Monitoring Queries (Event Detection) in Sensor Networks
(Adapted from the VLDB’05 and SIGMOD’07 slides by authors)

Sensor Networks
- A large network consisting of many small nodes
  - Motes
    - Small wireless computing devices
    - With sensing, networking and computation capabilities
    - Battery-powered
  - Constraints
    - Storage
    - Volatile Networking
    - Power / Energy

Metric: Communication
- Lifetime from one pair of AA batteries
  - 2-3 days at full power
  - 6 months at 2% duty cycle
- Communication dominates cost
  - ~ few mS to compute
  - 30mS to send message
- Key metric: communication!

Declarative Queries
- Users specify the data they want
  - Simple, SQL-like queries
  - Using predicates, not specific addresses
- Challenge is to provide:
  - Expressive & easy-to-use interface
  - High-level operators
    - Well-defined interactions
    - "Transparent Optimizations" that many programmers would miss
  - Sensor-net specific techniques
    - Power efficient execution framework
- Question: Do sensor networks change query processing?

Data Management
High level abstraction:
- Data centric programming
- Interact with sensor network as a whole
- Extensible framework
Under the hood:
- Intelligent query processing
- Fault Mitigation

Query Delivery
• App
  • Query, Trigger
  • Data

TinyDB
- Sensor Network

Tree-based Routing
- Used in:
  • Query delivery
  • Data collection
  • In-network aggregation
Basic Aggregation

- In each epoch:
  - Each node samples local sensors once
  - Generates partial state record (PSR)
    - local readings
    - readings from children
  - Outputs PSR during assigned comm. interval
    - Interval assigned based on depth in tree

At end of epoch, PSR for whole network output at root
New result on each successive epoch

Illustration: In-Network Aggregation

SELECT COUNT(*) FROM sensors

Sensor #
Interval #
Interval
Illustration: In-Network Aggregation

SELECT COUNT(*) FROM sensors

<table>
<thead>
<tr>
<th>Interval #</th>
<th>Sensor #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Example Filter Query

SELECT s.nodeid, a.condition_type
FROM sensors AS s, alert_table AS a
WHERE s.temp > a.temp_thresh
AND s.humidity > a.humidity
AND s.time = a.time
SAMPLE PERIOD 1s

<table>
<thead>
<tr>
<th>Condition type</th>
<th>Time</th>
<th>Temp</th>
<th>Humid</th>
<th>nodeid</th>
<th>Time</th>
<th>Temp</th>
<th>Humid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9 h</td>
<td>&gt; 73</td>
<td>&gt; 80</td>
<td>1054</td>
<td>9 h</td>
<td>70</td>
<td>50%</td>
</tr>
<tr>
<td>2</td>
<td>10 h</td>
<td>&gt; 75</td>
<td>&gt; 87</td>
<td>1055</td>
<td>10 h</td>
<td>76</td>
<td>75%</td>
</tr>
<tr>
<td>3</td>
<td>11 h</td>
<td>&gt; 80</td>
<td>&gt; 80</td>
<td>1056</td>
<td>11 h</td>
<td>85</td>
<td>85%</td>
</tr>
</tbody>
</table>

Alert_table (External table) Sensor (Virtual)

SELECT s.nodeid, a.condition_type
FROM sensors AS s, alert_table AS a
WHERE s.temp > a.temp_thresh
AND s.humidity > a.humidity
AND s.time = a.time
SAMPLE PERIOD 1s

Naïve Join Algorithm

- Send all tuples from data table to root; perform join at root
- Communication overhead is worst if sampling rate is high

Ideal Join Algorithm

- Send join table to each node
- At node, perform join
- Problem: Severe Node Memory Constraints
- Optimization: Only certain intermediate nodes store the table
- Good for small tables

REED Algorithm

- Cluster nodes into groups
- Store portion of predicate table in each group member
- Good for intermediate size predicate table
- Send sensor data tuples to every member of group
- A cost model is needed.
- Can one node belongs to multiple groups?
REED (Group Formation)

A Group is a set of nodes where every node is in broadcast range of every other node. The cumulative storage should be large enough to hold the predicate table.

Process:
1. Every node maintains a list of nodes it can hear by listening in on packets.
2. After a random interval, a node P which is not in a group broadcasts a request to form a group.
3. Every node N which hears that request and is not currently in a group replies to P with a list of neighbors and amount of free space.
4. Node P collects the replies, and determines who should be in the group. For every node N which replied, P sends either a group reject or a group accept message.
5. Group accept message contains a list of nodes in the group.

Group Formation

A Group is a set of nodes where every node is in broadcast range of every other node. The cumulative storage should be large enough to hold the predicate table.

A Group is a set of nodes where every node is in broadcast range of every other node. The cumulative storage should be large enough to hold the predicate table.

REED (Join Table Distribution)

Group members figure out amongst themselves how the table will be divided across group.

Process:
1. When a node enters a group, it sends a request to the root for join table data.
2. Per group, the root gives out non-overlapping segments of the join table to every member.
3. Once all the members in a group have received join tuples, they begin processing data tuples as a group.

REED (Operation)

For nodes not in group:
1. When generating a data tuple or receiving data tuple from child, pass on to parent.
2. When receiving a result from child, pass on to parent.

For nodes in group:
1. When generating a data tuple or receiving data tuple from child, broadcast to group (including self).
2. Upon receiving data tuple broadcast from group, join with stored subset of join table and pass result up to parent.
3. When receiving a result from child, pass on to parent.

Bloom Filter Optimization (Good for Large Predicate Tables)

Step 1: Hash domain of sensor values onto Bloom Filter
Step 2: Send Bloom Filter to Each Sensor Node

Might produce false positives but never false negatives
Can be used in conjunction with previous REED algorithm

Cache Diffusion (Used for Large Table)

• Cache non-joining ranges on a per node basis
• Also will produce false positives but no false negatives
• Challenge: What range to be maintained at which node?
• Good when many sensor tuples will match
Motivation II: Volcano Monitoring (Paper II)

- **Continuous Monitoring Queries**
  - Scientists are interested in the pressures detected around the volcanic mountain
    - whether the pressures detected have crossed over a certain threshold and are continuously increasing?
  - Such questions cannot be answered by aggregate or selection query

Observations

- **Self-join** - the join condition is posed on one column of the same table
- **Window predicate** - sliding window
- **Selection predicate** - indicating possible starts of events of interest

Execution Semantics (cont.)

- Evaluate $Q^*$ continuously
  - If a tuple $T$ is found where $p_i(T)$ is true, a window is defined starting from $t_s$ with size $W$
  - If there are multiple tuples with equal timestamps satisfying $p_i$, only one window is defined
  - For tuples that have different timestamps, a new window is defined

Declarative Query

```
SELECT P1.pressure, P1.time, P2.pressure, P2.time
FROM Pressure AS P1, Pressure AS P2
WHERE P1.pressure > P2.pressure
AND P1.time < P2.time
AND P1.time - P2.time < h
```

- **Direct Query Evaluation Methods**
  - Send-to-sink
  - Flooding

Execution Semantics

- **Send-to-sink** - centralized approach
  - At each sampling interval, each sensor node $N_i$ sends to the base station its readings as a tuple $T_i = \langle att_1, att_2, ..., att_n, ts_i \rangle$
  - $att_i$'s - sensor readings for multiple attributes
  - $N_i$ - sensor ID and $ts_i$ - timestamp
  - Tuples are stored at the base station in the Sensor relation
- **Consider Queries of form $Q^*$**
  - Evaluate $Q^*$ over relation Sensor
    - $att_i$ - subset of attributes from Sensor
    - $w$ - size of the sliding window

Query Preprocessing

- Given a user query in the form of $Q^*$, it is rewritten into two queries $Q^1$ and $Q^2$
  - $Q^1$, detects interesting events, and $Q^2$, finds important correlations between readings after events are detected

```
Q^1:
SELECT S1.att, S1.att
FROM Sensor AS S1
WHERE \( p(S1.att) \)
AND window(S1.ts, S1.ts, W)
```

```
Q^2:
SELECT S2.att
FROM Sensor AS S2
WHERE \( p(S2.att) \)
AND window(S2.ts, S2.ts, W)
```
Example – Query Preprocessing

• Scientists want to monitor continuously increasing high pressure larger than $\delta$ within period of $h$

\[
\text{SELECT } P_1\cdot\text{pressure}, P_1\cdot\text{time}, P_2\cdot\text{pressure}, P_2\cdot\text{time} \\
\text{FROM Pressure AS } P_1, \text{Pressure AS } P_2 \\
\text{WHERE } P_1\cdot\text{pressure} > \delta \\
\text{AND } P_2\cdot\text{pressure} > P_1\cdot\text{pressure} \\
\text{AND window}(P_1\cdot\text{time}, P_2\cdot\text{time}, h)
\]

Two-Phase Self-Join Processing

• Phase 2
  – Evaluate query $Q^*_1$
  – Each node performs a join between local sensor data and results of query $Q^*_1$, i.e., table $R_1$.

• Phase 2 - Naïve Solution
  – Compile relation $R_1$ as in TPSJ and start a new window self-join for each $R_1$ (Evaluate $Q^*_2$ for each window)

Continuous TPSJ (cont.)

• Phase 1
  – Run $Q^*_1$ continuously
  – Tuples satisfying $p_1$ are continuously sent to the base station

• Phase 2 - Naïve Solution
  – Compile relation $R_1$ as in TPSJ and start a new window self-join for each $R_1$ (Evaluate $Q^*_2$ for each window)
Continuous TSPJ

Rule A – One Window Self-Join Per Sampling Interval

- At most one window self-join is triggered for one time clock
  - For all the tuples that have identical timestamps, Phase 2 is triggered only once
  - The same as in TPSJ for one window execution

Rule B – Delay New Window Triggering As Much As Possible

1. If \( R_1 \subset R_1' \)
   a. If the join predicate is \( att_j \neq att_h \)
      - Part of \( Q' \)'s answer is included in \( Q \)'s answer
      - Delay executing \( Q' \)
      - Modify \( W' = [t_s', t_e'] \) to \( [t_e', t_e' + 1] \)

2. Similar rules can be defined if \( R_1' \subset R_1 \)

Rule B – Delay New Window Triggering As Much As Possible

3. If the join predicate \( att_j \neq att_h \)
   - \( R_1' \) and \( R_1 \) are of size 1
   - Assume \( R_1(a) \) and \( R_1'(b) \)
   a. If \( b < a \)
      i. If \( op \in \{<, \leq\} \), then \( Q' \)'s answer during overlapping window is completely included in \( Q \)'s answer; so \( Q' \) is injected into the network and processed with the join predicate replaced by \( b \) op \( att_h \) and \( Q \) is stopped.
Rule B – Delay New Window Triggering As Much As Possible

3. If the join predicate $\text{att}_j \text{op} \text{att}_h \in \{<, >, \leq, \geq\}$:
   - Assume $R_1 = \{a\}$ and $R_1' = \{b\}$.
   a. If $b < a$:
      i) If $\text{op} \in \{>, \geq\}$, then part of Q's answer is included in Q's answer; Q is then injected with predicate $b \text{op} \text{att}_h$ and the window adjusted to $W' - W$.
   b. Similar for $b > a$.

4. If $R_1 = R_1'$, delay Q' (since Q and Q' have exactly the same join results during the overlapping window). Why delay Q'??