The Design of an Acquisitional Query Processor For Sensor Networks

Samuel Madden, Michael J. Franklin & Joseph M. Hellerstein (UC Berkeley)

Wei Hong (Intel Research, Berkeley)

Reviewed by

Hareesh Nagarajan
hnagaraj@cs.uic.edu
Department of Computer Science
University of Illinois at Chicago

Overview of the paper

- Design of an acquisitional query processor for data collection in sensor networks (SN)
- What are acquisitional issues?
  - Where
  - When
  - Frequency

  (with which the data is acquired/sampled)
- By focusing on locations and costs of acquiring data authors were able to **significantly reduce power consumption** over traditional passive systems which assume “a priori” existence of data

Overview of the paper

- The authors describe extensions to SQL for
  - Controlling data acquisition
- The also show how acquisitional issues influence
  - Query optimization
  - Query dissemination
  - Query execution

Preliminary: Where should data be stored?

What paper talks about!

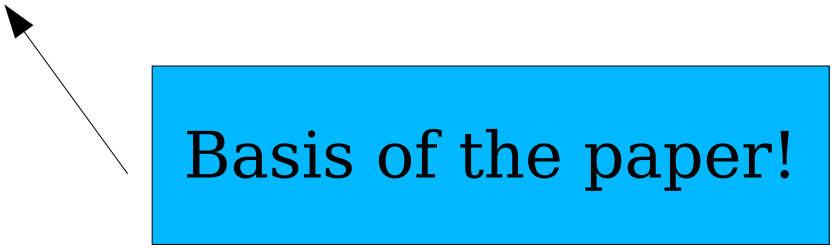
- **Centralized Storage & Querying**
  - Power Inefficient
  - Too much data isn't useful
- **Multi-resolution Storage & Indexing**
  - Raw data at leaves & processsed data at higher powers
  - Processing & Hierarchial storage needs power

- **Local Storage & Distributed Querying**
  - Query processing is performed on demand so energy efficient
  - It is useful when:
    - Queries are simple
    - Schemes can deal with node failure

Other schemes

Prior Systems V/S AQCP

- Prior systems tend to view query processing in SN as a
  - Power-constrained version of traditional query processing
- Authors present **Acquisitional Query Processing** (ACQP) which make use of the fact that
  - Smart sensors have control over where, when and how often data is physically aquired (sampled) and delivered to query processing operators



More on AQCP

- By focusing on
  - **Location**
  - **Costs of acquiring data**

  System significantly reduces power consumption.
- They also say acquisitional issues arise at all levels of query processing
  - **Query optimization** (because there is a cost involved with sampling sensors!)
  - **Query dissemination** (due to the physical co-location of sampling & processing)
  - **Query execution** (here choices of when to sample and which samples to process are made!)

Basic Architecture

Queries are submitted to base station which parses, optimizes and then sends query on the SN

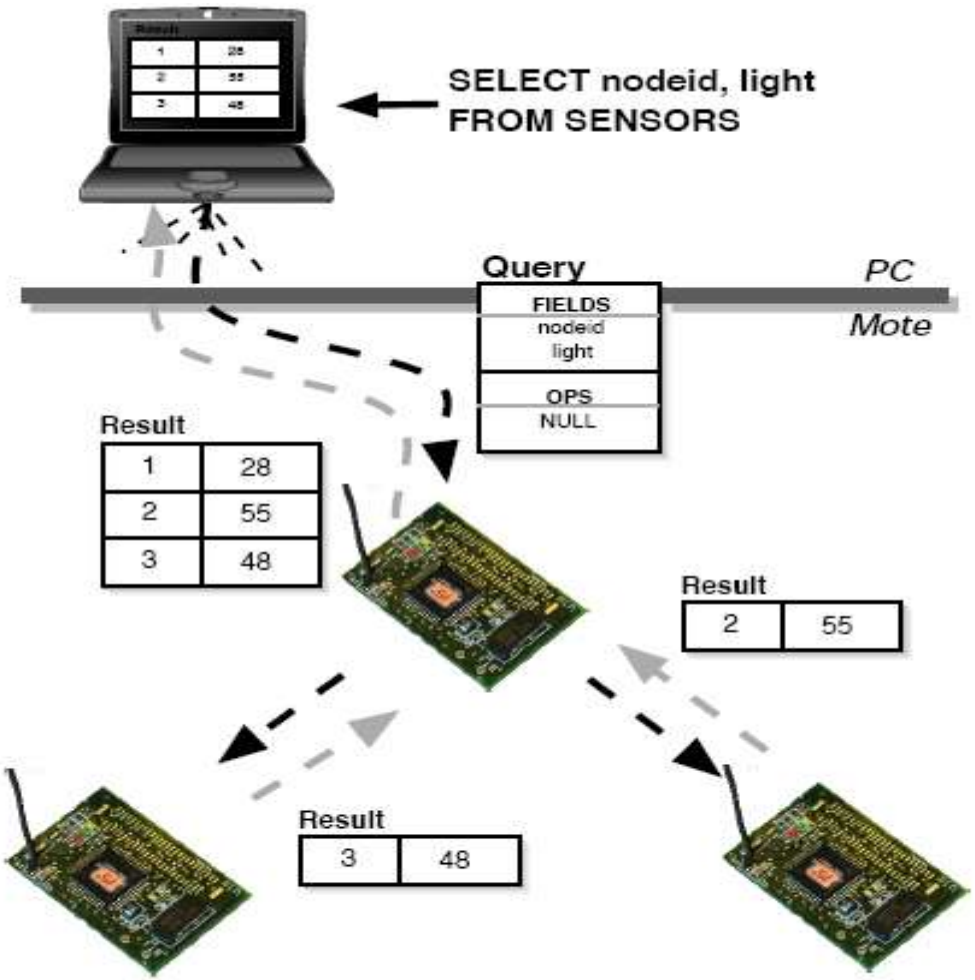


Figure 1: A query and results propagating through the network.

Basic architecture

- Authors have designed and implemented an AQCP engine called TinyDB – distributed query processor that runs on each of the nodes in SN.
- TinyDB runs on top of Berkeley mica “mote” platform, which in turn runs on top of TinyOS
- TinyDB eliminates the need to write C code for TinyOS users
- TinyDB has many features of traditional query processor
  - Select, Join, Project, Aggregate Data

Some properties of sensor devices

- E.g.: Mica motes operating at 2% duty cycle can achieve lifetimes in 6 months on 2-AA batteries. Hence active time is limited to 1.2 seconds/minute.
- Power consumption is dominated by radio communication.
- E.g.: A bit of data transmitted = energy expended in executing 1000 CPU instructions.

Phases of power consumption in TinyDB

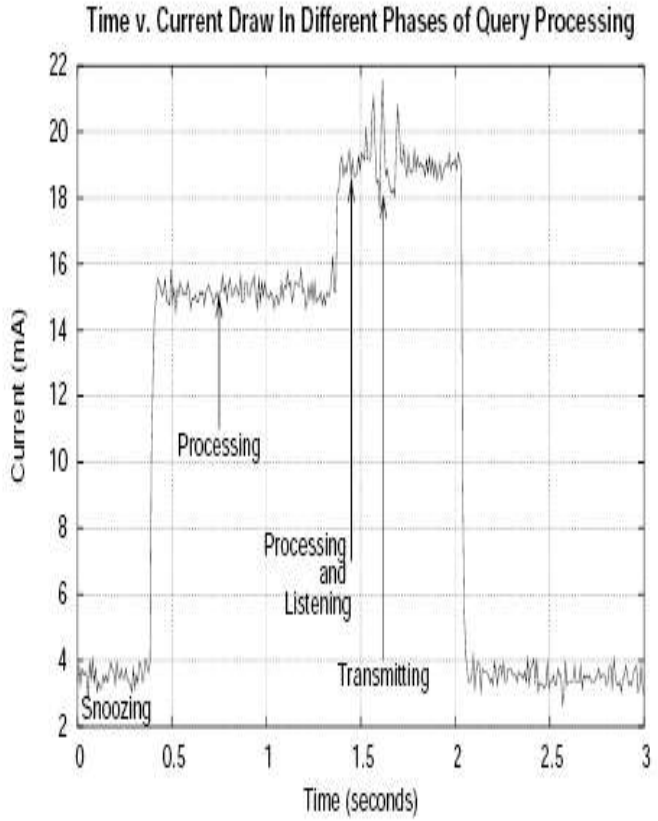


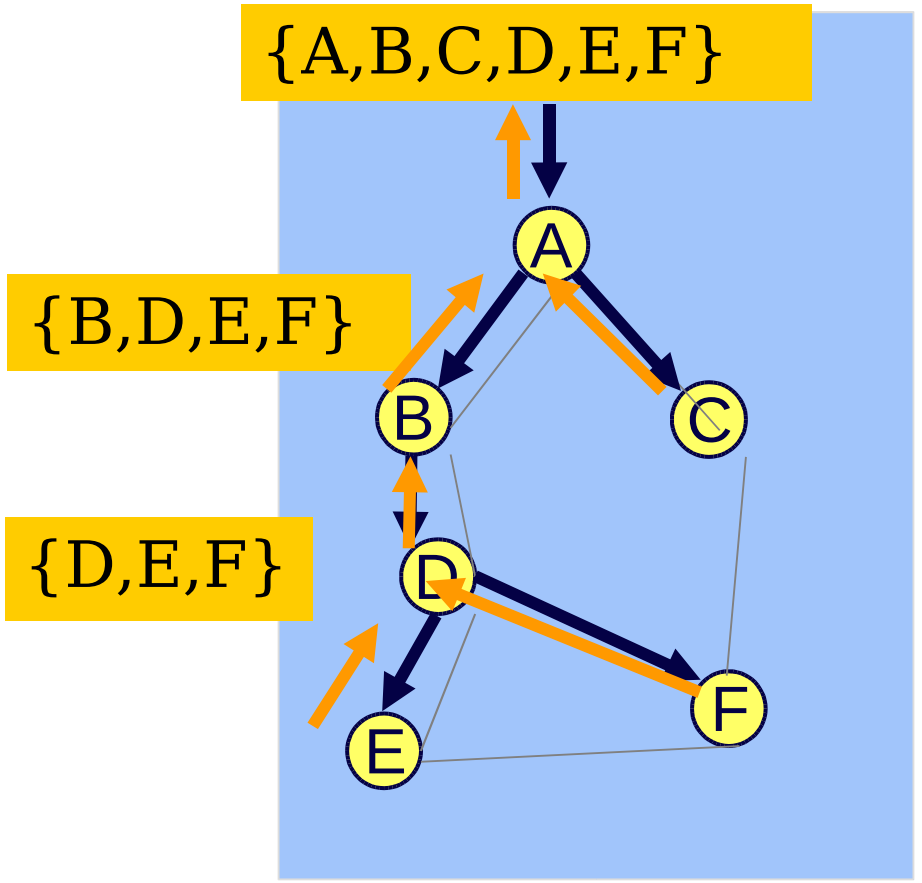
Figure 2: Phases of Power Consumption In TinyDB

- **Snoozing:** waiting for timer to expire or external event to wake up device
- **Processing:** when device wakes up. Here query results are generated locally
- **Processing & Receiving:** Results are collected from neighbors over radio
- **Transmitting:** Results for the query are delivered by the local mote

Communication in SN

- Multi-hop communication (where intermediate nodes relay information to peers)
- On Mica motes all communication is broadcast
- OS provides software filters to address messages to particular node
- Link-level acknowledgement. No end-end acknowledgement
- Communication topologies are automatically discovered (ad-hoc)

Execution of a query in TinyDB



Written in SQL-Like Language With Extensions For :

- Sample rate
- Offline delivery
- Temporal Aggregation

Period of time between sample intervals is called an **EPOCH**. They provide a convenient mechanism for structuring computation to minimize power consumption.

Basic language features

- SELECT nodeid, light, temp FROM sensors SAMPLE INTERVAL 1s for 10s
- SELECT nodeid, light FROM sensors WHERE light > 400 EPOCH DURATION 1s

Epoch	Nodeid	Light	Temp	Accel	Sound
0	1	455	x	x	x
0	2	389	x	x	x
1	1	422	x	x	x
1	2	405	x	x	x

- sensors table is (conceptually) an unbounded, continuous data stream of values

Landmark query

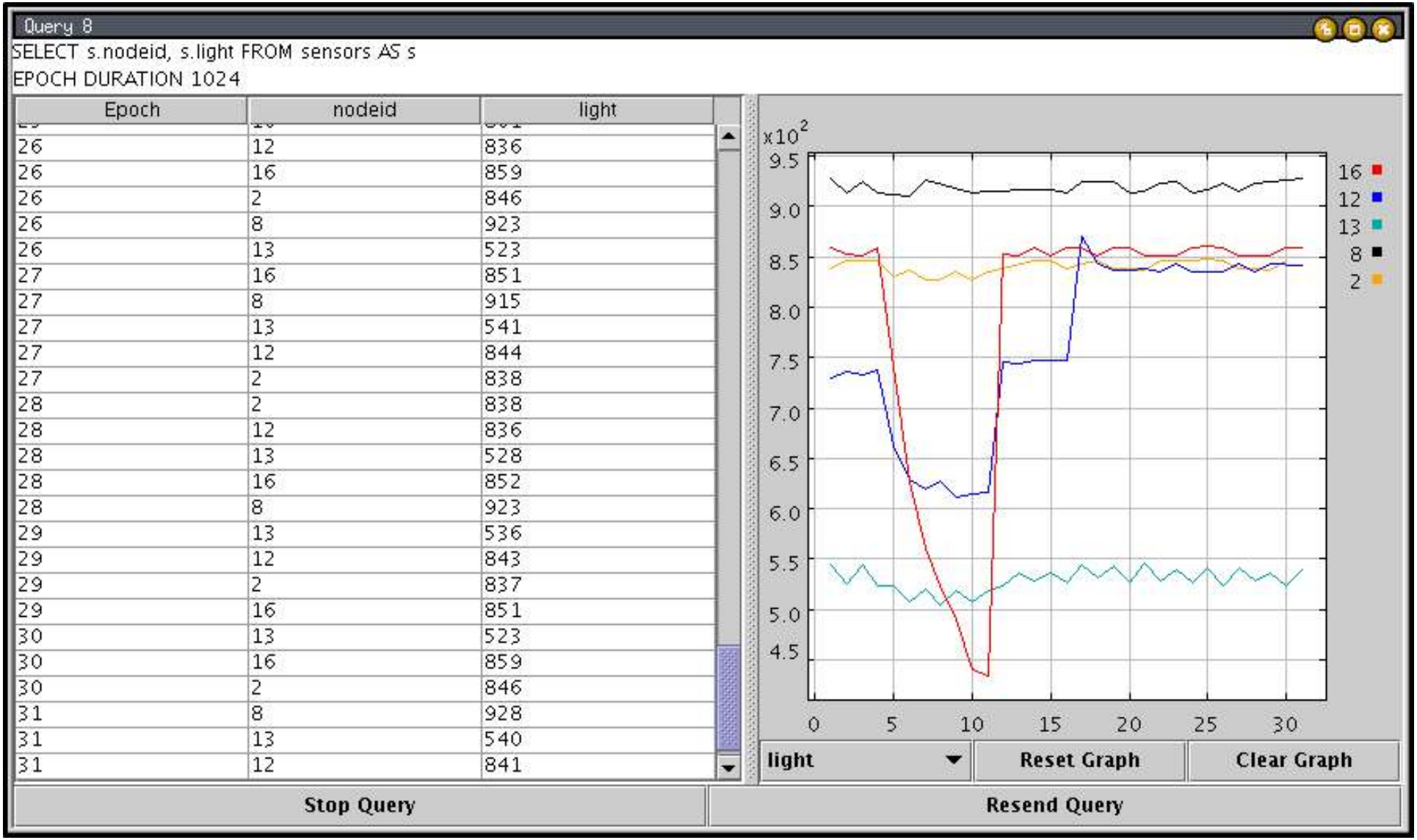
- Nodes can store intermediate query results much like **materialized views** in RDBMS
  - Sensors push data to view nodes where interactive queries pull them or push them to other view nodes or a base station.
- SELECT COUNT(*) FROM sensors AS s, recentLight AS rl where rl.nodeid = s.nodeid AND s.light < rl.light SAMPLE INTERVAL 10s
  - If storage point (a TABLE such as recentLight that is created as a result of a streaming query; the creation of the storage point hasn't been shown) and outer query deliver data at different rates, a rate matching construct is provided (If outer query is faster or if inner query is faster!)
- Landmark query is common in streaming systems.

Sliding window query

- Aggregation (min, max, avg, count…) have the property that it reduces the quantity of data that must be transmitted through the network and hence they are widely used in SN.
- SELECT WINAVG(volume, 30s, 5s) FROM sensors SAMPLE INTERVAL 1s
  - Will report avg. volume over last 30sec once every 5sec sampling 1/sec.
  - Could be used to detect occupancy in a conference room!

TinyDB screenshot

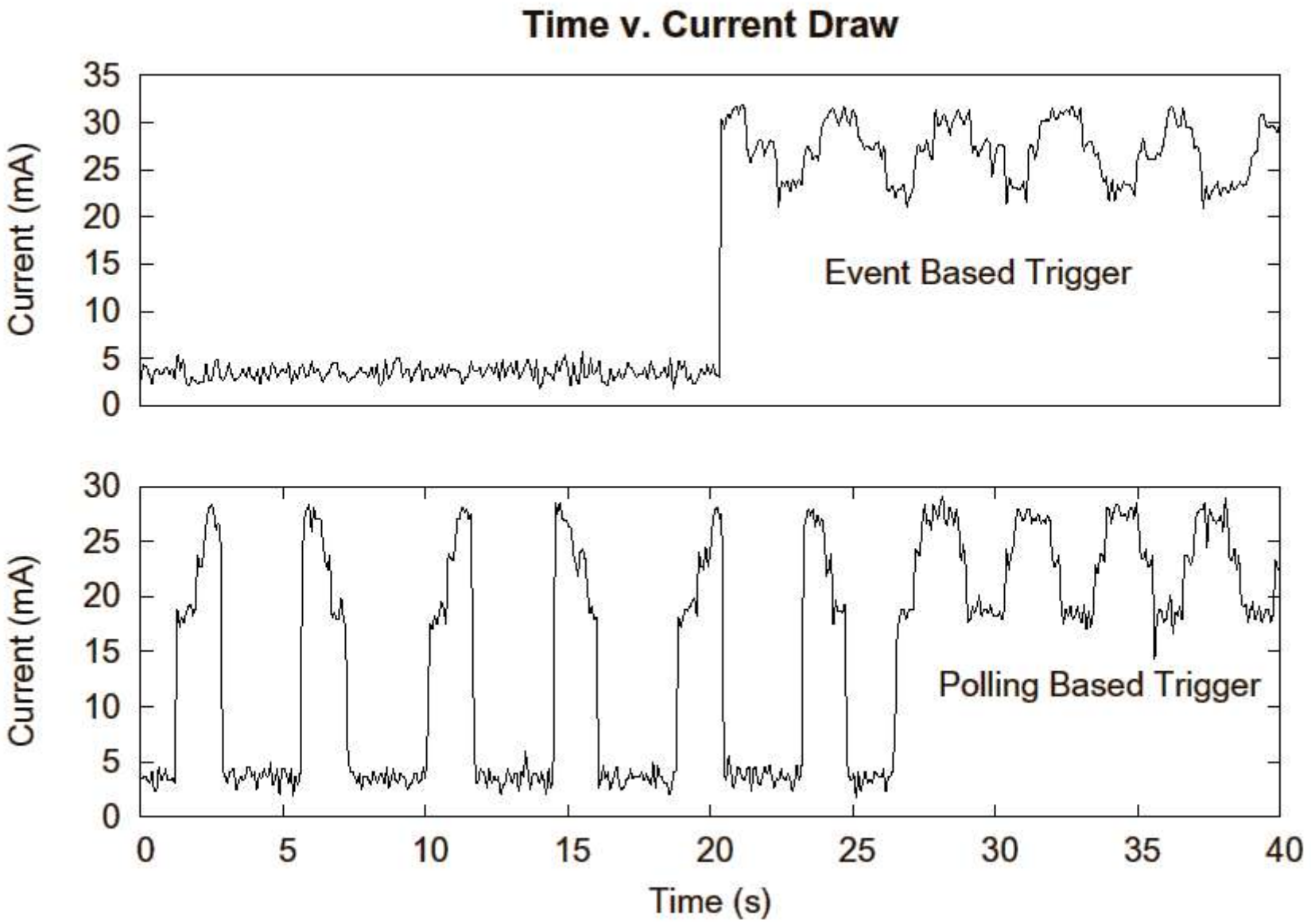
(Not in the paper! Papers would be so much more interesting if authors put in screenshots!)



Event-based queries

- Events in TinyDB are generated by **another query or by the OS**
- ON EVENT bird_detect(loc): SELECT AVG(light), event.loc FROM sensors AS s where dist(s.loc, event.loc) < 10m SAMPLE INTERVAL 2s FOR 30s
  - Every time bird_detect event occurs, the query is issued for detecting the node and the avg light are collected from nearby nodes every 2sec for 30sec.
- Events **allow the system to be dormant** until some external condition occurs, instead of continually polling/blocking waiting for data to arrive.
- Events can also serve as **stopping condition for queries**. STOP ON EVENT(p) WHERE cond(p)

Events can significantly reduce power consumption!



Lifetime-based queries

- Specifying lifetime is a much more intuitive way for users to reason about power consumption
- SELECT nodeid, accel FROM sensors LIFETIME 30 days
  - This query specifies that the network should run for atleast 30 days sampling light and acceleration sensors **at a rate that is as quick as possible** and **still satisfies this goal**
- To satisfy a lifetime clause, TinyDB performs lifetime estimation

Lifetime estimation

- The goal of lifetime estimation is to compute a sampling and transmission rate given a number of joules remaining

Parameter	Description	Units
l	Query lifetime goal	hours
c_{rem}	Remaining Battery Capacity	Joules
E_n	Energy to sample sensor n	Joules
E_{trans}	Energy to transmit a single sample	Joules
E_{rcv}	Energy to receive a message	Joules
σ	Selectivity of selection predicate	
C	# of children routing through node	

Table 1: Parameters used in lifetime estimation

The first step is to determine the available power p_h per hour:

$p_h = c_{rem} \ / \ l$

We then need to compute the energy to collect and transmit one sample, e_s, including the costs to forward data for our children:

$e_s = (\sum_{s=0}^{numSensors} E_s) + (E_{rcv} + E_{trans}) \times C + E_{trans} \times \sigma$

Finally, we can compute the maximum transmission rate, T (in samples per hour), as :

$T = p_h / e_s$

Lifetime estimation

- Since sensors need to sleep between relaying of samples, it is important that senders and receivers synchronize their wake cycles.
  - So, nodes are allowed to transmit when their parents in the routing tree are awake and listening
- TinyDB also allows the user to monitor physical phenomenon at a particular granularity using the optional MIN SAMPLE RATE r clause

Power-Aware Optimization

- TinyDB a simple cost-based optimizer to choose a query plan that will yield the lowest overall power consumption
- A key aspect of AQCP is
  - Cost of a particular plan is often dominated by the cost of sampling the physical sensors and transmitting query results **rather than** the cost of applying individual operators

Power-aware optimization

- SELECT accel, mag FROM sensors WHERE accel > c1 AND mag > c2 SAMPLE INTERVAL 1s
  - Case 1: Magnetometer and accelerometer are sampled before either selection is applied
  - Case 2: Magnetometer (Smag) is sampled and the selection over its reading is applied before the accelerometer is sampeled or filtered
  - Case 3: Accelerometer (Saccel) is sampled first ...
- Case 1 was most expensive
- Case 2 more expensive than Case 3 when Saccel is much more selective than Smag
- When Smag is highly selective, it can be cheaper to sample the magnetometer first

Exemplary aggregate pushdown

```
SELECT WINMAX(light,8s,8s)
FROM sensors
WHERE mag > x
SAMPLE INTERVAL 1s
```

Unless > x is very selective, correct ordering is:

- **Sample light**
- **Check if it's the maximum**
- **If it is:**
  - **Sample mag**
  - **Check predicate**
  - **If satisfied, update maximum**

Event query batching to conserve power

```
ON EVENT E(nodeid)
SELECT a
FROM sensors AS s
WHERE s.nodeid = e.nodeid
SAMPLE INTERVAL d FOR k
```

- This query will cause an instance of the internal SELECT... to be started every time the event e occurs. The internal query samples results every d seconds for a duration of k seconds, at which point it stops running
- It is possible for multiple instances of the internal query to be running at the same time
- In such a situation, event based queries will be outweighed by the fact that each instance of the query consumes significant enery, sampling and delivering (independent) results.

Event query batching to conserve power

- So to prevent multiple copies from running, we employ a multi-query optimization technique based on rewriting!

We convert external events (of type e) into a stream of events, and rewrite the entire set of independent internal queries as a sliding window joing between events and sensors, with a window size of k seconds on the event stream, and no window on the sensor stream

```
SELECT s.a FROM sensors AS s, events AS e
WHERE s.nodeid = e.nodeid
AND e.type = E AND s.time – e.time < k AND s.time > e.time
SAMPLE INTERVAL d
```

- **Advantage: Only one query runs at a time no matter how frequently the events of type e are triggered**

Power sensitive dissemination and Routing

- Dissemination begins with a broadcast of the query from the root of the network
- As each sensor hears query, it decides if the query applies locally and/or needs to be broadcast to its children in routing tree
- If query does not apply at a node, and node does not have any children, then the entire subtree rooted at that node can be excluded from the query s**aving the costs** of **disseminating, executing and forwarding results** for the query across several nodes!

Challenge

- Is to determine when a node or its children **need not participate** in a particular query
  - Common situation: With constant valued attributed (nodeid or location) with a selection predicate that indicates the node need not participate
  - Similarly: If node knows none of its children will **ever satisfy** the value of selection predicate, it need not forward query down the routing tree
- To maintain information about child attribute values, authors proposed **semantic routing tree** (SRT)

Semantic Routing Trees

- An SRT is a routing tree designed to allow **each node to efficiently** determine if any of the nodes below it will need to participate in a given query over some constant attribute A
- Conceptually, an SRT is an index over A that can be used to locate nodes that have data relevant to the query
- Each **node stores a single unidimensional interval** representing the range of A values beneath each of its children
- When a query q with a predicate A arrives at node n, n checks to see if any child's value of A overlaps the query range of A in q. If yes -> Receive results and forward query
- If query applies locally, n begins executing the query itself

Building an SRT

- First the **SRT build request is flooded down** the network which includes the name of the attribute A over which the tree should be built.
- Node n may **have several possible choices of parent,** since many nodes in radio range may be closer to the root.
- If n has children it forwards the request to them, else it **chooses a node p from available parents to be its parent**, and then reports value of A to p in parent selection message
- If n has children it **chooses parent and sends a selection message** indicating the range of values of A which it and its descendents cover

Maintaining SRTs

- Node appearance and Link quality can both require a node to switch parents
  - To do this, node sends a parent selection message to its new parent, n
  - If message changes the range of n's interval, it notifies its parent and in this way updates can propagate to the root of the tree
- The authors also propose a technique to handle the disappearance of a child node

Disappearance of a child node

- Parents associate active_query_id and last_epoch with every child in the SRT
- When parent p forwards query q to child c, it sets childs active_query_id = id_of_q and last_epoch = 0
- When p forwards or aggregates a result for q from c, it updates c's last_epoch = epoch on which result was received
- If **p does not hear c for some number of epochs t**, it assumes c has moved away and removes its SRT entry
  - Then, p asks remaining children to transmit their ranges
  - It uses this information to construct new interval

SRT parent selection

- **Random approach**
  - Node picks a random parent
- **Closest parent approach**
  - Each parent reports the value of its index attribute with the SRT-build request, and children pick the parent whose attribute value is closest to their own
- **Clustered approach**
  - Same as above, except, if a node hears a sibling node send a parent selection message, it snoops on the message to determine its siblings parent and value and then picks its own parent
  - Authors provide values and say this approach is the most superior

Query Execution

- Query execution consists of a simple sequence of operations at each node during every epoch
  - Nodes sleep for as much of each epoch as possible
  - They wake up to sample sensors and relay and deliver results
  - All nodes sleep and wakeup at the same time to ensure that results will not be lost as a result of a parent sleeping when a child tries to propagate a message
- The time t_awake that a sensor node must be awake to do this is largely dependent on the number of nodes transmitting in the radio cell (authors have omitted the details)

Prioritizing data delivery

- After results have been sampled and local operators have been applied, they are enqueued onto a radio queue for delivery to the node's parent
- There are situations when this queue can overflow.
  - ACQP systems have the ability to make runtime decisions about the value of an individual item.
- So the system must be able to discard some type, or combine two tuples via some aggregation policy!

Prioritization schemes

- Naïve
  - No tuple is considered more valuable than the other, so tuples are dropped if they don't fit into the queue
- Winavg
  - Instead of dropping the two results at the head of the queue are averaged, to make room for new results
- Delta
  - A tuple is assigned an initial score relative to its difference from the most recent value successfully transmitted from this node.
  - At each point in time, the tuple with the highest score is delivered.
  - The tuple with the lowest score is evicted when the queue overflows

Results in the Prioritization schemes

- The authors say
  - Delta is considerably closer in shape to the original signal as it tends to emphasize extremes
  - Winavg tends to dampen the extremes

Summary of the paper/what did we see!

- We saw the novel issues and techniques that arise when taking an acquisitional perspective on query processing
- We saw event and lifetime clauses in the query language
- We then saw query optimization after which we saw query dissemination using SRTs
- Finally we saw the need to prioritize data

Any questions?