A Database System for Sensors

Jun Yang
CPS 296.1, Spring 2007
Sensor Data Processing

Announcements (Jan. 23)
- Auditing students
  - Please make sure you on are the mailing list
  - Please register as auditing
- Check your email today to sign up for the first round of presentations
- Thursday: Synopsis Diffusion by Narh et al.
  - No review due
- Next Tuesday: Monitoring of Extreme Values by Silberstein et al. and Contour Map by Xue et al.
  - One review due (you choose which)

Today’s paper

What’s different from a traditional DB?
- Data model?
- Query language and semantics?
- Indexing?
- Optimization?
- Processing?

Data model
- A single, append-only table
  *Sensors (nodeid, time, light, temp, …)*
- Just a conceptual view for posing queries; in reality:
  - Data is not already there at query time
    - Traditional database: queries independent of acquisition
    - Here: queries drive acquisition
      - Didn’t ask for light? Then it won’t be sampled!
  - Data may not be at one place
    - Like a distributed database, but here nodes/network are much less powerful/reliable
  - Data won’t be around forever
    - Similar to stream data processing

Epoch-based model of operation

*Time Versus Current Draw in Different Phases of Query Processing*
Simple queries

- SELECT nodeid, light, temp
  FROM Sensors
  SAMPLE PERIOD 1s FOR 10s;
- Or, instead of SAMPLE PERIOD:
  LIFETIME 30 days
  - Use remaining battery capacities to come up with an
    appropriate sample period to match required lifetime
    "How does multi-hop network complicate estimation?"
- Note continuous, periodic nature
  - Specify ONCE to get the traditional snapshot behavior

Storage points

- CREATE STORAGE POINT recentLight SIZE 8 AS
  (SELECT nodeid, light FROM Sensors
   SAMPLE PERIOD 10s);
  - A sliding window of recent readings, materialized locally
- Joining with the Sensors stream
  - SELECT COUNT(*)
    FROM Sensors s, recentLight rl
    WHERE rl.nodeid = s.nodeid AND s.light < rl.light
    SAMPLE PERIOD 10s;
  "TinyDB only allows joining a stream with a storage point—why?"

Aggregation

- Instantaneous aggregation
  - SELECT AVG(volume), room FROM Sensors
    WHERE floor = 6
    GROUP BY room
    HAVING AVG(volume) > threshold
    SAMPLE PERIOD 30s;
  - A batch of new result rows for every new set of readings
    taken at the same time
- Sliding-window aggregation
  - SELECT WINAVG(volume, 30s, 5s) FROM Sensors.
  "What does a node need to remember to process this query?"

Events

- Detecting/generating events
  - SELECT nodeid, temp FROM Sensors
    WHERE temp > threshold
    OUTPUT ACTION SIGNAL hot(nodeid, temp)
    SAMPLE PERIOD 10s;
- Event-triggered querying
  - ON EVENT hot(nodeid, temp):
    SELECT nodeid, humidity FROM Sensors s
    WHERE s.nodeid = nodeid
    SAMPLE PERIOD 2s FOR 30s;
  "So why this construct?"

Now, under the hood…

- Data model
- Query language and semantics
- Indexing?
- Optimization?
- Processing?

Query lifecycle in TinyDB

- Plan once
  - Parsing, validation, and optimization at base station
  - Execution plan (binary code) + choice of routing tree
  - Disseminate execution plan into network
- Execute continuously
  - Periodically, each node executes its local subplan
    (sampling, combining results from children, …), and
    sends results up
    "In-network processing greatly reduces amount of data transmitted"
  - Base station delivers final results to users
- Occasional replanning
Metadata

What info is needed for query processing/optimization?

- About attributes
  - E.g., light, temperature, etc., whose values can be acquired
  - Cost of acquisition (power, time)
  - Range of possible values
    - Can use more sophisticated histograms instead
  - Change characteristics
- About events, extensible aggregate system, energy consumption, ...

Reorder sampling and predicates

- Basic idea
  - Consider: if \( a \) and \( b \) then return \( c \)
  - If \( a \) is false, then no need to acquire readings needed to evaluate \( b \) or to return \( c \)
- Example
  - SELECT accel, mag FROM Sensors
    WHERE accel \( \geq \) \( c_1 \) AND mag \( \geq \) \( c_2 \)
    SAMPLE PERIOD .1s;
  - When is it better to evaluate \( \text{ accel} > c_1 \) before \( \text{ mag} > c_2 \)?
  - What info is needed to make this decision?

A more subtle example

- SELECT WINMAX(light, 8s, 8s)
  FROM Sensors
  WHERE mag \( \geq \) \( x \)
  SAMPLE PERIOD 1s;
- Non-optimized: test mag \( > x \); if true, then add light to the sliding window
  - If new light reading doesn’t affect the max in the current window anyway, there is no need to acquire mag
  - What info is needed to make this decision?

Batching

- One event-triggered query
  ON EVENT hot(nodeid, temp):
    SELECT nodeid, humidity
    FROM Sensors s
    WHERE s.nodeid = nodeid
    SAMPLE PERIOD 2s FOR 30s;
  - May generate many instances of the same query
  - Share processing among these queries
- Equivalent (?) to evaluating
  SELECT s.nodeid, s.humidity
  FROM Sensors s, HotEvents e
  WHERE s.nodeid = e.nodeid
  AND s.time – e.time <= 30s AND s.time > e.time
  SAMPLE PERIOD 2s;

Timing issues...

- When batching, what if different query instances start at different times?
- If we order sampling and predicates sequentially, we can no longer take readings synchronously
- When joining a storage point and a stream, what if their sampling points don’t align?
  - Tension between continuous signals and discrete events

An “index”: semantic routing tree

- SELECT … FROM Sensors WHERE \( A \in \) range...
  - Not sure which sensors have these \( A \) values?
  - Need to probe the entire network
- Use an index
  - Search tree = routing tree
  - Intermediate nodes store bounding boxes for subtrees
- What’s different from DB search trees?
Construction and maintenance

- Construction is like running an aggregation query
  - A lot more about in-network aggregation in next lecture
  - What are the factors to consider when a node chooses its parent in SRT?
- When topology or indexed attribute value changes, propagate updates upwards
  - When, and on what attributes, would you choose to construct SRTs?

Question answered?

- Once again, declarative interface rules!
  - Brings ease of programming and maintenance, and flexibility in under-the-hood changes
  - Supports declarative data acquisition too!
  - Allows acquisition to be optimized together with processing
  - Hides complexity of planning acquisition and communication together with local computation
  - Hides complexity of handling uncertainty (e.g., failures, bursts)
- Under the hood, engine looks different but its basic structure still remains intact
  - Compile and execute lifecycle
  - Cost-based optimization using statistics
  - In-network processing becomes critical

Other ideas

- Scheduling with a tree topology
- “Snooping”
- Prioritizing data delivery: important data first—you never know when things may go wrong, even with extensive planning

Q’s unanswered + future work ideas

- More signal processing than symbol processing?
- How is uncertainty quantified?
- Communication patterns other than trees?
- Are events the right building blocks for more dynamic adaptation/control?
  - How about in-network, but non-local control?
  - How about treating query parameters as query input?
- What about predictability?
  - E.g., when TinyDB automatically adapts to network condition and battery level
- A better index for attributes whose value change characteristics are not uniform across nodes?