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

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
```
Illustration: In-Network Aggregation

```
SELECT COUNT(*)
FROM sensors
```

Interval 3

Sensor #

Illustration: In-Network Aggregation

```
SELECT COUNT(*)
FROM sensors
```

Interval 2

Sensor #
Illustration: In-Network Aggregation

SELECT COUNT(*)
FROM sensors

Interval 1

Sensor #

Interval #

Illustration: In-Network Aggregation

SELECT COUNT(*)
FROM sensors

Interval 4

Sensor #

Interval #
### Illustration: In-Network Aggregation

```
SELECT COUNT(*)
FROM sensors
```

### Motivation I (Paper I)

```
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_thresh
AND s.time = a.time
SAMPLE PERIOD 1s
```
Example Filter Query

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Temp</th>
<th>MinTS</th>
<th>MaxTS</th>
<th>MinTemp</th>
<th>MaxTemp</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:00PM</td>
<td>70</td>
<td>2:30PM</td>
<td>75</td>
<td>70</td>
<td>75</td>
</tr>
<tr>
<td>2:30PM</td>
<td>73</td>
<td>3:00PM</td>
<td>78</td>
<td>73</td>
<td>78</td>
</tr>
<tr>
<td>3:00PM</td>
<td>75</td>
<td>3:30PM</td>
<td>80</td>
<td>75</td>
<td>80</td>
</tr>
<tr>
<td>3:30PM</td>
<td>83</td>
<td>4:00PM</td>
<td>88</td>
<td>83</td>
<td>88</td>
</tr>
<tr>
<td>4:00PM</td>
<td>85</td>
<td>4:30PM</td>
<td>90</td>
<td>85</td>
<td>90</td>
</tr>
<tr>
<td>4:30PM</td>
<td>70</td>
<td>5:00PM</td>
<td>75</td>
<td>70</td>
<td>75</td>
</tr>
<tr>
<td>5:00PM</td>
<td>72</td>
<td>5:30PM</td>
<td>77</td>
<td>72</td>
<td>77</td>
</tr>
<tr>
<td>5:30PM</td>
<td>75</td>
<td>6:00PM</td>
<td>80</td>
<td>75</td>
<td>80</td>
</tr>
</tbody>
</table>

Join Predicate:

\[ TS > \text{MinTS} \land TS < \text{MaxTS} \land (\text{Temp} < \text{MinTemp} \lor \text{Temp} > \text{MaxTemp}) \]

Ideally, selectivity should be low.

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 belong 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 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 form group request.
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 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**

Neighbor list: \(\{1, 2, 3, 4, 6\}\)

Choose Me! \(\{1, 3, 4\}\)

Space: 2

Space: 4

Potential: \(\{1, 2, 3, 4, 6\}\)

Neighbor list: \(\{1, 3, 4\}\)

Neighbor list: \(\{1, 3, 4, 6\}\)

Neighbor list: \(\{1, 4, 6\}\)

Neighbor list: \(\{1, 4, 6\}\)
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 nodes 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

- Temp: 20
- Temp: 90

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

Declarative Query

```
SELECT P_1.pressure, P_1.time, P_2.pressure, P_2.time
FROM Pressure AS P_1, Pressure AS P_2
WHERE P_1.pressure > δ
AND P_2.pressure > P_1.pressure
AND P_1.time < P_2.time
AND P_2.time - P_1.time < h
```

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

SELECT $P_1$.pressure, $P_1$.time, $P_2$.pressure, $P_2$.time
FROM Pressure AS $P_1$, Pressure AS $P_2$
WHERE $P_1$.pressure > $\delta$
AND $P_2$.pressure > $P_1$.pressure
AND window ($P_1$.time, $P_2$.time, $h$)

• 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

• 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_j = <att_1, att_2,...,att_n, N_i, ts_j>$$
  • $att_i$’s - sensor readings for multiple attributes
  • $N_i$ - sensor ID and $ts_j$ - timestamp
  – Tuples are stored at the base station in the Sensor relation
• Consider Queries of form $Q^*$
  – Evaluate $Q^*$ over relation Sensor
    • $AT_1$ - subset of attributes from Sensor
    • $W$ - size of the sliding window
    $$Q^*:
    SELECT $S_1$.AT_1, $S_2$.AT_2
    FROM Sensor AS $S_1$, Sensor AS $S_2$
    WHERE $p_1(S_1$.AT_1$)$
    AND $p_2(S_1$.att, $S_2$.att$)$
    AND window($S_1$.ts,$S_2$.ts,$W$)
Execution Semantics (cont.)

• Evaluate \( Q^* \) continuously
  – If a tuple \( T_j \) is found where \( p_1(T_j) \) is true, a window is defined starting from \( ts_j \) with size \( W \)
  – If there are multiple tuples with equal timestamps satisfying \( p_1 \), only one window is defined
  – For tuples that have different timestamps, a new window is defined

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

\[
Q^*:
SELECT S_1.AT_1, S_2.AT_2
FROM Sensor AS S_1, Sensor AS S_2
WHERE p_1(S_1.AT_1)
AND p_2(S_1.att_j, S_2.att_h)
AND window (S_1.ts, S_2.ts, W)
\]

\[
Q^*_1:
SELECT S_1.AT_1 INTO R_1
FROM Sensor AS S
WHERE p_1(S_1.AT_1)
\]

\[
Q^*_2:
SELECT S_2.AT_2
FROM Sensor AS S
WHERE p_2(R_1.att_j, S.att_h)
AND window (R_1.ts, S.ts, W)
\]
Example – Query Preprocessing

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

```
SELECT $P_1$.pressure, $P_1$.time, $P_2$.pressure, $P_2$.time
FROM Pressure AS $P_1$, Pressure AS $P_2$
WHERE $P_1$.pressure > $\delta$
AND $P_2$.pressure > $P_1$.pressure
AND window ($P_1$.time, $P_2$.time, $h$)
```

Two-Phase Self-Join Processing
(Single Execution)

• Phase 1
  – Evaluate query $Q^*_1$
  – Store the results of query $Q^*_1$ in table $R_1$
  – $R_1$ will be redistributed into the network in Phase 2
  – $R_1$ is used to filter local data at each sensor node

```
SELECT P.pressure, P.time
INTO R_1
FROM Pressure AS P
WHERE P.pressure > $\delta$
```

```
SELECT P.pressure, P.time
INTO R_1
FROM Pressure AS P
WHERE P.pressure > 500
```
Two-Phase Self-Join Processing

- Phase 2
  - Evaluate query \( Q^* \)
  - Each node performs a join between local sensor data and results of query \( Q^* \), i.e., table \( R_1 \)

- NOTE: We are dealing single window here. Some tuples actually satisfy \( Q^* \), e.g., node 3’s \( (520, 5) \), and node 5’s \( (550, 9) \)

```sql
SELECT P.pressure, P.time
FROM R_1, Pressure AS P
WHERE P.pressure > R_1.pressure
AND window(P.time, R_1.time, 10)
```

Result Table Dissemination

- How to reduce the size of the result table which need to be disseminated into the network
  - When the join predicate involves operator of \( \{<, >, \leq, \geq\} \), we can sort \( R_1 \) and choose to send the smallest or the largest value into the network
    - Join predicate becomes a selection predicate
  - When the join predicate involves operator of \( \{=, \neq\} \), we can apply some encoding technique, e.g., use bitmap to represent a result table
Continuous TPSJ

• There are tuples that may trigger new windows

• Trigger new window self-join at the base station carefully
  – To avoid unnecessary work within overlapping windows

Continuous TPSJ (cont.)

• Phase 1
  – Run $Q^*_1$ continuously
  – Tuples satisfying $p_i$ 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

\[ R_1 = \{ S_{2,T}, S_{4,T}, S_{5,T}, \ldots \} \]

\[ Q^*_T: \quad W \quad t \quad t_{sj} \]
Rule B – Delay New Window Triggering As Much As Possible

<table>
<thead>
<tr>
<th></th>
<th>The phase 2 query currently being executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q'$</td>
<td>The subsequent execution of the phase 2 query</td>
</tr>
<tr>
<td>$R_1$</td>
<td>The result table of phase 1 used in $Q$</td>
</tr>
<tr>
<td>$R'_1$</td>
<td>The result table of phase 1 used in $Q'$</td>
</tr>
</tbody>
</table>

- Join predicate $p_2$ is of form $R_1.\text{att}_j \text{ op } S.\text{att}_h$
- Window $W$ is described as $[t_s, t_e]$ where $t_s / t_e$ is the start / end time.

1. If $R_1 \subset R'_1$
   a. If the join predicate is $\text{att}_j = \text{att}_h$,
      - $Q$’s answer during overlapping window is completely included in $Q'$’s answer
      - Inject $Q'$ and $R'_1$ into the network and start executing $Q'$ instead of $Q$
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 $W' = [t_e, t_e]$ to $[t_e+1, t_e]$
      
      Assume $R_1 = \{5, 9\}$, $R_1' = \{2, 5, 9\}$
      the join predicate is: $att_j \neq att_h$
      - Apply rule B1.b

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

Rule B

3. If the join predicate $att_j \text{ op } att_h$, $\text{ op } \in \{<, >, \leq, \geq\}$,
   - Result tables $R_1$ and $R_1'$ are of size 1
   - Assume $R_1 = \{a\}$ and $R_1' = \{b\}$.
   a. If $b < a$
      i. If $\text{ op } \in \{<, \leq\}$, 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 $b \text{ op } att_h$ and $Q$ is stopped.
Rule B – Delay New Window Triggering As Much As Possible

3. If the join predicate \( att \_op \_att \), \( op \in \{<, >, \leq, \geq \} \),
   - Result tables \( R_i \) and \( R_i' \) are of size 1
   - Assume \( R_i = \{a\} \) and \( R_i' = \{b\} \),
   a. If \( b < a \)
      i) If \( op \in \{>, \geq\} \), then part of \( Q' \)'s answer is included in \( Q \)'s
         answer; \( Q' \) is delayed until end of \( Q \)'s execution window; \( Q' \) is
         then injected with predicate \( b \_op \_att \) and the window adjusted
to \( W' - W \).
   b. Similar for \( b > a \)

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

Rule C – Hidden Query

- When a query \( Q' \) is delayed by some query \( Q \).
  - It is possible that, while \( Q' \) is waiting to be issued, a
    new query \( Q'' \) is triggered that makes issuing \( Q' \)
    unnecessary
  - We call \( Q' \) a hidden query

Assume \( R_i = \{60\}, R_i' = \{70\}, R_i'' = \{40\} \)
and the join predicate is \( att \_op \_att \),
- \( Q' \) is a hidden query and need not be
  executed.
Rule C – Hidden Queries

• Suppose Q’ (due to R’) is waiting, and Q” (due to R”’) is initiated
  – If R’ ⊆ R” AND join predicate is \( att_j = att_h \), then Q”’s results during W’ – W is completely covered by the result of Q”. Thus, result of Q’ is covered by Q ∪ Q” and Q’ can be dropped
  – Other cases can be similarly derived.

Example

• Join predicate: \( R_1.att_j < S.att_h \)
• Consider a series of new queries \( \{q_1, q_2, q_3, q_4, q_5, q_6\} \),
  \( q_j: <v, t_s, t_e>, R_j = \{v\}, W = [t_s, t_e], |W| = 10 \)
  \( q_1: <50, 5, 15> \)
  \( q_2: <60, 7, 17> \)
  \( q_3: <40, 8, 18> \)
  \( q_4: <70, 10, 20> \)
  \( q_5: <80, 12, 22> \)
  \( q_6: <60, 14, 24> \)

Only 3 queries

Active Query: \( q_1, q_3, q_6 \)
Delayed Query: \( q_2, q_4, q_5 \)
Hidden Query: \( q_2, q_4, q_5 \)
Conclusions

• Event-driven queries are common in sensor networks
• REED: designed for joining sensor data against a static dataset
• TPSJ: processes continuous self-joins in-network
• Experimental results show that both can significantly reduce data transmission in most cases