Motivation: Why REED?
- In many different environments (e.g., industrial plants) there is a need to identify constraints or regions of concern in a sensor network. This is known as event detection.
- Traditional query systems (e.g., TinyDB, Cougar) do not have an efficient method to implement event detection, since they lack the use of a join operator.
- Solution: REED
  Performs in-network joins of queries and thereby better supports event detection applications.

Introduction: What is REED?
- REED = Robust and Efficient Event Detection
- Built as an extension to TinyDB, REED uses the same syntax with additional operators for join
- Works by storing filter predicates in a table and then distributing in-network optimization to join queries.
REED: Robust, Efficient Filtering and Event Detection in Sensor Networks

### REED Query Code Example

<table>
<thead>
<tr>
<th>Condition</th>
<th>Time</th>
<th>Temp</th>
<th>Humid</th>
</tr>
</thead>
<tbody>
<tr>
<td>lon &gt;100C</td>
<td>6pm</td>
<td>65%</td>
<td>45%</td>
</tr>
<tr>
<td>lon &gt;110C</td>
<td>7pm</td>
<td>75%</td>
<td>55%</td>
</tr>
<tr>
<td>lon &gt;115C</td>
<td>8pm</td>
<td>85%</td>
<td>65%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**JOIN TABLE**

<table>
<thead>
<tr>
<th>Nod Id</th>
<th>Condition Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>node1</td>
<td>2</td>
</tr>
<tr>
<td>node2</td>
<td></td>
</tr>
<tr>
<td>node3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### REED Query Classes

- **REED defines 3 classes of joins:**
  - **Small:** "join tables that fit in the RAM of a single node"
    - **Small Classes:**
      - "join tables that fit in the RAM of a single node"
    - **Intermediate:**
      - "join tables that exceed the memory of a single node, but can fit in the aggregate memory of a small group of nodes"
    - **Large:**
      - "join tables that exceed the aggregate memory of a group of nodes"

### Single Node Join

- **Root floods message announcing query an extended version of TinyDB “new query” messages**
- The message includes: schema sensor data tuple, name, size, schema of the join table, schema of the result tuples, set of expressions that form join predicate
- Each node checks to see (1) if it shall participate (2) storage capacity
- At each node the join is performed locally

### Distributed Join

- Adaptation from Daniel Abadi, Sam Madden, Wolfgang Lindner (MIT, VLDB -2005)
- When the predicates table does not fit on one node, the table must be horizontally partitioned.
- To do this, nodes organize themselves into groups “cumulatively store the entire table, where all group members are within broadcast range to each other.”
- Each member sends any joined results up to the original root node

### Single Node Join

- Adaptation from Daniel Abadi, Samuel Madden, Wolfgang Lindner (MIT, VLDB -2005)

### Group Operation

- A master begins the creation of a group
- In the node’s finite state machine:
  - If the master (or slave) does not hear from enough nodes, it will go to NEED GROUP state
  - If the event the master (or slave) hears back from original (thought to be failed node) it can transition back to original state
- Given enough nodes respond the master will then contact the root and the root will send the table.
- Groups change with time
- Periodic heartbeat messages sent through the group ascertain activity of nodes and node failures.
- Nodes are responsible for forwarding all child sensor data tuples (even if they are not in the same group)
Finite State Machine

- Robust, Efficient Filtering and Event Detection in Sensor Networks

Optimizations

- Bloom Filters
  - Allows nodes to avoid transmitting when no join will occur.
  - A k-bit Bloom filter $f$, over set values, $J$ that appear in the join columns of the predicates table.
  - Nodes are also programmed with hash function $H$.

- A node will then compute $H(v)$ and verify to see if the corresponding entry is set in $f$. If not it will not join.

Optimizations

- Partial Joins
  - Allows sensors to identify tuples that do not join with any predicates in the range.

- Cache Diffusion
  - Policy to reduce communication cost by filtering tuple of sensor data that is not needed for the join to occur.
  - Each node has cache: (1) local value cache (2) empty range descriptions (ERDs) of the join.
  - An ERD consists of a set of: "ranges in the domain of these attributes: $[x_1 \cdots y_1] \cdots [x_n \cdots y_n]$ such that a tuple contains values for each of these attributes."
  - When the root receives a tuple that does not join it sends a max ERD one hop in the direction of that tuple. If a tuple within ERD range is found then it is removed and the ERD (priority 1).

Experimental Results

- Figure 4: Total Transmissions vs. Selectivity
- Figure 5: Breakdown of Transmission Types for Distributed Join with Varying Selectivity
- Figure 6: Received Tuples vs. Selectivity for Distributed Join Algorithm
Discussion

- Event representation
- Large join is common in practice
- How to determine which optimization strategy is applicable?
- Can the approach adapt when the event table is incrementally updated?

Conclusion

- REED demonstrates how complex filters may be expressed as a table of predicates for in-network joins to reduce the total transmissions
- REED takes into consideration memory constrained networks
- REED accommodates for node failure or loss and thus produces a robust method of detecting an event in a network of many sensors