Lecture 7:

TinyDB, Data Aggregation and Median Computation

Rik Sarkar
Data Collection

- Large number of nodes
- Not all data is of interest: allow the user to query
  - Network as a database
- Redundant data
- Summarize data
- Respond with “Useful Information” not raw data
- Tolerance to noise, errors

Today:


• Nisheeth Shrivastava, Chiranjeeb Buragohain, Divy Agrawal, Subhash Suri, *Medians and Beyond: New Aggregation Techniques for Sensor Networks*, SenSys '04,
TinyDB

- Designed for sensor networks with tiny nodes
- Network as a database
- SQL for networks
- Uses tree based aggregation structure
- Small amount of In-network processing, most of the intelligence outside the network
TinyDB Architecture

PC side

TinyDB GUI
TinyDB Client API

JDBC
DBMS

Mote side

Sensor network

TinyDB query processor

Database

• The entire network is one infinitely long table
• Columns: all known attributes
  – Sensor reading
  – Metadata: node id, location
  – State information: battery level, routing table, timestamp
• All nodes need not have data with all attributes: return NULL for unknown attributes
TinySQL

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

Comparison with SQL

• Single table in FROM clause
• Only conjunctive comparison predicates in WHERE and HAVING
• No subqueries
• No column alias in SELECT clause
• Arithmetic expressions limited to column op constant
• Only fundamental difference: SAMPLE PERIOD clause
TinySQL Examples

“Find the sensors in bright nests.”

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>

Sensors
TinySQL Examples (cont.)

2. SELECT AVG(sound)
   FROM sensors
   EPOCH DURATION 10s

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

“Count the number occupied nests in each loud region of the island.”

<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>
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 happen
• Event examples
  – Button pushed
  – Message arrival
  – Bird enters nest
• Analogous to triggers but events are user-defined
TAG: Tiny Aggregation

• Query Distribution by flooding
  – Root broadcasts query
  – Every node sends to all their neighbors
  – Every node selects a parent: aggregation tree

• Data collection: Use aggregation tree to send values to root
  – A node aggregates data from all its subtrees and forwards to parent
TAG

Query distribution

Query collection

Summary

- Network as a database
- SQL for networks
- Uses tree based aggregation structure
- Less sophisticated than location based storages (GHT, DIM, fractional cascading)
- Small amount of In-network processing
- Easier to implement, needs fewer assumption on network
Issues in aggregation

• Transmission may be lossy
  – Needs acknowledgement and retransmit scheme
  – Are there better routing methods?
• Packets may arrive out of order, or multiple copies
  – How do we avoid double counting?
• Sizes of aggregates
  – How does the message grow?
Classes of Aggregates

• Exemplary: One or more values from the set provide the value e.g. Max, Min
  – Packet loss loses the aggregate
  – Hard to estimate with sampling

• Summary: Computed from all values e.g. Sum, Average
  – Random samples can provide good estimates
  – Not so sensitive to packet loss

Classes of Aggregates

• Duplicate Insensitive: Unaffected by duplicate readings e.g. Max, Min
  – Independent of routing method
  – Can be combined with robust multi-path routing
Classes of Aggregates

• Monotonic aggregates:
• Suppose $s = s_1 \oplus s_2$
• Always $e(s) \geq \max\{s_1, s_2\}$ or
• Always $e(s) \leq \min\{s_1, s_2\}$
• E.g. Max, Min
Classes of Aggregates

• Data for partial state records
  – **Distributive**: Partial state data is just the aggregate of partial data. E.g.: Max, Min, Sum
  – **Algebraic**: Not aggregates of partial data, but of const. size. E.g. Average
  – **Holistic**: Partial state records are entire partial data. E.g. Median
  – **Unique**: Partial state depends on number of unique values. E.g. Count Unique
  – **Content-sensitive**: Proportional to some statistical property of data. E.g. Fixed width histograms, wavelet
Communication costs

Almost all data has to reach the sink.

In-network aggregation does not help.

Partial States Large.
Median Computation

• Harder than computing sums
• Partial results cannot be merged
• Suppose we have two lists A and B
• With medians \( m(A) \) and \( m(B) \)
• We cannot infer the median of \( A + B \) from \( m(A) \) and \( m(B) \)
Handling medians: Quantile Digest

- q-digest data structure
  - Find approximately the kth largest value
  - Range queries: kth to lth largest values
  - Histograms and frequent items
  - Error-memory (state size) tradeoff
Q-digest

• Input: frequency of occurrence of different values in range \([1, \sigma]\)
• Data is divided into different buckets
  – 1 Bucket of size \(\sigma\)
  – 2 Buckets of size \(\sigma/2\)
  – 4 Buckets of size \(\sigma/4\) etc..
• Each bucket \(v\) has range \([v_{\text{min}}, v_{\text{max}}]\)
An Example

Input data bucketed

Q-digest

Information loss

Properties of q-digest

1. \( \text{count}(v) \leq \frac{n}{k} \) (unless \( v \) is a leaf)
2. \( \text{count}(v) + \text{count}(\text{parent}) + \text{count}(\text{siblings}) > \frac{n}{k} \)
   (unless \( v \) is root)

\( k \): compression parameter
Construction of q-digest (centralized)

- Check digest property bottom-up
Q-digest: distributed in an aggregation tree

• Each sensor starts with a digest with only one value
• Sends to its parent in aggregation tree
• Parent merges the digests of all children and itself
• And repeats
• Note: to send q-digest, only need to send non-empty nodes
Merging digests

Add values in corresponding buckets

Process the digest property bottom up
Space complexity

• A q-digest with compression parameter k has at most 3k non-empty buckets

• By property 2, for all non empty buckets v in Q:
  – $\text{count}(v) + \text{count}(\text{parent}) + \text{count}(\text{Sibling}) > n/k$
  – $\sum_v [\text{count}(v) + \text{count}(\text{parent}) + \text{count}(\text{Sibling})] > |Q| \cdot n/k$
  – $\sum_v [\text{count}(v) + \text{count}(\text{parent}) + \text{count}(\text{Sibling})] \leq 3[\sum_v \text{count}(v)] = 3n$
  – $|Q| < 3k$
Error bounds

• A value that was in v may have propagated to a parent

• \( \text{count}(v) \) has max error \( \log \sigma \cdot n/k \)
  – \( \text{error}(v) \leq \sum_{\text{ancestor } p} \text{count}(p) \leq \sum_{\text{ancestor } p} \frac{n}{k} \leq \log \sigma \cdot \frac{n}{k} \)

• Merge maintains the same relative error
  – \( \text{error}(v) \leq \sum_i \text{error}(v_i) \leq \sum_i \log \sigma \cdot \frac{n_i}{k} \leq \log \sigma \cdot \frac{n}{k} \)
Median and quantile query

• Given $q \in (0,1)$, find the value at rank $q_n$

• Relative error $\varepsilon = |r - q_n|/n$, where $r$ is the true rank of the value found by our algorithm

• Error bound: $\varepsilon = \log \sigma / k$
Other queries

- **Inverse quantile**: Given a value x, determine its rank: traverse the tree post order, and take counts < x. Error within $\varepsilon n$
- **Range query**: Find #values in range $[k, l]$: two inverse quantiles and take diff. error $2\varepsilon n$
- **Frequent items**: find all values reported by sn sensors ($s$ is a fraction)
  - Count leaf buckets whose counts are more than $(s-\varepsilon)n$
Simulation

- Compute an aggregation tree from BFS tree
- 4000-8000 nodes
Data

- Random data
- Correlated data: 3D elevation value from death valley
Comparison with true histogram

![Comparison with true histogram](image-url)
Reminder

• Project proposal due by Dec 1.
• Problem set 1 will be given out soon!