queries at the individual data sources. Our approach seems more scaleable with respect to the number of devices that are manageable, and we intend to introduce query processing close to the devices in the near future.

To define the semantics of queries over device databases we introduced time stamps as attributes in every relation and joining conditions on these time stamps. This is a classical data representation in temporal databases [T+92, GM92, LM90, T+93]. Usually, join conditions are expressed over time intervals. Using our notion of timestamped relations, queries usually contain join conditions that have equality predicates on time stamps because timestamped base relations and timestamped synchronous virtual relations contain a tuple for every possible time stamp.

We defined snapshot and long-running semantics for queries over device databases. Similar notions have been introduced in Alert [S+91], the Tapestry system [T+92] and [LP97]. Alert [S+91] defines active queries over append-only relations (called active relations). Active queries provide a fetch_wait cursor that offers a blocking read behavior. This fetch_wait cursor provides new answers as new tuples are inserted in one of the active tables and blocks waiting while new answers are not available. The answer to the active query is the set of answers that would be obtained from a classical query executed when the cursor is closed.

The Tapestry system [T+92] defines *continuous queries* over append-only relations with time-stamps. For each continuous query, an incremental query is defined to retrieve all answers obtained in an interval of t seconds. The interval query is asked repeatedly, every t seconds, and the union of the answers it provides constitute the answer to the continuous query.

The DIOM system [LP97] generalizes this approach and defines a continual query as a triple (Q, Tcond, Stop), where Q is a SQL query, Tcond is a trigger condition and Stop is a stopping condition. The answer to the continual query is the set of answers to Q, repeatedly asked every time Tcond is true until Stop is true. This approach does not assume that the underlying relations are append-only, it assumes that any update can be performed and that all these updates are known to the system (because they have been logged or detected).

Our definition of long-running queries is similar to the definition of *continuous queries* [T+92] or *active queries* [S+91]. We however integrate the notion of synchronous and asynchronous methods that are not taken into account in these approaches. We assume that methods are applied repeatedly, as often as possible, within a time interval, because in the device network applications that we have encountered so far this semantic seems to be the most adquate. We might however need to consider more complex mechanisms for invoking functions repeatedly and introduce trigger conditions as in the DIOM system[LP97]. Event-Condition-Action rules described in the literature on active database systems [MD89,S+91,WC96,P99]provide an alternative database abstraction for accessing a device network and a complement to long-running queries. The representation of devices as ADTs and the use of virtual join operators to execute asynchronous methods on ADTs would still be relevant in this context.

We introduced a virtual join operator for the execution of asynchronous methods on device ADTs. This idea is based on the observation that the classical tuple-at-a-time execution of methods in a condition list or in a projection list does not allow overlapping the different invocations. A similar idea underlies the work on client-site UDFs [MS99] where methods are executed on the client site either as a client-site join or a semi-join between server and client. Execution as a join hides communication latencies that would dominate tuple-at-a-time execution. In our case, tuple-at-a-time would not only be

inefficient, but incorrect, because it would miss many results. Representing the execution of a method as a join for optimization purposes has been introduced in the context of LDL [CGK89] and has also been used as an execution technique in [CDY95]. The different strategies for the virtual join operator stem from the distributed database literature [SA80, ML86].

In this paper, our main goal was to introduce the idea of using database technology for querying device networks. We are currently working on more efficient processing strategies using the idea of caching data close to the devices [D+96, BI94, HSW94] and pushing processing to the devices [LP97, D+96, T+92, FJK93].

7 Conclusion

The expanding use of device networks poses an opportunity and a challenge for database technology. In the near future, very large numbers of physical objects will integrate processing power and memory. These devices will be interconnected using wireless ad-hoc networks. Device networks are starting to be used for detection, measurement and surveillance applications.

We propose to adapt database technology for providing declarative and scalable access to device networks. Our approach is based on the novel idea of representing devices as ADT objects in an object-relational database system. A characteristic of a device ADT is that methods can be asynchronous, and we introduced precise semantics for queries involving such asynchronous methods. SQL queries can now be used to obtain the temperature from a set of sensors, or to program the device network to raise an alarm every time an abnormal event is detected.

We described query evaluation techniques in a device database system. Existing object-relational query processing techniques do not work for asynchronous methods, and we introduced a new physical algebra operator called a virtual join for executing asynchronous methods. We presented alternative implementation strategies for virtual joins, described our implementation in the Cornell COUGAR system, and illustrated the performance characteristics of the different implementation strategies.

We believe that device database systems are a promising new field for database research. There are several issues concerning online query processing, transaction management, and query optimization, to only name a few, that we have not addressed in this paper. We propose to address these issues in future research.

Acknowledgements.

Dan Mahashin built the proxy subsystem and Chang Won Choi built the simulator. 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.

8 Bibliography

[B+98] Josh Broch and David A. Maltz and David B. Johnson and Yih-Chun Hu and Jorjeta Jetcheva: A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM-98), pp. 85-97, ACM Press, October 25-30 1998.

[B+99] Philippe Bonnet, Kyle Buza, Zhiyuan Chen, Victor Cheng, Randolph Chung, Takako M. Hickey, Ryan Kennedy, Daniel Mahashin, Tobias Mayr, Ivan Oprencak, Praveen Seshadri, Hubert Siu: The Cornell Jaguar System: Adding Mobility to PREDATOR. SIGMOD Conference 1999: 580-581

[BI94] Daniel Barbará, Tomasz Imielinski: Sleepers and Workaholics: Caching Strategies in Mobile Environments. SIGMOD Conference 1994: 1-12

[C+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The Tsimmis Project: Integration of Heterogeneous Information Sources. Proceedings of the 100th Anniversary Meeting of the Information Processing Society of Japan, pages 7-18, Tokyo, Japan, October 1994.

[CDY95] S.Chaudhuri, U.Dayal, T.Yan. Join Queries with External Text Sources: Execution and Optimization Techniques. In Proceedings of the 1995 ACM-SIGMOD Conference on the Management of Data. San Jose, CA.

[CGK89] D.Chimenti, R.Gamboa, and R.Krishnamurthy. Towards an Open Architecture for LDL. In Proceedings of the International VLDB Conference, Amsterdam, August 1989.

[CR96] Stefano Ceri, Raghu Ramakrishnan: Rules in Database Systems. Computing Surveys 28(1): 109-111 (1996)

[CS93] S.Chaudhuri and K.Shim. Query Optimization in the Presence of Foreign Functions. In Proceedings of the 19th International VLDB Conference, Dublin, Ireland, August 1993.

[CS97] S.Chaudhuri and K.Shim. Optimization of Queries with User-Defined Predicates. 22nd VLDB Conference, Mumbai, India, 1996.

[D+96] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, Michael Tan: Semantic Data Caching and Replacement. VLDB 1996: 330-341

[E+99] D.Estrin, R.Govindan, J.Heidemann, S.Kumar: Next Century Challenges: Scalable Coordination in Sensor Networks. Mobicom'99, Seattle, Washington: 263 – 270

[FCL97] Michael J. Franklin, Michael J. Carey, Miron Livny: Transactional Client-Server Cache Consistency: Alternatives and Performance. TODS 22(3): 315-363 (1997)

[FJK93] Michael J. Franklin, Björn Þór Jónsson, Donald Kossmann: Performance Tradeoffs for Client-Server Query Processing. SIGMOD Conf. 1996: 149-160 [A93] N.Adams: An Infrared Network for Mobile Computers. In Proc. USENIX Mobile and Location-Independent Comp. Symp., Cambridge, MA, August 1993.

[FP] The Factoid Project. http://www.research.digital.com/wrl/projects/Factoid/index.html

[G+98] W.Grimson, C.Stauffer, R.Romano, L.Lee: Using Adaptive Tracking to Classify and Monitor Activities in a Site. CVPR 1998.

[G93] J.J.Garcia-Luna-Aceves. Loop-Free Routing usin Diffusin Computations. IEEE/ACM Transactions on Networking, February 1993

[GM91] Dov M. Gabbay, Peter McBrien: Temporal Logic & Historical Databases. VLDB 1991: 423-430

[GS98] GemPlus SmartCards, Documentation available at www.gemplus.com, 1998.

[H+90] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce G. Lindsay, Hamid Pirahesh, Michael J. Carey, Eugene J. Shekita: Starburst Mid-Flight: As the Dust Clears. TKDE 2(1): 143-160 (1990)

[HN97] J.M.Hellerstein and J.F.Naughton. Query Execution Techniques for Caching Expensive Methods. In Proceedings of the 1996 ACM-SIGMOD Conference on the Management of Data, pages 423-434, Montreal, May 1996.

[HP98] Zygmunt J. Haas, Marc R. Pearlman: The Performance of Query Control Schemes for the Zone Routing Protocol. SIGCOMM 1998: 167-177

[HS93] J.M.Hellerstein and M.Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data, Washington, D.C., May 1993.

[HSW94] Yixiu Huang, A. Prasad Sistla, Ouri Wolfson: Data Replication for Mobile Computers. SIGMOD Conference 1994: 13-24

[IB93] Tomasz Imielinski, B. R. Badrinath: Data Management for Mobile Computing. SIGMOD Record 22(1): 34-39 (1993)

[IK96] T.Imielinski, H.F.Korth (editors): Mobile Computing. Kluwer Academic Publishers, Boston. 1996.

[IW98] 1-Wire Weather Station Experiment: Dallas Semi Conductor, http://www.ibutton.com/weather/index.html, 1998.

[JR98] The IButton JavaRing, Documentation available at www.ibutton.com, 1998.

[KKP99] J.M.Kahn, R.H.Katz, K.S.J.Pister: Next Century Challenges: Mobile Networking for "Smart Dust". Mobicom'99, Seattle, Washington: 271 – 277

[L+96] Ling Liu, Calton Pu, Roger S. Barga, Tong Zhou: Differential Evaluation of Continual Queries. ICDCS 1996: 458-465

[LHY99] SangKeun Lee, Chong-Sun Hwang, HeonChang Yu: Supporting Transactional Cache Consistency in Mobile Database Systems. MobiDE, Seattle, WA, 1999: 6 – 13.

[LM90] T. Y. Cliff Leung, Richard R. Muntz: Query Processing for Temporal Databases. ICDE 1990: 200-208

[LP97] L. Liu and C. Pu. ``Dynamic Query Processing in DIOM", To appear in *IEEE Bulletin of the Technical Committee on Data Engineering,* Special issues on Improving Query Responsiveness, September 1997 Vol.20, No. 3

[MD89] Dennis R. McCarthy, Umeshwar Dayal: The Architecture Of An Active Data Base Management System. SIGMOD Conference 1989: 215-224

[ML86] L.F.Mackert, G.M.Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In Proceedings of the International VLDB Conference, pages 149-159, Kyoto, Japan, August 1986. Management Systems. ICOD 1980: 204-215

[MS99] Tobias Mayr and Praveen Seshadri: Client-Site Query Extensions. In Proceedings of the ACM SIGMOD Conference 1999, Philadelphia, PA, June 1999.

[P95] A.Pentland: Machine Understanding of Human Action. In Proc. Of 7th Int. Forum Frontier of Telecommunication Tech., November 1995.

[P99] Norman W.Paton (Editor): Active Rules in Database Systems. Springer Verlag, New York, 1999.

[P99] The Cornell Predator Project: http://www.cs.cornell.edu/database

[PB94] Charles Perkins and Pravin Bhagwat: Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications, pp. 234-244, August 1994.

[RS97] Mary T. Roth, Peter Schwarz: Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. 23rd VLDB Conference Athens, Greece, 1997.

[S+91] Ulf Schreier, Hamid Pirahesh, Rakesh Agrawal, C. Mohan: Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. VLDB 1991: 469-478

[S86] Michael Stonebraker: Inclusion of New Types in Relational Data Base Systems. ICDE 1986: 262-269

[S98] Praveen Seshadri. Enhanced Abstract Data Types in Object-Relational Databases. VLDB Journal 7(3): 130-140 (1998).

[S99] DARPA: SenseIT Project: http://www.darpa.mil/ito/research/sensit/background.html

[SA80] Patricia G. Selinger, Michel E. Adiba: Access Path Selection in Distributed Database Management System. ACM SIGMOD 1979, p.23-34, Boston, MA, USA, June 1979.

[Sesh98] Praveen Seshadri. Enhanced Abstract Data Types in Object-Relational Databases. VLDB Journal 7(3): 130-140 (1998).

[T+92] Douglas B. Terry, David Goldberg, David Nichols, Brian M. Oki: Continuous Queries over Append-Only Databases. SIGMOD Conference 1992: 321-330

[T+93] Abdullah Uz Tansel, James Clifford, Shashi K. Gadia, Arie Segev, Richard T. Snodgrass (Eds.): Temporal Databases: Theory, Design, and Implementation. Benjamin/Cummings 1993, ISBN 0-8053-2413-5

[TRV98] A.Tomasic, L.Raschid,P.Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. IEEE Trans. on Knowledge and Data Engineering, 10(5), September/October 1998.

[W98] UCLA: The WINS Project: http://www.janet.ucla.edu/wins, 1998.

[W+92] Roy Want, Andy Hopper, Veronica Falcao, Jonathan Gibbons: The Active Badge Location System. TOIS 10(1): 91-102 (1992)

[W91] M.Weiser: The Computer for the 21st Century. Scientific American, September 1991.

[WC96] Jennifer Widom, Stefano Ceri: Active Database Systems: Triggers and Rules For Advanced Database Processing. Morgan Kaufmann 1996, ISBN 1-55860-304-2

[WJH97] A.Ward, A.Jones, A.Hopper: A New Location Technique for the Active Office. IEEE Personal Communications, 4(5), October 1997.

[WL98] J.Werb and C.Lanzl: Designing a Positioning System for Finding Things and People Indoors. IEEE Spectrum, September 1998.