DBMS on Small Scale Devices

Based on the papers:
“PicoDBMS: Scaling down database techniques for the smartcard” by Philippe Pucheral, Luc Bouganim, Patrick Valduriez, Christophe Bobineau

“Smart card embedded information systems: a methodology for privacy oriented architectural design” by C. Bolchini, F.A. Schreiber

By Amy Nathanson

Agenda
• Small device (smartcard) overview
• Problems with DBMS on smartcards
• Solutions
  – PicoDBMS: a new database architecture
  – A methodology for Privacy Management
• Summary of DBMS on smartcards

Overview of small devices
• Portable computing device
  – Secure
  – widely used
    • Banking
    • Healthcare
    • Insurance
• Smartcards (example: credit card)
  – Single, issuer-dependent application
    • Moving to multi-application
    • Merge many cards to one

Smartcards and DBMS
• Volume of data growing
• Complexity of queries increasing
• Privacy issues
• ACID
• Separate management code from data code
• High security and availability

Problems of scaling down DBMS for small devices
• Small size and low cost
  – 96 kB ROM → stores OS, fixed data, standard routines
  – 4 kB RAM → for the stack and calculations
  – 128 kB EEPROM → persistent data
  • VERY slow write time (> 1ms/word)

<table>
<thead>
<tr>
<th>Memory type</th>
<th>EEPROM</th>
<th>FLASH</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read time (ms)</td>
<td>10 to 150</td>
<td>75 to 200</td>
<td>150 to 300</td>
</tr>
<tr>
<td>Write time (ms)</td>
<td>1 to 2ms</td>
<td>5 to 10ms</td>
<td>150 to 300ms</td>
</tr>
<tr>
<td>Erase time (ms)</td>
<td>None</td>
<td>50 to 800ms</td>
<td>None</td>
</tr>
<tr>
<td>Operations</td>
<td>10^9 write cycles</td>
<td>10^9 erase cycles</td>
<td>10^9 to 10^12 write cycles</td>
</tr>
</tbody>
</table>

* A memory cell can be overwritten a finite number of times.

Design Requirements for DBMS
• Minimize data structure size
• Minimize RAM usage
• Minimize write operations
• Maximize fast read and direct access capability of stable memory
• Don’t externalize private data and minimize algorithm complexity for security
• Enforce ACID
Storage model problem: scale down data structures

- **FS (flat storage)** – tuples stored sequentially with attributes imbedded
  - Space consuming and inefficient
- **DS (domain storage)** – factoring out values into a domain table for data compactness
  - Factor out values and put in domain
  - Only use if:
    1. data size > pointer size
    2. and duplicates exist

RS (ring storage) – index compactness

Value 1
Value 2
Value n

Index on S.a

Ring index on a regular attribute

Relation R
Relation S
Relation S

Ring index on a foreign-key attribute

SOLUTION: Use a combination of FS, DS and RS

Query Processing problem: use no RAM, no write and simple algorithms

- **Select/Project/Join queries**
  - Query Execution Plan (QEP) should be extreme one-side tree for no RAM usage
  - Implement pipelining using Iterator Model so no need for materialization
  - Project operators are pushed up (no materialization)
  - Ring index makes it time consuming to find values for attributes... done at end

Query Processing problem cont...

- **Aggregate/Sort/Duplicate Removal queries**
  - Group incoming tuples by distinct values
  - Begin with group-by attribute and join with domain table
  - Pipelines aggregate/duplicate removal queries
  - Order is preserved in pipeline operators because they handle tuples in arrival order

- **Solution**: Use pipelining and enforce order at tree leaves

Query Processing Example

Query = Number of prescriptions per type of drug.

```
count
prescription

prescription

prescription
drug

drug

drug

drug

drug

drug

Drug(DrugId, name, type, ...)

Prescription(VisId, DrugId, qty, ...)
```

Transaction Management problem: enforce ACID

- **Atomicity**: commit or rollback persistency
  - Local: enforced by write-ahead logging (WAL)
    - Problem: cost is higher in small DBMS
  - Global: enforced by ACP
- **Consistency**: tradition form used because all integrity constraints satisfied
- **Isolation**: not an issue because single-user
- **Durability**: committed updates never lost
Issues with Local Atomicity:
Shadow update vs. Update in-place

- Shadow update
  - Poorly adapted to pointer-based storage
  - Object location changes for every update
  - Writing costs are high
- Update in-place
  - WAL needed to undo aborted transactions
    - Relative cost is higher in smaller DBMS
  - Pointer-based logging
    - Log tuple location instead of value
  - Deferred garbage-collector
    - Delete all unreferenced domain values

Summary of PicoDBMS

- Smartcards becoming multipurpose
- Need database to handle data
- Problems with hardware size to fit on card
  - small ROM: change architecture
  - no write/RAM: change query execution

Database Privacy Design
Methodology for smartcards

- Problem: privacy is an issue
- Cards get stolen
- Many different people trying to access them
- Many different organizations have access
- How to manage privacy?

Privacy Issues For Smartcards

- Only store data on the card and let the user be responsible for duplicate information
- Don’t put identification on cards that provide access codes
  - Example: pre-paid calling card
- Segregate the card into zones for multi-app use
  - Use cryptography at the storage level
  - Use views at the conceptual/logical level
- Use public key cryptography

Methodology for privacy management

1. Database: choose user access profiles design
   - Customized owner-multiguest schema
   - Generalized owner-guest schema

2. Physical mechanism to protect data
   - Three possible user types to access smartcard:
     1. Data owner—authority or institution with read/write access
     2. Guest—another institution viewing information not in its domain with read-only access
     3. Card holder—has read-only access to all information on the card

1. Database privacy management

- Customized owner-multiguest schema
  - Definition of views
  - Identification of groups of users
  - Customization of access permissions for users within groups
  - Data owner specification of:
    1. Relations
    2. Set of views for each relation
    3. List of guests (read permission only)

- Generalized owner-guest schema
  - Designer specifies:
    1. Relations it owns
    2. Set of views for the relations
## 2. Physical Privacy Management

- Each view of database is determined by user to enforce data protection

- Common format system table for a smartcard
  - Object description table—tables and views stored
  - User description table—users and their access levels
  - Privilege description table—database privileges and views for users

- SCQL dictionary is a view on the system table to access information
  - Provide distinct SCQL dictionaries for each user
  - View definition stored in data folders

## Summary

- Smartcards are emerging as a multipurpose technology

- Need for DBMS that will fit on a small, flexible card is increasing

- Limitations destroy foundations DBMS have been built on

- Must reexamine each component of DBMS to solve problem
Database Systems for Sensor Networks
Cem Goncu
December 2, 2004

Outline
- Sensor network overview
- Regular databases vs. sensornets
- Constraints
- Opportunities
- Model-based approach
- Comparative systems
- Snapshot queries
- Conclusion and suggestions for future work

Sensor Network Overview
- Tiny devices embedded in the physical world
- Battery powered microprocessors
- Combine sensing, computation and communication
- Monitor environment for interesting events
- Acquire and transmit data at specified intervals

Sensornets
- Distributed data acquisition with multiple sensors (nodes)
- Could consist of any number of nodes (N)
- For large N, no concern for reliability of a single sensor
- Wireless communication between nodes
- Requires position detection, fault tolerance, aggregation, etc.

Sensornet Applications
- Habitat/environmental monitoring
  - Temperature, light, humidity
  - Voltage, radiation
- Military surveillance & reconnaissance
- Traffic
  - Movement, velocity, acceleration
  - Vehicle tracking

Regular databases vs. Sensornets
- Regular DBMS process information about a stored collection of data (complete)
- Sensornets work with real-time information about the environment
- The set of relevant data is continuous both in time and space (infinite)!
- Impossible to gather all relevant data
- Acquire samples of physical phenomena at discreet points in time and space
- Provide approximate answers with a degree of uncertainty
Constraints

- Energy: limited power (batteries)
- Limited memory, processing capabilities
- Sensors unreliable
  - Random failures, come on/offline
- Position: (e.g. hostile environment)
  - self configuration
  - self-maintenance
  - Ad hoc deployment

Opportunities

- High correlation between measurements of different nodes in a neighborhood
  - If $d(n_1,n_2) < D$, expect $|T_1-T_2|$ to be small
  - Use $T_1$ to predict $T_2$
- Other types of correlation between measurements of different variables
  - E.g. correlation between temperature and voltage at a given sensor
  - Use $V_i$ to predict $T_i$
- Create a model that exploits these correlations in order to save time and energy!

Model-Driven Approach

- BBQ: a tiny model query system
- Incorporate statistical models of real-world processes into a sensornet query processing architecture
- Build a model that captures correlations between sensors
- Idea: using a subset of nodes is sufficient to gather information about the whole system

Query Processing

- Let queries specify error tolerances and target confidence bounds
  - E.g. $e = 0.1\,^\circ C$, $\delta = .05$
- Goal: given an error bound $e$, and a degree of confidence $1-\delta$, determine the subset of nodes to provide an approximate answer at the minimum cost
- Use the chosen subset of nodes to report all requested information
  - Within error bound $e$
  - With confidence $1-\delta$

Optimization

- Setting: We have a set $O$ of potential observations requested by a query
- Idea:
  - Each extra observation $o_i$ increases our confidence about the answer by $R(o_i)$
  - Each extra observation comes at a certain cost $C(o_i)$
- Goal: Choose the subset of observations $O_s$ that satisfies $R(O_s) > 1-\delta$ at the minimum cost $C(O_s)$

Optimization (cont.)

- Question:
  - Which nodes to use
  - Which attributes to observe
- Focus: select nodes and attributes that are expected to increase the confidence in the answer to the query, at a minimum cost
- Given a query and a model, choose an observation plan according to a cost/benefit analysis
Definition of Cost

- Let \( O = \{o_1, o_2, \ldots, o_n\} \) be a set of \( n \) observations
- \( C(O) = \sum C(o) \)
- The system cost of an observation is the sum of acquisition and transmission costs:
  - \( C(o) = C_a(o) + C_t(o) \)
  - \( C(O) = C_a(O) + C_t(O) \)

Data acquisition cost \( C_a \)

- Sum of energy required to observe attributes \( O \)
  - \( C_a(O) = \sum C_a(o) \)
- Observations of different variables require different amounts of energy per sample:
  - Voltage: 0.00009
  - Humidity and temperature: 0.5
  - Barometric pressure: 0.003
  - Solar radiation: 0.525

Data transmission cost \( C_t \)

- Communication cost required to download the data
- Expect transmission cost to be proportional to the number of nodes used: \( C_t = kN \)
- Depends on data collection mechanism used to collect observations from network (TinyDB, approximate caching)
- Depends on network topology
- If topology is unknown or changing, cost function is basically random
- Therefore, assume networks with known topologies

Definition of Benefit

- Let \( O = \{o_1, o_2, \ldots, o_n\} \) be a set of \( n \) observations
- \( R_i(o) \): benefit to the accuracy of a reading \( X_i \) given the set of observation values \( o \)
  - For value and average queries: \( X_i = x_i \)
    - \( R_i(o) = P(X_i \in [x_i-e, x_i+e] | o) \)
  - For range queries: \( X_i \in [a_i, b_i] \)
    - \( R_i(o) = \max\{P(X_i \in [a_i, b_i] | o), 1- P(X_i \in [a_i, b_i] | o)\} \)

Expected benefit

- Specific value \( o \) of \( O \) is not known a priori
- Must compute expected benefit \( R_i(O) \)
  - \( R(O) = \int p(o) R_i(o) \, do \)
- For a set of queried readings \( Q \) define the average benefit as
  - \( R(o) = 1/|Q| \sum_{i \in Q} R_i(o) \)
- Use average benefit to decide when to stop observing new attributes

Choosing an observation plan

- Problem: Given an error bound \( e \) and confidence level 1-d, pick the set of observations \( O_s \) from \( O \) to
  - Minimize \( C(O_s) \) such that \( R(O_s) \geq 1-d \)
- Solutions:
  - Option 1 - exhaustive search
  - Option 2 - greedy algorithm
Exhaustive search
- Exhaustively search over all possible subsets of possible observations, O
- Finds the optimal subset $O_s$ with minimum cost $C(O_s)$
- Exponential running time

Greedy algorithm
- Start with an empty set of observations, $O = \emptyset$
- For each observation $o_i$ that is not in our set $O$
  - Compute the new expected benefit $R(O \cup o_i)$ and expected cost $C(O \cup o_i)$
  - If a subset of observations $G$ reach the desired confidence such that $R(O \cup o_i) \geq 1-d$ for every $o_j \in G$
    - Pick $o_i$ with the lowest cost $C(o_i)$, and terminate search
  - Else if $G = \emptyset$, simply keep on adding $o_i$ with the highest benefit over cost ratio to the existing set $O$ until $R(O) \geq 1-d$

A simple example
- Query:
  ```sql
  SELECT node id, temp + - .1 ºC, conf( .95)
  WHERE node id in (1..8)
  ```
- Observation plan:
  - $\{[\text{Voltage},1], [\text{Voltage}, 2], [\text{temp}, 4]\}$
- Data: $\{[V1=2.73], [V2=2.65], [T4=22.1]\}$
- Results:
  - $\{[22.5, 97%], [25.6, 99%], [24.4, 98%], [22.1, 100%], \ldots\}$

Review of alternative approaches
1. TinyDB-style Querying
2. Approximate Caching

TinyDB
- Query disseminated into the sensor network using a tree structure
- At each mote, sensor reading is observed
- Results reported back along the same tree to the base station
- Combine results on the way back to minimize communication costs
TinyDB (continued)

- **Advantages:**
  - Answer all queries with 100% confidence as long as all nodes are functioning
  - Reduce communication costs by combining the results on the way back
- **Disadvantages:**
  - Brute force, have to query all the existing nodes at all times
  - Time & energy inefficiencies
  - Shorter lifespan for an average node (as it remains active all the time)

Approximate Caching

- **Base station maintains view of readings at all motes**
- **View is guaranteed to be within a certain interval of the actual sensor readings**
- If value of sensor falls outside this interval, motes are required to report new reading

Approximate Caching (cont.)

- **Advantages:**
  - Reduces communication costs by not reporting readings within the specified interval
- **Disadvantages:**
  - Acquisition costs remain high, as each mote is required to make observations at every time step

Experimental results

- **Goal:** measure the performance of BBQ on several real-world data sets
- Compare results to alternative systems such as TinyDB and Approximate Caching

Data Set: “Garden”

- One month trace of 83,000 readings
- 11 sensors in a redwood tree at UC Botanical Garden in Berkeley
- Sensors placed at four different altitudes
- Collected light, humidity, temperature and voltage readings once every five minutes
- Data split into training and test data sets
  - Model built on training set

BBQ - Methodology

- Build the model from the training data
  - Includes transitional model for each hour of the day
- Issue one query against the model per hour
- Compute a priori probabilities for each predicate being satisfied, add additional readings if confidence bounds not met
- At the end of the hour, compare predicted values from the model to observed values from the test data
- Update the model accordingly
**BBQ - Performance**

- Query: requires system to report temperatures at all motes to within specified error bound
  - Confidence 95%, with varying e
  - Different values of e lead to varying cost of observation C(O)

**Results**

- Varied e from between 0 and 1 degrees C
- The cost of BBQ falls rapidly as e increases
- The percentage of errors stays well below the specified confidence threshold of 5%

**Comparison**

- TinyDB:
  - Makes no mistakes
  - Cost remains constant for all e
- Approximate Caching:
  - Always reports values to within e
  - Makes no mistakes
  - Average observation error close to that of BBQ

**Comparison (cont.)**

- BBQ:
  - Succeeds to report observations within the given error bound at least 95% of the time
  - For reasonable values of epsilon, uses significantly less communication
  - More efficient use of time and energy

**BBQ - Cost Efficiency**

- Percentage of sensors that BBQ observes by hour
  - Varying e

- As e gets small (<0.1), must observe all nodes on every query
  - Variance between nodes high enough that it cannot infer value of one sensor from another’s with any accuracy
- As e gets large (>1), few observations are needed
  - Changes in one sensor predict values of others
- Intermediate e
  - More observations are needed, especially during times when readings change drastically
Decreasing confidence intervals or epsilon reduces energy per query

- Confidence 95%
- Errors 0.5
- Reduce expected energy cost from 5.4 J to 150 mJ per query
- Factor of 40 reduction

Another approach

- Snapshot Queries: (Kotidis)
- Data-driven approach in which a node can represent another node in a query when their collected measurements are similar
- Algorithm for nodes to elect a local representative
- Determine a threshold value $T$ such that $d(\text{actual}, \text{estimate}) \leq T$

Idea: expect a lot of correlations among the collected measurements of neighboring nodes
Goal: Use only a subset of nodes (a representative from each neighborhood) to create a “snapshot” of the whole system
Answer certain queries (snapshot queries) without using the other nodes to save time and energy
Reduction of up to 90% in the number of nodes that need to participate in a snapshot query

Local algorithm for picking up representatives:
- $N_i$ can represent $N_j$ if $d(x_j, x_{ij}) < T$
  - where $x_j$ is the actual reading of node $j$, and
  - $x_{ij}$ is $N_i$’s estimate of $x_j$
- The “snapshot” is not static, but changes over time:
  - $N_i$ may fail ($N_i$ requests a new representative)
  - Due to the dynamic nature of the environment, $d(x_j, x_{ij})$ might get bigger than the threshold value
  - Ideally, we would like to have a rotating set of representatives so that energy resources are drained uniformly (larger lifespan for an average node)

Snapshot vs. BBQ

- BBQ: a global model to capture dependencies assuming a relatively stable network topology
- Snapshot: capture localized correlations in highly dynamic networks
- Snapshot more successful in networks consisting of a large number of nodes $N>1000$?

Conclusions

- General idea: tolerate a certain amount of uncertainty in return for crucial time and energy savings
- Exploit spatiotemporal correlations among individual nodes to enable better estimates
- Use only a subset of nodes to gather information about the whole system
Database applications

- Sensor networks are actually dealing with similar constraints familiar to all DBMS
  - limited energy, storage, processing capabilities
  - and face the same expectations
  - maintenance, durability, consistence
- Due to the nature of sensornets (small, battery-powered devices) these constraints are more extreme than for regular DBMS
- Accordingly, solutions are more extreme as well:
  - Redesign storage
  - Redesign the whole data model

Extensions and future directions

- Goal: unifying probabilistic models with declarative queries
- Detect faulty sensors and outliers
  - Give correct answers to queries in the presence of faulty sensors
- Enable the model to work with dynamic network topologies
  - Current model assumes static topology
  - What if new sensors added, existing ones removed