Towards Sensor Database Systems

Philippe Bonnet, Johannes Gehrke, Praveen Seshadri[1]

Computer Science Department, Upson Hall
Cornell University
Ithaca, NY, 14853 USA
{bonnet,johannes,praveen}@cs.cornell.edu

Abstract. Sensor networks are being widely deployed for measurement, detection and surveillance applications. In these new applications, users issue long-running queries over a combination of stored data and sensor data. Most existing applications rely on a centralized system for collecting sensor 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 regardless of the queries that are submitted. In our new concept of sensor database system, queries dictate which data is extracted from the sensors. In this paper, we define the concept of sensor databases mixing stored data represented as relations and sensor data represented as time series. Each long-running query formulated over a sensor database defines a persistent view, which is maintained during a given time interval. We also describe the design and implementation of the COUGAR sensor database system.

1 Introduction

The widespread deployment of sensors is transforming the physical world into a computing platform. Modern sensors not only respond to physical signals to produce data, they also embed computing and communication capabilities. They are thus able to store, process locally and transfer the data they produce. Still, at the heart of each sensor, a set of signal processing functions transform physical signals such as heat, light, sound, pressure, magnetism, or a particular motion into sensor data, i.e., measurements of physical phenomena as well as detection, classification or tracking of physical objects.

Applications monitor the physical world by querying and analyzing sensor data. Examples of monitoring applications include supervising items in a factory warehouse, gathering information in a disaster area, or organizing vehicle traffic in a large city [6]. Typically, these applications involve a combination of stored data (a list of sensors and their related attributes, such as their location) and sensor data. We call *sensor database* the combination of stored data and sensor data. This paper focuses on sensor query processing – the design, algorithms, and implementations used to run

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

queries over sensor databases. The concepts developed in this paper were developed under the DARPA Sensor Information Technology (SensIT) program [23].

We define a *sensor query* as a query expressed over a sensor database. A typical monitoring scenario involves aggregate queries or correlation queries that give a bird's eye view of the environment as well as queries zooming on a particular region of interest. Representative sensor queries are given below in Example 1.

Example 1 (Factory Warehouse): Each item of a factory warehouse has a stick-on temperature sensor attached to it. Sensors are also attached to walls and embedded in floors and ceilings. Each sensor provides two signal-processing functions: (a) *getTemperature()* returns the measured temperature at regular intervals, and (b) *detectAlarmTemperature(threshold)* returns the temperature whenever it crosses a certain threshold. Each sensor is able to communicate this data and/or to store it locally. The sensor database stores the identifier of all sensors in the warehouse together with their location and is connected to the sensor network. The warehouse manager uses the sensor database to make sure that items do not overheat. Typical queries that are run continuously include:

– Query 1: "Return repeatedly the abnormal temperatures measured by all sensors."
– Query 2: "Every minute, return the temperature measured by all sensors on the third floor".
– Query 3: "Generate a notification whenever two sensors within 5 yards of each other simultaneously measure an abnormal temperature".
– Query 4: "Every five minutes retrieve the maximum temperature measured over the last five minutes".
– Query 5: "Return the average temperature measured on each floor over the last 10 minutes".

These example queries have the following characteristics:

- Monitoring queries are long-running.
- The desired result of a query is typically a series of notifications of system activity (periodic or triggered by special situations).
- Queries need to correlate data produced simultaneously by different sensors.
- Queries need to aggregate sensor data over time windows.
- Most queries contain some condition restricting the set of sensors that are involved (usually geographical conditions).

As in relational databases, queries are easiest to express at the logical level. Queries are formulated regardless of the physical structure or the organization of the sensor network. The actual structure and population of a sensor network may vary over the lifespan of a query.

Clearly, there are similarities with relational database query processing. Indeed, most applications combine sensor data with stored data. However, the features of sensor queries described here do not lend themselves to easy mapping to relational databases and sensor data is different from traditional relational data (since it is not stored in a database server and it varies over time).

There are two approaches for processing sensor queries: the warehousing approach and the distributed approach. The warehousing approach represents the current state-of-the-art. In the warehousing approach, processing of sensor queries and access to the sensor network are separated. (The sensor network is simply used by a data collection mechanism.) The warehousing approach proceeds in two steps. First, data is extracted from the sensor network in a predefined way and is stored in a database located on a unique front-end server. Subsequently, query processing takes place on the centralized database. The warehousing approach is well suited for answering predefined queries over historical data.

The distributed approach has been described by Bonnet et al. in [2][3] and is the focus of this paper. In the distributed approach, the query workload determines the data that should be extracted from sensors. The distributed approach is thus flexible – different queries extract different data from the sensor network – and efficient – only relevant data are extracted from the sensor network. In addition, the distributed approach allows the sensor database system to leverage the computing resources on the sensor nodes: a sensor query can be evaluated at the front-end server, in the sensor network, at the sensors, or at some combination of the three.

In this paper, we describe the design space for a sensor database system and present the choices we have made in the implementation of the Cornell COUGAR system. This paper makes the following contributions:

1. We build on the results of Seshadri et al. [20] to define a data model and long-running queries semantics for sensor databases. A sensor database mixes stored data and sensor data. Stored data are represented as relations while sensor data are represented as time series. Each long-running query defines a persistent view, which is maintained during a given time interval.
2. We describe the design and implementation of the Cornell COUGAR sensor database system. COUGAR extends the Cornell PREDATOR object-relational database system. In COUGAR, each type of sensor is modeled as a new Abstract Data Type (ADT). Signal-processing functions are modeled as ADT functions that return sensor data. Long-running queries are formulated in SQL with little modifications to the language. To support the evaluation of long-running queries, we extended the query execution engine with a new mechanism for the execution of sensor ADT functions. The initial version of this system has been demonstrated at the Intel Computing Continuum Conference [7].

Addressing these two issues is a necessary first step towards a sensor database system. In addition, a sensor database system should account for sensor and communication failures; it should consider sensor data as measurements with an associated uncertainty not as facts; finally, it should be able to establish and run a distributed query execution plan without assuming global knowledge of the sensor network. We believe that these challenging issues can only be addressed once the data model and internal representation issues have been solved.

2 Sensor Database Systems

In this section, we introduce the concepts of sensor databases and sensor queries. We build on existing work by Seshadri et al [20] to define a data model for sensor data and an algebra of operators to formulate sensor queries.

2.1 Sensor Data

A sensor database involves stored data and sensor data. Stored data include the set of sensors that participate in the sensor database together with characteristics of the sensors (e.g., their location) or characteristics of the physical environment. These stored data are best represented as relations. The question is: how to represent sensor data? First, sensor data are generated by signal processing functions. Second, the representation we choose for sensor data should facilitate the formulation of sensor queries (data collection, correlation in time, and aggregates over time windows).

Note that time plays a central role. Possibly, signal processing functions return output repeatedly over time, and each output has a time-stamp. In addition, monitoring queries introduce constraints on the sensor data time-stamps, e.g., Query 3 in Example 1 assumes that the abnormal temperatures are detected either simultaneously or within a certain time interval. Aggregates over time windows, such as in Query 4 and 5, reference time explicitly.

Given these constraints, we represent sensor data as time series. Our representation of sensor time series is based on the sequence model introduced by Seshadri et al. [20]. Informally, a sequence is defined as a 3-tuple containing a set of records R, a countable totally ordered domain O (called ordering domain – the elements of the ordering domain are referred to as positions) and an ordering of R by O. An ordering of a set of records R by an ordering domain O is defined as a relation between O and R, so that every record in R is associated with some position in O. Sequence operators are n-ary mappings on sequences; they operate on a given number of input sequences producing a unique output sequence. All sequence operators can be composed. Sequence operators include: select, project, compose (natural join on the position), and aggregates over a set of positions. Because of space limitation, we refer the reader to Bonnet et al. [4] for a formal definition of sensor time series

We represent sensor data as a time series with the following properties:

1. The set of records corresponds to the outputs of a signal processing function over time.
2. The ordering domain is a discrete time scale, i.e. a set of time quantum; to each time quantum corresponds a position. In the rest of the paper, we use natural numbers as the time-series ordering domain. Each natural number represents the number of time units elapsed between a given origin and any (discrete) point in time. We assume that clocks are synchronized and thus all sensors share the same time scale.
3. All outputs of the signal processing function that are generated during a time quantum are associated to the same position p. Note that in case a sensor does not generate events during the time quantum associated to a position, the Null record is associated to that position.
4. Whenever a signal processing function produces an output, the base sequence is updated at the position corresponding to the production time. Updates to sensor time series thus occur in increasing position order.

2.2 Sensor Queries

Sensor queries involve stored data and sensor data, i.e. relations and sequences. We define a sensor query as an acyclic graph of relational and sequence operators. The inputs of a relational operator are base relations or the output of another relational operator; the inputs of a sequence operator are base sequences or the output of another sequence operator. Thus relations are manipulated using relational operators and sequences are manipulated using sequence operators. We have defined two other operators that combine relations and sequences: (a) a projection operator that takes a sequence as input and outputs a relation by projecting out the position attribute in the sequence, and (b) a cross product operator that takes as input a relation and a sequence and outputs a sequence by performing a position-wise cross product.

Sensor queries are long running. To each sensor query is associated a time interval of the form [O, O + T] where O is the time at which the query is submitted and T is the number of time quantums (possibly 0) during which the query is running.

During the span of a long-running query, relations and sensor sequences might be updated. An update to a relation R can be an insert, a delete, or modifications of a record in R. An update to a sensor sequence S is the insertion of a new record associated to a position greater than or equal to all the undefined positions in S (see Section 3.1.1). Concretely, each sensor inserts incrementally the set of records produced by a signal processing function at the position corresponding to the production time.

A sensor query defines a view that is persistent during its associated time interval. This persistent view is maintained to reflect the updates on the sensor database. In particular, the view is maintained to reflect the updates that are repeatedly performed on sensor time series.

Jagadish et al. [13] showed that persistent views over relations and sequences can be maintained incrementally without accessing the complete sequences, given restrictions on the updates that are permitted on relations and sequences, and given restrictions on the algebra used to compose queries. Informally, persistent views can be maintained incrementally if updates occur in increasing position order and if the algebra used to compose queries does not allow sequences to be combined using any relational operators. Both conditions hold in our concept of a sensor database.

3 The COUGAR Sensor Database System

In this section, we discuss the representation of sensor data, as well as the formulation and evaluation of sensor queries in the initial version of COUGAR. We discuss the limitations of this system and the conclusions that we have drawn.

We have introduced in Section 2 the concept of a sensor database. We took a set of design decisions when implementing this model in the COUGAR system. We distinguish between the decisions we took concerning:

1. User representation: How are sensors and signal processing functions modeled in the database schema? How are queries formulated?
2. Internal representation: How is sensor data represented within the database components that perform query processing? How are sensor queries evaluated to provide the semantics of long-running queries?

3.1 User Representation

In COUGAR, signal-processing functions are represented as Abstract Data Type (ADT) functions. Today's object-relational databases support ADTs that provide controlled access to encapsulated data through a well-defined set of functions [21]. We create a sensor ADT for all sensors of a same type (e.g., temperature sensors, seismic sensors). The public interface of a sensor ADT corresponds to the specific signal-processing functions supported by a type of sensor. An ADT object in the database corresponds to a physical sensor in the real world.

Signal-processing functions are modeled as scalar functions. Repeated outputs of an active signal processing functions are not explicitly modeled as sequences but as the result of successive executions of a scalar function during the span of a long-running query. This decision induced some limitation. For example, as we will see below, queries containing explicit time constraints (such as aggregates over time windows) cannot be expressed.

Sensor queries are formulated in SQL with little modifications to the language. The 'FROM' clause of a sensor query includes a relation whose schema contains a sensor ADT attribute (i.e., a collection of sensors). Expressions over sensor ADTs can be included in the 'SELECT' or in the 'WHERE' clause of a sensor query.

The queries we introduced in Section 1 are formulated in COUGAR as follows. The simplified schema of the sensor database contains one relation *Sensors(loc POINT, floor INT, s SENSORNODE)*, where *loc* contains the location of the sensor, *floor* denotes the floor where the sensor is located in the data warehouse and *s* represents a sensor node. *SENSORNODE* is a sensor ADT that supports the methods *getTemp()* and *detectAlarmTemp(threshold)*, where *threshold* is the temperature above which abnormal temperatures are returned. Both ADT functions return temperature represented as float.

- Query 1: "Return repeatedly the abnormal temperatures measured by all sensors"
  SELECT Sensors.s.detectAlarmTemp(100)
  FROM Sensors
  WHERE $every();
  The expression *$every()* is introduced as a syntactical construct to indicate that the query is long-running.
- Query 2: "Every minute, return the temperature measured by all sensors on the third floor".
  SELECT Sensors.s.getTemp()
  FROM Sensors
  WHERE Sensors.floor = 3 AND $every(60);
  The expression *$every()* takes as argument the time in seconds between successive outputs of the sensor ADT functions in the query.
- Query3: "Generate a notification whenever two sensors within 5 yards of each other measure simultaneously an abnormal temperature".
  SELECT R1s.detectAlarmTemp(100), R2.s. detectAlarmTemp (100)
  FROM Sensors R1, Sensors R2
  WHERE $SQRT($SQR(R1.loc.x – R2.loc.x) + $SQR(R1.loc.y – R2.loc.y)) < 5
  AND R1.s > R2.s AND $every();
  This formulation assumes that the system incorporates an equality condition on the time at which the temperatures are obtained from both sensors.

Query 4 and 5 cannot be expressed in our initial version of COUGAR because aggregates over time windows are not supported.

3.2 Internal Representation

Query processing takes place partly on a database front-end. The query execution engine on the database front-end includes a mechanism for interacting with remote sensors. On each sensor a lightweight query execution engine is responsible for executing signal processing functions and sending data back to the front-end.

In COUGAR, we assume that there are no modifications to the stored data during the execution of a long-running query. Strict two-phase locking on the database front-end ensures that this assumption holds.

The initial version of COUGAR does not consider a long-running query as a persistent view; the system only computes the incremental results that could be used to maintain such a view. These incremental results are obtained by evaluating sensor ADT functions repeatedly and by combining the outputs they produce over time with stored data.

The execution of Sensor ADT functions is the central element of sensor queries execution. In the rest of the section, we show why the traditional execution of ADT functions (which is explained below) is inappropriate for sensor queries and we present the mechanisms we have implemented in COUGAR to evaluate sensor ADT functions.

Problems with the Traditional ADT Functions Execution

In most object-relational database systems, ADT functions are used to form expressions together with constants and variables. When an expression containing an ADT function is evaluated, a (local) function is called to obtain its return value. It is assumed that this return value is readily available on-demand. As for client-side ADT functions [15], this assumption does not hold in a sensor database for the following reasons:

1. Scalar sensor ADT functions incur high latency due to their location or because they are asynchronous;
2. When evaluating long-running queries, sensor ADT functions return multiple outputs.

To illustrate these problems, let us consider Query 1 in our example. One possible execution plan for Query 1 would be the following. For each temperature sensor in the relation R, the scalar function detectAlarmTemp(100) is applied.

There is a serious flaw in this execution. First, the function *detectAlarmTemp (100)* is asynchronous, i.e. it returns its output after an arbitrary amount of time. While the system is requesting an abnormal temperature on the first sensor of the relation *R*, the other temperature sensors have not been yet been contacted. It may very well be that some temperature sensors could have detected temperatures greater than 100 while the system is blocked waiting for the output of one particular function.

Second, during the span of a long-running query, *detectAlarmTemp (100)* might return multiple outputs. The evaluation plan presented above scans relation *R* once and thus does not respect the semantics of long running queries we have introduced in Section 2.

Virtual Relations

To overcome the problems outlined in the previous paragraph, we introduced a relational operator to model the execution of sensor ADT functions. This relational operator is a variant of a join between the relation that contains the sensor ADT attribute and the sensor ADT function represented in a tabular form. We call the tabular representation of a function a virtual relation.

A virtual relation is a tabular representation of a method. A record in a virtual relation (called a virtual record) contains the input arguments and the output argument of the method it is associated with.[2] Such relations are called virtual because they are not actually materialized, as opposed to base relations, which are defined in the database schema.

If a method M takes m arguments, then the schema of its associated virtual relation 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. In our example Query 1, the virtual relation *VRdetectAlarmTemp* is defined for the Sensor ADT function *detectAlarmTemp().* Since this function takes one input arguments, the virtual relation has four attributes: *SensorId, Temp, Value, and TimeStamp, i.e.*, the identifier of the Sensor device that produces the data *SensorId*, the input threshold temperature *Temp*, the *Value* of the measured temperature and the associated *TimeStamp*.

We observe the following:

- A virtual relation is append-only: New records are appended to a virtual relation when the associated signal processing 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 sensor ADT: A virtual relation is associated to a sensor ADT function, to each sensors of these type is associated a fragment of the virtual relation. The virtual relation is the union of all these fragments.

The latter observation has an interesting consequence: a device database is internally represented as a distributed database. Virtual relations are partitioned across a set of devices. Base relations are stored on the database front-end. Distributed query processing techniques are not implemented in the initial version of COUGAR; their design and integration is the main goal of COUGAR V2 that we are currently implementing.

Query Execution Plan

Virtual relations appear in the query execution plan at the same level as base relations. Base relations are accessed through (indexed) scans. Each virtual relation fragment is accessed on sensors using a virtual scan. A virtual scan incorporates in the query execution plan the propagation mechanism necessary to support long-running queries.

Our notion of virtual scan over a virtual relation fragment is similar to the fetch_wait cursor over an active relation in the Alert database system [19]. A fetch_wait cursor provides a blocking read behavior. This fetch_wait cursor returns new records as they are inserted in the active relation and blocks when all records have been returned. A classical cursor would just terminate when all records currently in the relation have been returned.

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

The join between a base relation and a virtual relation is basically a nested loop with a pipelined access to the virtual scans that encapsulate the execution of the sensor ADT function. (Note that we make the simplifying assumption that arguments to the sensor ADT functions are constants.) Indeed, the sensor ADT function is applied with identical parameters on all sensors involved in the query. The algorithm is presented below.

```
In: Base relation R, sensor ADT function f
Out: join between relation R and virtual relation associated to f
Initialize virtual scans for the virtual relation fragments associated to f on all sensors
involved in the query
FOREVER DO
  Get next output from the sensor virtual scan
  Find a matching sensor id in the base relation R
  If match is found then return record
END DO
```

The incremental results produced by a virtual join are directly transmitted to the client, or they are pipelined to the root of the execution plan (as the outer child in a nested loop join for instance)[3]. Consequently, queries with relational aggregates or ‘ORDER BY’ clauses do not return an incremental result. Indeed, such queries require an operator to accumulate all the results produced by its children. With such operators no incremental results are produced before the query is stopped.

3.3 Conclusions

Here are the conclusions that we have drawn from our experience with the initial version of COUGAR:

1. Representing stored data as relations with an ADT attribute representing sensors and sensor data as the output of ADT functions is a natural way of representing a sensor database.
2. Virtual joins are an effective way of executing ADT functions that do not return a value in a timely fashion (because they are often asynchronous, because they generally incur high latency or because they return multiple values over time).
3. Representing all signal processing functions as scalar functions fails to capture the ordering of sensor data in time. As a result, queries involving aggregates over time windows or correlations are difficult to express. This problem has previously been identified in the context of financial data [22].

4 Related Work

Two representative projects that build wireless sensor network infrastructures are the WINS project at UCLA [18] and the Smart Dust project at UC Berkeley [14]. The COUGAR system is implemented on top of the WINS infrastructure.

The goals of the DataSpace project at Rutgers University are quite similar to the goals of a sensor database system [9]. Imielinski et al. recognized the advantages of

[3] Note that queries with sensor ADT functions applied on more than one collection of sensors require that the join between two virtual joins is a double-pipelined join.

the distributed approach over the warehousing approach for querying physical devices. In a DataSpace, devices that encapsulate data can be queried, monitored and controlled. Network primitives are developed to guarantee that only relevant devices are contacted when a query is evaluated. We are currently integrating COUGAR with similar networking primitives, i.e., the Declarative Routing Protocol developed at MIT-LL [5], and the SCADDS diffusion-based routing developed at ISI [10]. Other related projects include the TELEGRAPH project at UC Berkeley [1], which studies adaptive query processing techniques, and the SAGRES project at the University of Washington [11], which explores the use of data integration techniques in the context of device networks.

The environment of a sensor network with computing power at each node resembles a mobile computing environment [8]. Sensors differ from mobile hosts in that sensors only serve external requests but do not initiate requests themselves. Also, recent work on indexing moving objects, e.g. [17], is relevant in such environments. The techniques proposed however assume a centralized warehousing approach.

Our definition of sensor queries bears similarities with the definition of continuous queries [23]. Continuous queries are defined 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 incremental query is issued repeatedly, every t seconds, and the union of the answers it provides constitute the answer to the continuous query. Instead of being used to maintain a persistent view, incremental results are directly returned to users. The answers returned by the initial prototype of COUGAR respect the continuous queries semantics.

Time series can be manipulated in object-relational systems such as Oracle [16] or in array database systems such as KDB [12]. These systems do not support the execution of long-running queries over sequences.

5 Conclusion

We believe that sensor database systems are a promising new field for database research. We described a data model and long-running query semantics for sensor database systems where stored data are represented as relations and sensor data are represented as sequences. The version of the Cornell COUGAR system that we presented is a first effort towards such a sensor database system. The second version of COUGAR [4] improves on the initial prototype in that sequences are explicitly represented. This allows for more expressive sensor queries. In particular, queries containing aggregates over time windows can be expressed.

This first generation of the Cornell COUGAR systems showed much promise for providing flexible and scalable access to large collections of sensors. It helped us identify a set of challenging issues that we are addressing with our ongoing research:

- Due to the large scale of a sensor network, it is highly probable that some of the sensors and some of the communication links will fail at some point during the processing of a long-running query. We are studying how sensor database systems can adjust to communication failures and return a more accurate answer at the cost of increased response time and resource usage.
- Sensor data are measurements not facts. Sensor values can be thought of as drawn from a continuous distribution, e.g. a normal distribution. We are working on a data model and an associated algebra for representing and manipulating continuous distributions.

- Because of the large scale and dynamic nature of a sensor network, we cannot assume that a centralized optimizer maintains global knowledge and thus precise meta-information about the whole network. We are studying how to maintain meta-data in a decentralized way and how to utilize this information to devise good query plans.

Acknowledgements
We would like to thank Tobias Mayr and Raoul Bhoedjiang who helped debug earlier versions of this paper. This paper benefited from interactions with the SensIT community. In particular, Bill Kaiser provided valuable information concerning the Sensoria WINS network. Tok Wee Hyong has implemented most of the sequence ADT extension for COUGAR V2. Joe Hellerstein suggested the relevance of sequences for sensor databases. 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 and by the National Science Foundation under Grant No. EIA 97-03470

References

1. Ron Avnur, Joseph M. Hellerstein: Eddies: Continuously Adaptive Query Processing. SIGMOD Conference 2000: 261-272
2. Ph. Bonnet, P.Seshadri. Device Database Systems. Proceedings of the International Conference on Data Engineering ICDE'99, San Diego, CA, March, 2000.
3. Ph.Bonnet, J.Gehrke, P.Seshadri. Querying the Physical World. IEEE Personal Communications. Special Issue ìNetworking the Physical Worldî. October 2000.
4. Ph.Bonnet, J.Gehrke, P.Seshadri. Towards Sensor Database Systems. Cornell CS Technical Report TR2000-1819. October 2000
5. D.Coffin, D.Van Hook, S.McGarry, S.Kolek. Declarative AdHoc Sensor. SPIE Integrated Command Environments. 2000.
6. D.Estrin, R.Govindan, J.Heidemann (Editors): Embedding the Internet. CACM 43(5) (2000)
7. The Intel Computing Continuum Conference, San Francisco, May, 2000. http://www.intel.com/intel/cccon/
8. Tomasz Imielinski, B. R. Badrinath: Data Management for Mobile Computing. SIGMOD Record 22(1): 34-39 (1993)
9. Tomasz Imielinski, Samir Goel: DataSpace - Querying and Monitoring Deeply Networked Collections in Physical Space. MobiDE 1999: 44-51
10. C.Intanagonwiwat, R.Govindan, D.Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. Mobicom'00.
11. Z. G. Ives, A. Y. Levy, J. Madhavan, R. Pottinger, S. Saroiu, I. Tatarinov, S. Betzler, Q. Chen, E. Jaslikowska, J. Su, W. Tak and T.Yeung: Self-Organizing Data Sharing Communities with SAGRES. SIGMOD Conference 2000: 582
12. Kx Systems Home Page: http://www.kx.com.
13. H. V. Jagadish, Inderpal Singh Mumick, Abraham Silberschatz: View Maintenance Issues for the Chronicle Data Model. PODS 1995: 113-124
14. J. M. Kahn, R. H. Katz and K. S. J. Pister, "Mobile Networking for Smart Dust", ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99), Seattle, WA, August 17-19, 1999

15. Tobias Mayr and Praveen Seshadri: Client-Site Query Extensions. In Proceedings of the ACM SIGMOD Conference 1999, Philadelphia, PA, June 1999.
16. Oracle8™ Time Series Data Cartridge. 1998. http://www.oracle.com/
17. Dieter Pfoser, Christian S. Jensen, Yannis Theodoridis: Novel Approaches in Query Processing for Moving Objects. VLDB 2000:
18. G.Pottie, W. Kaiser: Wireless Integrated Network Sensors (WINS): Principles and Approach. CACM 43(5) (2000)
19. U. Schreier, H. Pirahesh, R. Agrawal, C. Mohan: Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. VLDB 1991: 469-478
20. Praveen Seshadri, Miron Livny, Raghu Ramakrishnan: SEQ: A Model for Sequence Databases. ICDE 1995: 232-239
21. P. Seshadri. Enhanced Abstract Data Types in Object-Relational Databases. VLDB Journal 7(3): 130-140 (1998).
22. D.Shasha: Time Series in Finance: The Array Database Approach. 1998. http://cs.nyu.edu/shasha/papers/jagtalk.html
23. D.Tennenhouse: Proactive Computing. CACM 43(5) (2000)
24. Douglas B. Terry, David Goldberg, David Nichols, Brian M. Oki: Continuous Queries over Append-Only Databases. SIGMOD Conference 1992: 321-330