Monitoring Queries (Event Detection) in Sensor Networks

(Adapted from the VLDB'05 and SIGMOD'07 slides by authors)

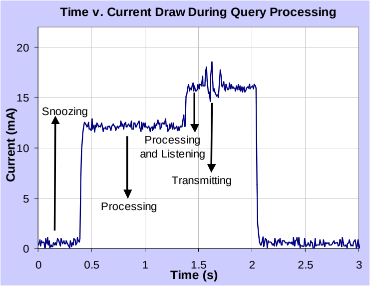
Sensor Networks

- 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

Berkeley MICA2 mote:
7 MHz processor
4 KB RAM & 512 KB flash
A 38.6 Kbps radio with
30 meters range
Runs on TinyOS
Powered by AA batteries.

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

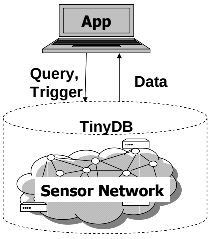
Extensible framework

Under the hood:

Intelligent query processing

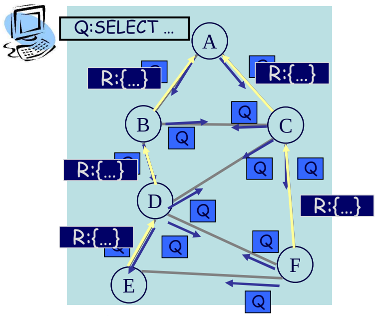
Fault Mitigation

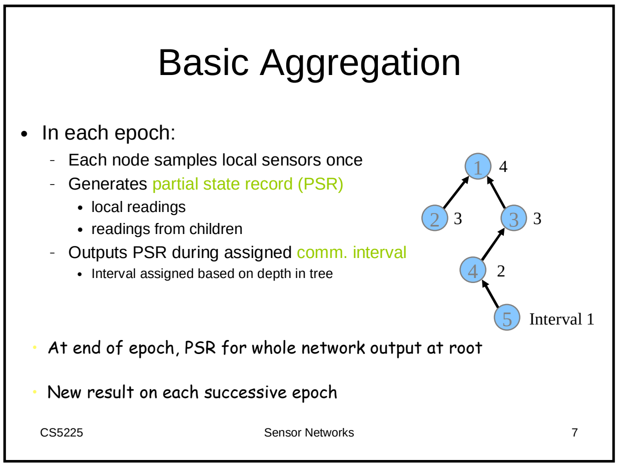
```
SELECT MAX(mag)
FROM sensors
WHERE mag > thresh
SAMPLE PERIOD 64ms
```



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

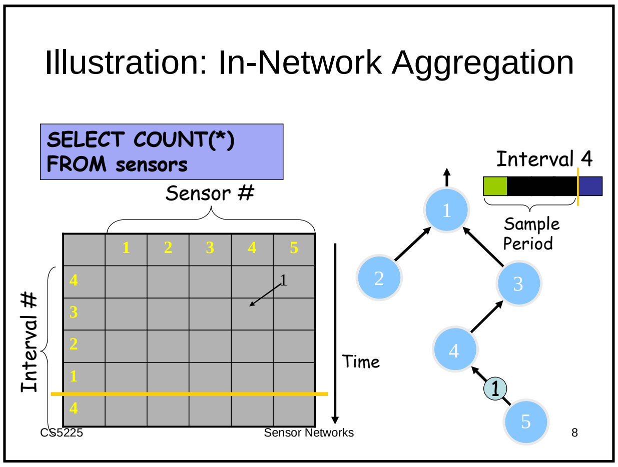


Illustration: In-Network Aggregation

SELECT COUNT(*)
FROM sensors

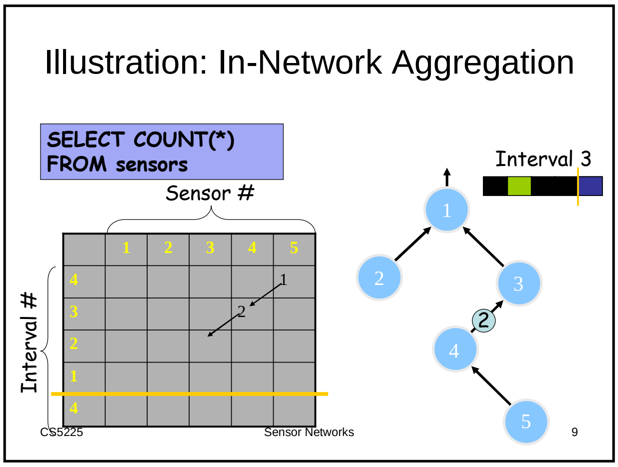


Illustration: In-Network Aggregation

SELECT COUNT(*)
FROM sensors

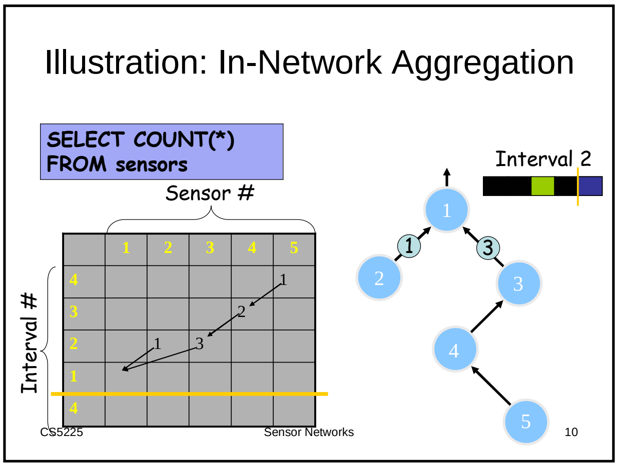


Illustration: In-Network Aggregation

SELECT COUNT(*)
FROM sensors

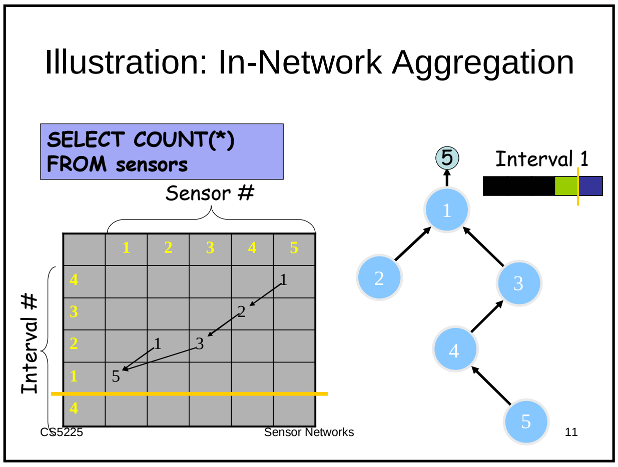


Illustration: In-Network Aggregation

SELECT COUNT(*)
FROM sensors

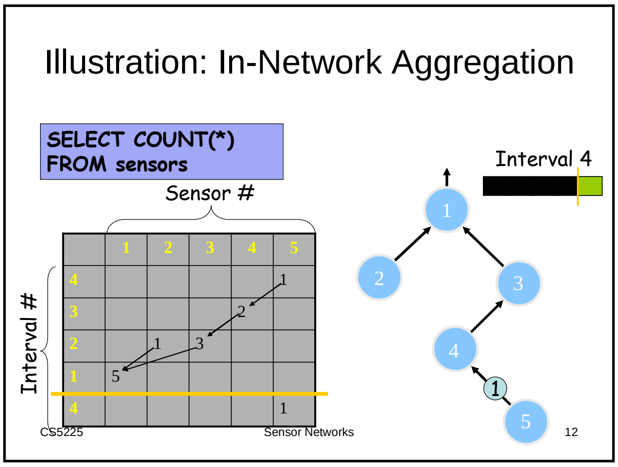
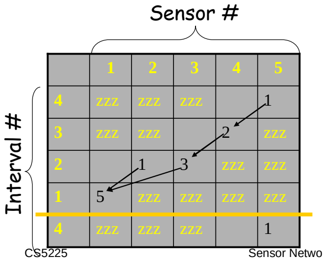


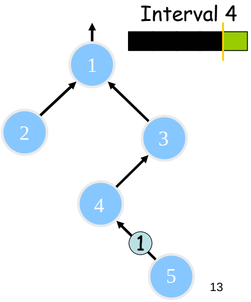
Illustration: In-Network Aggregation

SELECT COUNT(*)
FROM sensors

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.humid_thresh
AND s.time = a.time
SAMPLE PERIOD 1s
```

Condition _type	Time	Temp _thresh	Humid _thresh
1	9 h	> 73	> 60%
2	10 h	> 75	> 67%
3	11 h	> 80	> 80%
…	…	…	…

Alert_table (External table)

nodeid	Time	temp	humidity
1054	9 h	70	50%
1055	10 h	76	70%
1056	11 h	85	90%
…	…	…	…

Sensor (Virtual)

Example Filter Query

Timestamp	Temp
3:05PM	74

Sensor Data

⋈

MinTS	MaxTS	MinTemp	MaxTemp
2:00PM	2:30PM	70	75
2:30PM	3:00PM	73	78
3:00PM	3:30PM	75	80
3:30PM	4:00PM	83	88
4:00PM	4:30PM	85	90
4:30PM	5:00PM	70	75
5:00PM	5:30PM	72	77
5:30PM	6:00PM	75	80

Predicate Table

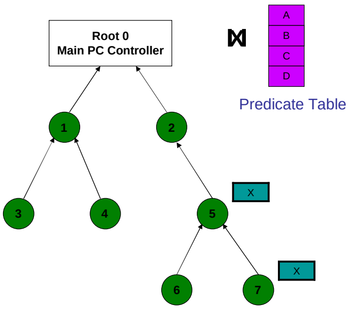
Join Predicate:

TS > MinTS && TS < MaxTS && (Temp < MinTemp || Temp > MaxTemp)

Ideally, selectivity should be low.

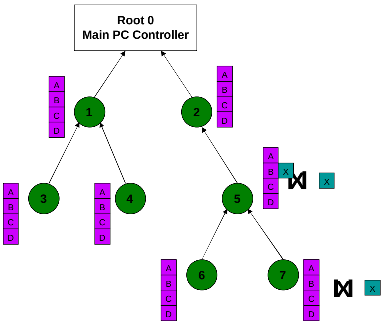
Naïve Join Algorithm

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



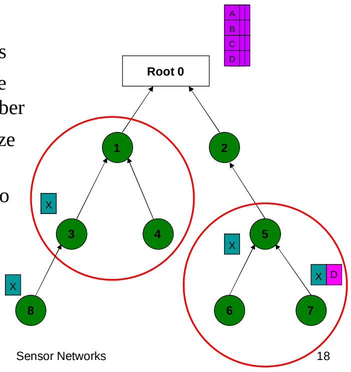
Ideal Join Algorithm

- 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 belongs to multiple groups?

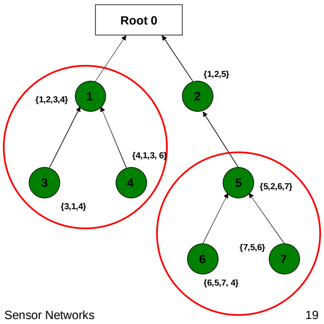


REED (Group Formation)

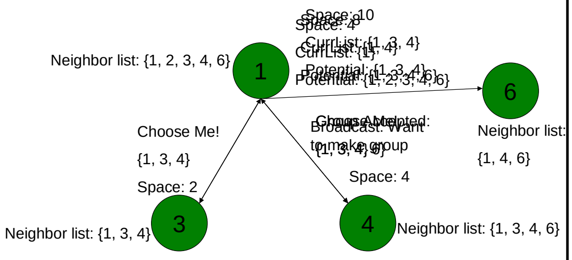
A Group is a set of nodes where **every node is in broadcast range of every other node**. The **cummulative storage should be large enough** to hold the predicate table

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}

Space: 10

Space: 8

Space: 4

CurrList: {1, 3, 4}

CurrList: {1, 4}

CurrList: {1}

Potential: {1, 3, 4}

Potential: {1, 3, 4, 6}

Potential: {1, 2, 3, 4, 6}

Choose Me!

{1, 3, 4}

Space: 2

Group Accepted:

{1, 3, 4, 6}

Broadcast: Want to make group

Space: 4

Neighbor list:

{1, 4, 6}

Neighbor list: {1, 3, 4}

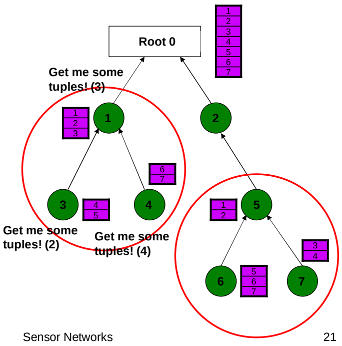
Neighbor list: {1, 3, 4, 6}

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



Get me some tuples! (3)

Get me some tuples! (2)

Get me some tuples! (4)

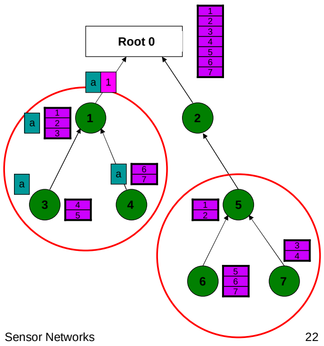
REED (Operation)

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:

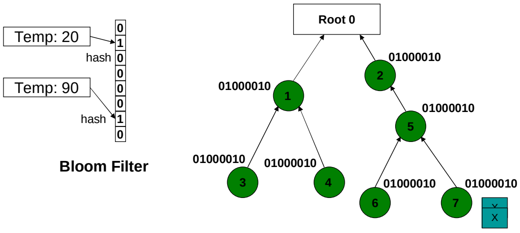
1. When generating a data tuple or receiving data tuple from child, broadcast to group (including self).
2. Upon receiving data tuple broadcast from group, join with stored subset of join table and pass result up to parent.
3. When receiving a result from child, pass on to parent.



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

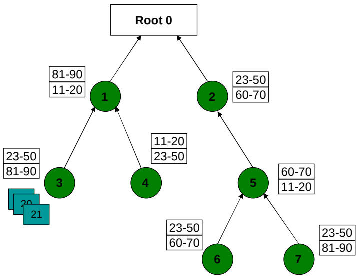


Bloom Filter

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

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

Observations

```
SELECT P1.pressure, P1.time, P2.pressure, P2.time
FROM Pressure AS P1, Pressure AS P2
WHERE P1.pressure > δ
AND P2.pressure > P1.pressure
AND window (P1.time, P2.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

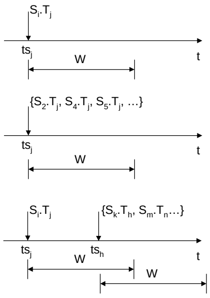
Execution Semantics

- *Send-to-sink* - centralized approach
  - At each sampling interval, each sensor node N_i sends to the base station its readings as a tuple $T_j = <att^i_1, att^i_2, \ldots, att^i_n, N_i, ts_j>$
    - att^i_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_i*** - subset of attributes from *Sensor*
    - ***W*** - size of the sliding window

```
Q*:
SELECT S1.AT1, S2.AT2
FROM Sensor AS S1,
     Sensor AS S2
WHERE p1(S1.AT3)
AND p2(S1.attj, S2.atth)
AND window(S1.ts,S2.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 S1.AT1, S2.AT2
FROM Sensor AS S1, Sensor AS S2
WHERE p1(S1.AT3)
AND p2(S1.attj, S2.atth)
AND window (S1.ts, S2.ts, W)
```

```
Q*1:
SELECT S1.AT1 INTO R1
FROM Sensor AS S
WHERE p1(S.AT3)
```

```
Q*2:
SELECT S.AT2
FROM R1, Sensor AS S
WHERE p2(R1.attj, S.atth)
AND window (R1.ts, S.ts, W)
```

Example – Query Preprocessing

- Scientists want to monitor continuously increasing high pressure larger than δ within period of *h*

```
SELECT P1.pressure, P1.time, P2.pressure, P2.time
FROM Pressure AS P1, Pressure AS P2
WHERE P1.pressure > δ
AND P2.pressure > P1.pressure
AND window (P1.time, P2.time, h)
```

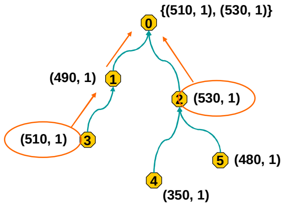
```
SELECT P.pressure,P.time
      INTO R1
FROM Pressure AS P
WHERE P.pressure > δ
```

```
SELECT P.pressure,P.time
FROM R1, Pressure AS P
WHERE P.pressure>R1.pressure
AND window(P.time,R1.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 R1
FROM Pressure AS P
WHERE P.pressure > 500
```



Two-Phase Self-Join Processing

- Phase 2
  - Evaluate query Q^*_2
  - Each node performs a join between local sensor data and results of query Q^*_1 i.e., table R_1
- NOTE: We are dealing single window here. Some tuples actually satisfy Q^*_1, e.g., node 3's {520, 5}, and node 5's {550, 9}

```
SELECT P.pressure,P.time
FROM R1, Pressure AS P
WHERE P.pressure > R1.pressure
AND window(P.time,R1.time,10)
```

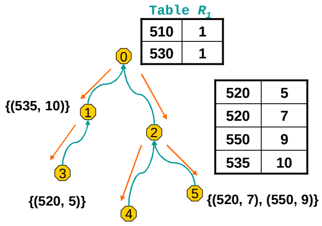


Table R_1

510	1
530	1

520	5
520	7
550	9
535	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 {<, >, ≤, ≥}, 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 {=, ≠}, we can apply some encoding technique, eg, use bitmap to represent a result table

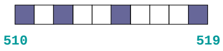
```
Join Predicate:
P.pressure > R1.pressure
```

Table R_1

510	1
530	1

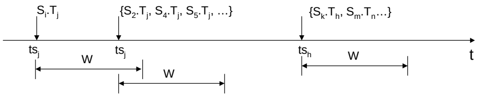
Need only to send this tuple into the network

Table R_1 ={510,512,515,519}



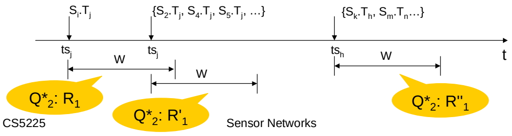
Continuous TPSJ

- 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

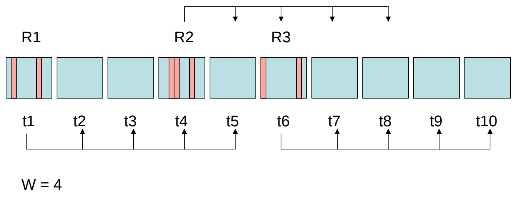


Continuous TPSJ (cont.)

- Phase 1
  - Run Q^*_1 continuously
  - Tuples satisfying p_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

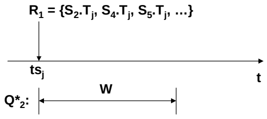


🟦 Sensor table

🟥 Sensor records that matches the selection predicate

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

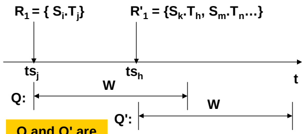


Rule B – Delay New Window Triggering As Much As Possible

-

Q	The phase 2 query currently being executed
Q'	The subsequent execution of the phase 2 query
R_1	The result table of phase 1 used in *Q*
R'_1	The result table of phase 1 used in *Q'*

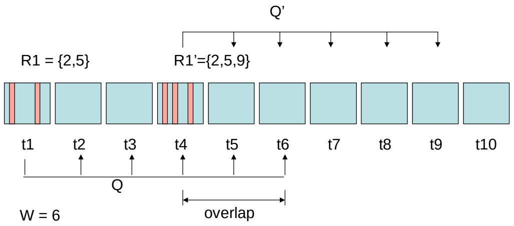
  - Join predicate p_2 is of form $R_1.att_j$ ***op*** $S.att_h$
  - Window W is described as $[t_s, t_e]$ where t_s / t_e is the start / end time.



Q and Q' are of form Q^*_2

Rule B – Delay New Window Triggering As Much As Possible

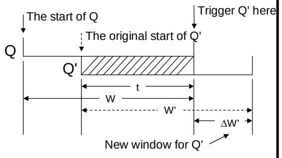
1. If $R_1 \subset R_1$'
   a. If the join predicate is $att_j = att_h$,
      - Q's answer during overlapping window is completely included in Q''s answer
      - Inject Q' and R_1' into the network and start executing Q' instead of Q



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 $W' = [t_s', t_e']$ 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

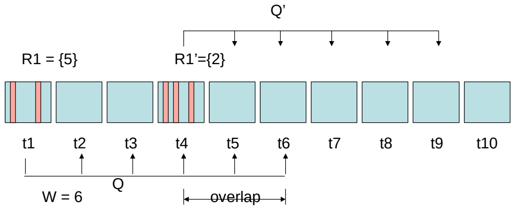


2. Similar rules can be defined if $R_1' \subset R_1$

Rule B

3. If the join predicate att_j op att_h, op $\in$ {<, >, ≤, ≥},
   - 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 op $\in$ {<, ≤}, , 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.

Q: 5 < Sensor.att
Q': 2 < Sensor.att
Q answer ⊆ Q' answer



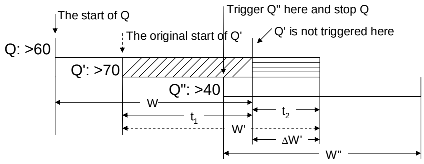
Rule B – Delay New Window Triggering As Much As Possible

3. If the join predicate att_j op att_h, op ∈{<, >, ≤, ≥},
   - Result tables R_1 and R_1' are of size 1
   - Assume R_1={a} and R_1'={b},
   - a. If b < a
     - ii) If op ∈ {>, ≥}, 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_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'??

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_1 = {60}, R_1' = {70}, R_1'' = {40} and the join predicate is $att_h > att_j$
- *Q'* is a *hidden query* and need not be executed

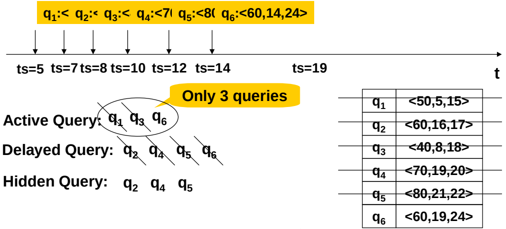


Rule C – Hidden Queries

- 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_i: <v, t_s, t_e>, R_1 = {v}, $W = [t_s,t_e]$, $|W|$ = 10



Active Query: ~~q_1~~ ~~q_3~~ q_6

Delayed Query: ~~q_2~~ ~~q_4~~ ~~q_5~~ ~~q_6~~

Hidden Query: q_2 q_4 q_5

~~q_1~~	~~<50,5,15>~~
~~q_2~~	~~<60,16,17>~~
~~q_3~~	~~<40,8,18>~~
~~q_4~~	~~<70,19,20>~~
~~q_5~~	~~<80,21,22>~~
q_6	<60,19,24>

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