Query the Sensor Network

Jie Gao
Computer Science Department
Stony Brook University
Make use of data from sensornets

• Data collection, storage, mining, query.

• Integration with the Internet

• “google” service for the physical world.
Challenges: data

- Data type: numerical sensor readings.
- Rich and massive data, spatially distributed and correlated.
- Data dynamics: data streaming and aging.
- Uncertainty, noise, erroneous data, outliers.
- Semantics. Raw data ➔ knowledge.
Challenges: query variability

• **Data-centric query**: search for “car detection”, instead of sensor node ID.
• **Geographical query**: report values near the lake.
• **Real-time detection & control**: intruder detection.
• **Multi-dimensional query**: spatial, temporal and attribute range.
• **Query interface**: fixed base station or mobile hand held devices.
Data processing

- In-network aggregation
- In-network storage
- Distributed data management
- Statistical modeling
- Intelligent reasoning
In-network data aggregation

- Communication is expensive, bandwidth is precious.
  - “In-network processing”: process raw data before transmit.

- Single sensor reading may not hold much value.
  - Inherently unreliable, outlier readings.
  - Users are often interested in the hidden patterns or the global picture.

- Data compression and knowledge discovery.
  - Save storage; generate semantic report.
Distributed In-network Storage

- Flash drive, etc. enables distributed in-network storage

- Challenges
  - Distributed indexing for fast query dissemination
  - Explore storage locality to benefit data retrieval.
  - Resilience to node or link failures.
  - Graceful adaptation to data skews.
  - Alleviate the “hot spot” problem created by popular data.
Sound statistical models

- Raw data may misrepresent the physical world.
  - Sensors sample at discrete times. Sensors may be faulty. Packets may be lost.
  - Most sensor data may not improve the answer quality to the query. Data can be compressed.
  - Correlation between nearby sensors or different attributes of the same sensor.
Model-based query

- Build statistical models on the sensor readings.
  - Generates observation plan to improve model accuracy.
  - Answers query results.
- Pros:
  - Improve data robustness.
  - Explore correlation
  - Decrease communication cost.
  - Provide prediction of the future.
  - Easier to extract data abstraction.

**Probabilistic Queries**

```
"SELECT nodeId, temp ± 1°C, conf(.95)
WHERE nodeId in {1..8}"
```

**Query Results**

```
1, 22.3 97%
2, 25.6 99%
3, 24.4 95%
4, 22.1 100%
```

**Observation Plan**

```
"[[voltage,1],
[voltage,2],
[temp,4]]"
```

**Data**

```
1, voltage = 2.73
2, voltage = 2.65
4, temp = 22.1"
```
Reasoning and control

- Reason from raw sensor readings for high-level semantic events.
  - Fire detection.

- Events triggered reaction, sensor tasking and control.
  - Turn on fire alarm. Direct people to closest exits.
Data privacy, fault tolerance and security

- Under what format should data be stored?
- What if a sensor die? Can we recover its data?
- What information is revealed if a sensor is compromised?
- Adversary injects false reports and false alarms.
Approximation and randomization

• Connection to streaming data model:
  – No way to store the raw data.
  – Scan the data sequentially.
  – Maintain sketches of massive amount of data.
  – One more challenge in sensor network: the streaming data is spatially distributed and communication is expensive.

• Approximations, sampling, randomization.
Papers


TinyDB

• Philosophy:
  – Sensor network = distributed database.
  – Data are stored locally.
  – Top-down SQL query.
  – Results aggregated back to the query node.
  – Most intelligence outside the network.
TinyDB Architecture

PC side

Mote side

Sensor network

TinyDB query processor

TinyDB GUI

TinyDB Client API

DBMS

JDBC

The next few slides from Sam Madden, Wei Hong
Query Language (TinySQL)

SELECT <aggregates>, <attributes>
[FROM {sensors | <buffer>}]  
[WHERE <predicates>]
[GROUP BY <exprs>]
[SAMPLE PERIOD <const> | ONCE]
[INTO <buffer>]
[TRIGGER ACTION <command>]

16
TinySQL Examples

“Find the sensors in bright nests.”

1

SELECT nodeid, nestNo, light
FROM sensors
WHERE light > 400

EPOCH DURATION 1s

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Nodeid</th>
<th>nestNo</th>
<th>Light</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>17</td>
<td>455</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>25</td>
<td>389</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>17</td>
<td>422</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>25</td>
<td>405</td>
</tr>
</tbody>
</table>
“Count the number occupied nests in each loud region of the island.”

- **SELECT AVG(sound)**
  FROM sensors
  EPOCH DURATION 10s

- **SELECT region, CNT(occupied)**
  **AVG(sound)**
  FROM sensors
  GROUP BY region
  HAVING AVG(sound) > 200
  EPOCH DURATION 10s

<table>
<thead>
<tr>
<th>Epoch</th>
<th>region</th>
<th>CNT(...)</th>
<th>AVG(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>North</td>
<td>3</td>
<td>360</td>
</tr>
<tr>
<td>0</td>
<td>South</td>
<td>3</td>
<td>520</td>
</tr>
<tr>
<td>1</td>
<td>North</td>
<td>3</td>
<td>370</td>
</tr>
<tr>
<td>1</td>
<td>South</td>
<td>3</td>
<td>520</td>
</tr>
</tbody>
</table>

Regions w/ AVG(sound) > 200
Data Model

• Entire sensor network as one single, infinitely-long logical table: sensors
• Columns consist of all the attributes defined in the network
• Typical attributes:
  – Sensor readings
  – Meta-data: node id, location, etc.
  – Internal states: routing tree parent, timestamp, queue length, etc.
• Nodes return NULL for unknown attributes
Query over Stored Data

- Named buffers in Flash memory
- Store query results in buffers
- Query over named buffers
- Analogous to materialized views
- Example:
  - CREATE BUFFER name SIZE x (field1 type1, field2 type2, …)
  - SELECT a1, a2 FROM sensors SAMPLE PERIOD d INTO name
  - SELECT field1, field2, … FROM name SAMPLE PERIOD d
Event-based Queries

- ON event SELECT …
- Run query only when interesting events happens
- Event examples
  - Button pushed
  - Message arrival
  - Bird enters nest
- Analogous to triggers but events are user-defined
TAG: Tiny Aggregation

- **Query Distribution**: aggregate queries are pushed down the network to construct a spanning tree.
  - Root broadcasts the query, each node hearing the query broadcasts.
  - Each node selects a parent. The routing structure is a spanning tree rooted at the query node.

- **Data Collection**: aggregate values are routed up the tree.
  - Internal node aggregates the partial data received from its subtree.
TAG example

Query distribution

Query collection
TAG example

\[ m_4 = \max\{m_6, m_5\} \]

Count: \( c_4 = c_6 + c_5 \)

Sum: \( s_4 = s_6 + s_5 \)
Considerations about aggregations

• Packet loss?
  – Acknowledgement and re-transmit?
  – Robust routing?

• Packets arriving out of order or in duplicates?
  – Double count?

• Size of the aggregates?
  – Message size growth?
Classes of aggregations

- **Exemplary** aggregates return one or more representative values from the set of all values; **summary** aggregates compute some properties over all values.
  - MAX, MIN: exemplary; SUM, AVERAGE: summary.
  - Exemplary aggregates are prone to packet loss and not amendable to sampling.
  - Summary aggregates of random samples can be treated as a robust estimation.
Classes of aggregations

- **Duplicate insensitive** aggregates are unaffected by duplicate readings.
  - Examples: MAX, MIN.
  - Independent of routing topology.
  - Combine with robust routing (multi-path).
Classes of aggregations

• **Monotonic aggregates**: when two partial records $s_1$ and $s_2$ are combined to $s$, either 
  $e(s) \geq \max\{e(s_1), e(s_2)\}$ or $e(s) \leq \min\{e(s_1), e(s_2)\}$.
  
  – Examples: MAX, MIN.
  
  – Certain predicates (such as HAVING) can be applied early in the network to reduce the communication cost.
Classes of aggregations

- Partial state of the aggregates:
  - **Distributive**: the partial state is simply the aggregate for the partial data. The size is the same with the size of the final aggregate. Example: MAX, MIN, SUM
  - **Algebraic**: partial records are of constant size. Example: AVERAGE.
  - **Holistic**: the partial state records are proportional in size to the partial data. Example: MEDIAN.
  - **Unique**: partial state is proportional to the number of distinct values. Example: COUNT DISTINCT.
  - **Content-sensitive**: partial state is proportional to some (statistical) properties of the data. Example: fixed-size bucket histogram, wavelet, etc.
# Classes of aggregates

<table>
<thead>
<tr>
<th></th>
<th>Duplicate sensitive</th>
<th>Exemplary, Summary</th>
<th>Monotonic</th>
<th>Partial State</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX, MIN</td>
<td>No</td>
<td>E</td>
<td>Yes</td>
<td>Distributive</td>
</tr>
<tr>
<td>COUNT, SUM</td>
<td>Yes</td>
<td>S</td>
<td>Yes</td>
<td>Distributive</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>Yes</td>
<td>S</td>
<td>No</td>
<td>Algebraic</td>
</tr>
<tr>
<td>MEDIAN</td>
<td>Yes</td>
<td>E</td>
<td>No</td>
<td>Holistic</td>
</tr>
<tr>
<td>COUNT DISTINCT</td>
<td>No</td>
<td>S</td>
<td>Yes</td>
<td>Unique</td>
</tr>
<tr>
<td>HISTOGRAM</td>
<td>Yes</td>
<td>S</td>
<td>No</td>
<td>Content-sensitive</td>
</tr>
</tbody>
</table>
Communication cost

Send all data to the sink

Partial states too large!
Problem with median

- Computing average is simple on an aggregation tree.
  - Each node $x$ stores the average $a(x)$ and the number of nodes in its subtree $n(x)$.
  - The average of a node $x$ can be computed from its children $u$, $v$. $n(x) = n(u) + n(v)$. $a(x) = (a(u)n(u) + a(v)n(v))/n(x)$.

- Computing the median with a fixed amount of message is hard.
  - We do not know the rank of $u$’s median in $v$’s dataset.
  - We resort to approximations.
Deal with computing median

• Resort to approximation.
  – Random sampling approach.
  – A deterministic approach.
Approach I: Random sampling

- Problem: compute the median $a$ of $n$ unsorted elements $\{a_i\}$.
- Solution: Take a random sample of $k$ elements $K$. Compute the median $x$ of $K$.
- Claim: $x$ has rank within $(\frac{1}{2}+\varepsilon)n$ and $(\frac{1}{2}-\varepsilon)n$ with probability at least $1-\frac{2}{\exp\{2k\varepsilon^2\}}$. (Proof left as an exercise.)
- Choose $k=\ln(2/\delta)/(2\varepsilon^2)$, then $x$ is an approximate median with probability $1-\delta$. 
Approach II: Quantile digest (q-digest)

- A data structure that answers
  - Approximate quantile query: median, the kth largest reading.
  - Range queries: the kth to lth largest readings.
  - Most frequent items.
  - Histograms.

- Properties:
  - Deterministic algorithm.
  - Error-memory trade-off.
  - Confidence factor.
  - Support multiple queries.
Q-digest

• Input data: frequency of data value \{f_1, f_2, ..., f_\sigma\}.
• Compress the data:
  – detailed information concerning frequent data are preserved;
  – less frequently occurring values are lumped into larger buckets resulting in information loss.
• Buckets: the nodes in a \textit{binary partition} of the range [1, \sigma]. Each bucket \(v\) has range \([v.\text{min}, v.\text{max}]\).
• Only store non-zero buckets.
Example

Input data bucketed

Q-digest

Information loss

n = 15, k = 5, σ = 8
Q-digest properties

- Store values in buckets.
  1. \( \text{Count}(v) \leq \frac{n}{k} \) (except leaf)
    - Control information loss.
  2. \( \text{Count}(v) + \text{Count}(p) + \text{Count}(s) > \frac{n}{k} \) (except root)
    - Ensure sufficient compression.
    - \( K \): compression parameter.
Construct a q-digest

- Each sensor constructs a q-digest based on its value.
- Check the digest property bottom up: two “small” children’s count are added up and moved to the parent.
Merging two q-digests

- Merge q-digests from two children
- Add up the values in buckets
- Re-evaluate the digest property bottom up.

Information loss: t undercounts since some of its value appears on ancestors.
Claim: A q-digest with compression parameter $k$ has at most $3k$ buckets.

- By property 2, for all buckets $v$ in $Q$,
  - $\sum_{v \in Q} [\text{Count}(v) + \text{Count}(p) + \text{Count}(s)] > |Q| \frac{n}{k}$.
  - $\sum_{v \in Q} [\text{Count}(v) + \text{Count}(p) + \text{Count}(s)] \leq 3 \sum_{v \in Q} \text{Count}(v) = 3n$.
  - $|Q| < 3k$. 

---

**Space complexity**
Error bound

Claim: Any value that should be counted in v can be present in one of the ancestors.

1. Count(v) has max error $\log \sigma \cdot n/k$.
   - $\text{Error}(v) \leq \sum_{\text{ancestor } p} \text{Count}(p) \leq \sum_{\text{ancestor } p} n/k \leq \log \sigma \cdot n/k$.

2. MERGE maintains the same relative error.
   - $\text{Error}(v) \leq \sum_i \text{Error}(v_i) \leq \sum_i \log \sigma \cdot n_i/k \leq \log \sigma \cdot n/k$. 
Median and quantile query

• Given $q \in (0, 1)$, find the value whose rank is $qn$.

• Relative error $\varepsilon = |r - qn|/n$, where $r$ is the true rank.

• Post-order traversal on $Q$, sum the counts of
Other queries

• **Inverse quantile**: given a value, determine its rank.
  – Traverse the tree in post-order, report the sum of counts \( v \) for which \( x > v.\text{max} \), which is within \([\text{rank}(x), \text{rank}(x) + \varepsilon n]\)

• **Range query**: find \# values in range \([l, h]\).
  – Perform two inverse quantile queries and take the difference. Error bound is \(2\varepsilon n\).

• **Frequent items**: given \( s \in (0, 1) \), find all values reported by more than \( sn \) sensors.
  – Count the leaf buckets whose counts are more than \( sn \).
  – Small false positive: values with count between \((s-\varepsilon)n\) and \(sn\) may also be reported as frequent.
Simulation setup

- A typical aggregation tree (BFS tree) on 40 nodes in a 200 by 200 area. In the simulation they use 4000~8000 nodes.
Simulation setup

- Random data;
- Correlated data: 3D elevation value from Death Valley.
Histogram v.s. q-digest

• Comparison of histogram and q-digest.
Tradeoff between error and msg size

![Graph showing the tradeoff between percentage error and message size for correlated and random data. The graph illustrates that as message size increases, the percentage error decreases.]
Saving on message size

Naïve solution

![Graph showing message size vs number of sensors](image)
Conclusion

• Aggregation sometimes requires careful design to tradeoff accuracy & storage/message size.

• Aggregation incurs information loss, making robust estimation more difficult. E.g. a single outlier reading can screw up MAX/MIN aggregates.
Project presentation

• 10/17, 10/19
• Each group please send me your group members and project title. I will send out a presentation list Thursday.