In-Network Execution of Monitoring Queries in Sensor Networks

Xiaoyan Yang, Hock-Beng Lim, M. Tamer Ozsu, Kian-Lee Tan

Presented by: Herodotos Herodotou
March 25, 2008

Outline

- Problem Setting
- Challenges
- Query Semantics
- Naïve Approaches
- Two-Phase Self-Join Approach
- Continuous TPSJ
- Experiments
- Discussion

Problem Setting

- Sensor Networks
  - Consist of many small sensor nodes with sensing, data processing and wireless communication capabilities
- Monitoring Queries
  - Track correlation among sensor data within a time window to detect events of interests
- In-network processing of window self-join queries

Challenges

- Sensor nodes are resource-constraint
  - Limited battery energy
  - Limited processing power
  - Limited storage
  - Limited communication bandwidth
- Ability to handle node failures
- Ability to handle communication failures

Query Semantics

Query Template:

SELECT $S_1.AT_1$, $S_2.AT_2$
FROM Sensor AS S_1, *Sensor AS* S_2 ⟵ Same Table (self-join)
WHERE $p_1(S_1.AT_3)$ ⟵ Selection Predicate
AND $p_2(S_1.att_j, S_2.att_h)$ ⟵ Join Predicate
AND $window(S_1.ts, S_2.ts, W)$ ⟵ Time window of size W

Notes:

- *Sensor* relation is sorted by the reading's timestamp
- AT_j represents subset of attributes from *Sensor* table
- p_1 is of the form att_j *opt Constant*
- p_2 is of the form att_j *opt* att_h
- *Opt* = { <, >, ≤, ≥, ≠, = }

Query Example

Volcano Monitoring Application:

SELECT $P_1.pressure, P_1.time, P2.pressure, P_2.time$
FROM *Pressure AS* P_1, *Pressure AS* P_2
WHERE $P_1.pressure > \delta$
AND $P_2.pressure > P_1.pressure$
AND $P_2.time > P_1.time$ *AND* $P_2.time - P_1.time < h$

Query Explanation:

Learn whether the pressure detected has crossed a certain threshold, δ, and is continuously increasing within some period of time, h.

Naïve Approach 1: Flooding

- Approach:
  - Propagate messages containing local sensor data to neighboring nodes
- Pros:
  - In-network query execution
- Cons:
  - Overwhelm limited network bandwidth
  - Need large amount of local storage
  - Incur a lot of computational power and memory

Naïve Approach 2: Centralization

- Approach:
  - All nodes periodically sense and transmit all data back to base station
- Pros:
  - Base station is more powerful
    => can handle self-join queries
- Cons:
  - High data transmission cost
  - Base station could still become bottleneck

Proposed Approach: TPSJ

- TPSJ: Two-Phase Self-Join
  - Phase 1:
    - Preliminary filtering
    - Find candidates that might be in final result
  - Phase 2:
    - Perform window join
    - Candidates found in phase 1 are used to do further filtering within the network

Preprocessing

- Query decomposition into 2 queries:

Q_1^*:

SELECT $S.AT_1$ *INTO* R_1
FROM *Sensor AS S*
WHERE $p_1(S.AT_3)$

Q_2^*:

SELECT $S.AT_2$
FROM R_1, *Sensor AS S*
WHERE $p_2(R_1.att_j, S.att_h)$
AND $window(R_1.ts, S.ts, W)$

Selection query:
- Finds tuples that satisfy selection predicate
- Stores results in temporary relation R_1

Join query:
- Between *Sensor* and intermediate table R_1

Preprocessing Example

- Volcano Monitoring Application:

Q_1^*:

SELECT P.pressure, P.time
INTO R_1
FROM Pressure AS P
WHERE P.pressure > δ

Q_2^*:

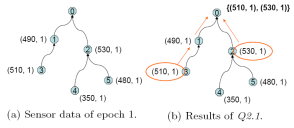
SELECT P.pressure, P.time
FROM R_1, *Pressure AS P*
WHERE P.pressure > R_1*.pressure*
AND $window(R_1.time, P.time, h)$

Selection query:
- Finds tuples twith pressure higher than a certain threshold, δ
- Stores results in temporary relation R_1

Join query:
- Between *Pressure* and intermediate table R_1

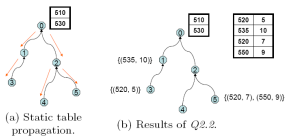
TPSJ: Phase 1

- **Goal**
  - Find candidates that might contribute to final result
- **Tasks**
  - Execute query Q_1^*
    - If tuple satisfies p_1, forward it to base station
- **Example**
  - $\delta = 500$



TPSJ: Phase 2

- **Goal**
  - Find matching tuples for candidates from Ph.1
  - Complete event detection in one time window
- **Tasks**
  - Construct R_1
  - Execute query Q_2^*
- **Example:**
  - $h = 10$



(a) Static table propagation. (b) Results of $Q2.2$.

Basic Optimizations

- Join predicate only involves $\{ <, >, \leq, \geq \}$
  - Sort R_1
  - Send smallest/largest value instead of R_1
- *Rewrite Q_2^* to a simple selection query*

SELECT $S.AT_2$
FROM *Sensor AS S*
WHERE $p_2(r.att_j, S.att_h)$
AND *window(W)*

Continuous TPSJ

- **Goal:**
  - Detect target events over a long period of time
- **Challenges:**
  - Some tuples may trigger a new window self-join within current processing window
  - Avoid unnecessary/dublicate work
- **Solution:**
  - New algorithm for efficient window triggering

Triggering of window self-joins

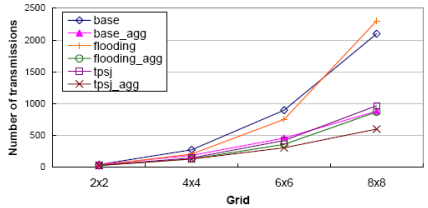
- **Rule A:**
  - One window self-join per sampling interval
- **Rule B:**
  - Delay Query Triggering As Much As Possible
- **Rule C:**
  - Hidden Query

⇩

- **Theorem:**
  - Number of phase 2 queries injected is minimal for self-join query involving only $\{ <, >, \leq, \geq \}$

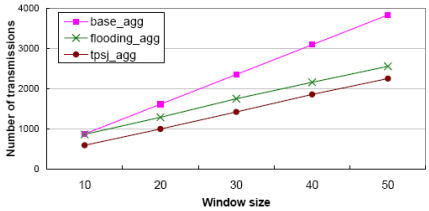
Experiments for TPSJ (1 window)

- Network Topology
  - Total Transmissions



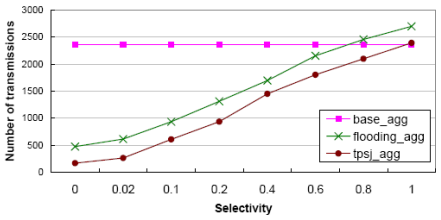
Experiments for TPSJ (1 window)

- Window Size
  - Total Transmissions



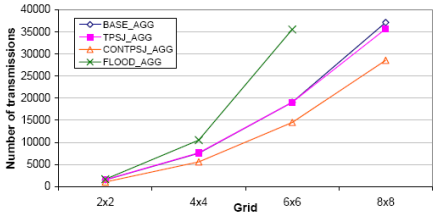
Experiments for TPSJ (1 window)

- Self-Join Selectivity
  - Total Transmissions



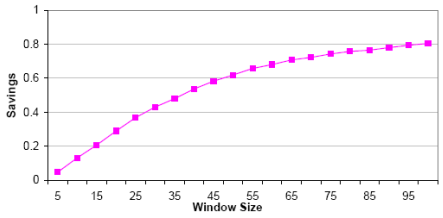
Experiments for Continuous TPSJ

- Network Topology
  - Total Transmissions



Experiments for Continuous TPSJ

- Window Size
  - Savings



Discussion

- Unrealistic Assumptions
  - Node Synchronization
  - Same sampling rate
  - Same transmission time from node to base station (no network delay, no distance issues)
- Scalability - Large number of tuples satisfying the selection predicate =>
  - How to disseminate efficiently?
  - How to store a number of queries in a memory-limited sensor node?
- Redundant tuple transfer from nodes to base station and back
- Insufficient optimization for queries with (in) equality constraints
- No Recursive Generalization for Phase 2 Optimizations
- Experimental Procedure: 8*8 nodes is very limited