Data Stream Management Systems
- Sensor Networks -

Vera Goebel, Thomas Plagemann
Department of Informatics, University of Oslo

* with slides from Ellen Munthe-Kaas (INF5100)

DSMS - Part 2

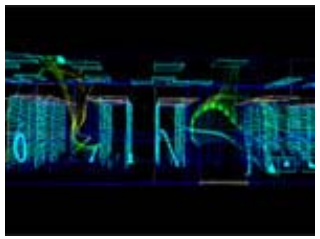
- Sensor networks
- Example 2:
  - DSMS for sensor networks
  - Aurora & Medusa
  - TinyDB
- Summary

Some Sensornet Applications

ZebraNet

Redwood forest
microclimate monitoring

Smart cooling in data centers



http://www.hpl.hp.com/research/dca/smart_cooling/

Environmental Applications

- Tracking the movements of birds, animals, insects
- Monitoring environmental conditions that affect crops and livestock
- Chemical/biological detection
- Biological, earth, and environmental monitoring in marine, soil, and atmospheric contexts
- Meteorological or geophysical research
- Pollution study, precision agriculture, irrigation
- Biocomplexity mapping of environment
- Flood detection, forest fire detection

Health Applications

- Integrated patient monitoring
- Telemonitoring of human physiological data
- Tracking and monitoring doctors and patients inside a hospital
- Tracking and monitoring patients and rescue personnel during rescue operations

Military Applications

- Monitoring friendly forces, equipment and ammunition
- Battlefield surveillance
- Reconnaissance of opposing forces and terrain
- Nuclear, biological and chemical (NBC) attack detection and reconnaissance

Commercial Applications

- Monitoring product quality
- Constructing smart office spaces
- Interactive toys
- Smart structures with sensor nodes embedded inside
- Machine diagnostics
- Interactive museums
- Managing inventory control
- Environmental control in office buildings
- Detecting and monitoring car thefts
- Vehicle tracking and detection

Application Examples

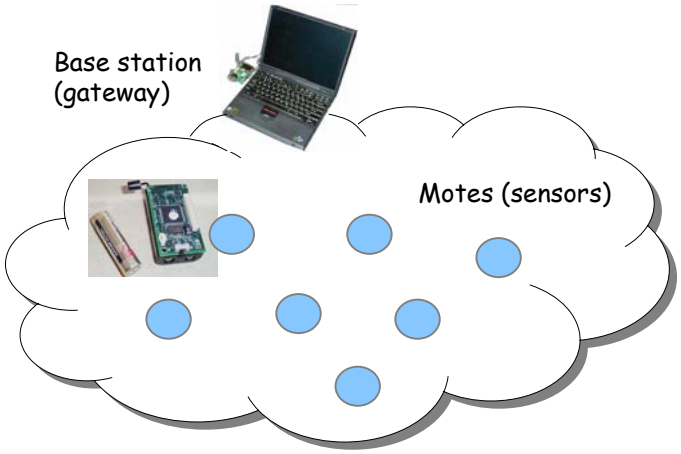
Habitat Monitoring: Storm petrels on Great Duck Island, microclimates on James Reserve.

Earthquake monitoring in shake-test sites.

Vehicle detection: sensors along a road, collect data about passing vehicles.

Traditional monitoring apparatus.

Sensor Networks



Sensor networks communication architecture

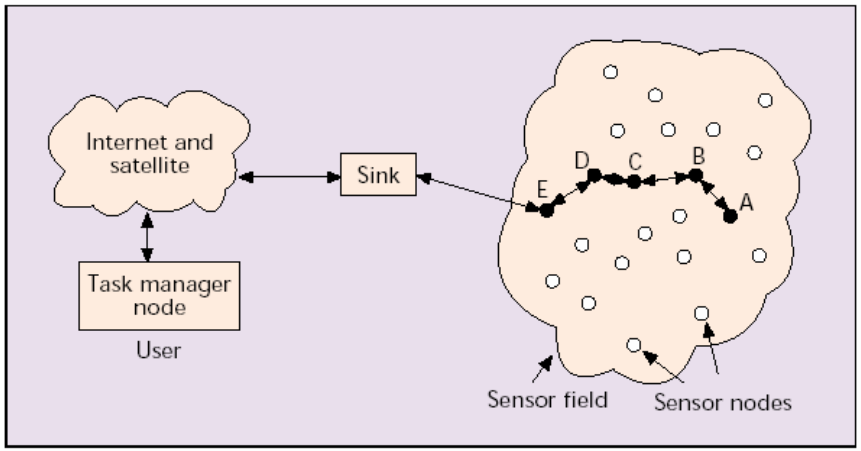


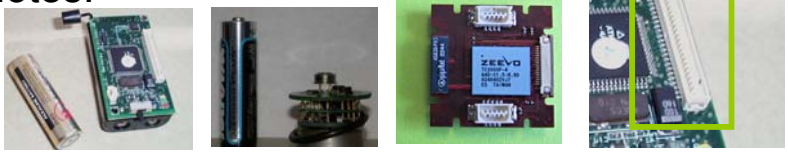
Figure 1. *Sensor nodes scattered in a sensor field.*

Sensor Network Characteristics

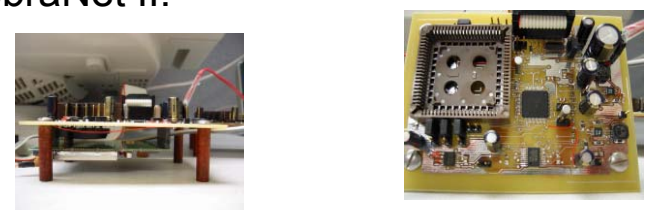
- Autonomous nodes
  - Small, low-cost, low-power, multifunctional
  - Sensing, data processing, and communicating components
- Sensor network is composed of large number of sensor nodes
  - Proximity to physical phenomena
    - Deployed inside the phenomenon or very close to it
- Monitoring and collecting physical data
- No human interaction for weeks or months at a time
  - Long-term, low-power nature

Sensor Hardware (cont.)

- Motes:



- ZebraNet II:



Motes

Mote: Short for *remote*. Refers to a wireless transceiver that is also a remote sensor

Mica2 mote with 2 AA batteries (provide power for one year's use)

Mica2DOT mote.
Powered with button battery

Spec smart dust; total size 5 mm^2

Sensor Hardware

- A sensor node is made up of four basic components
  - Sensing unit
    - usually composed of two subunits: sensors and analog to digital converters (ADCs).
  - Processing unit,
    - Manages the procedures that make the sensor node collaborate with the other nodes to carry out the assigned sensing tasks.
  - Transceiver unit
    - Connects the node to the network.
  - Power units (the most important unit)
- Matchbox-sized module
  - consume extremely low power,
  - operate in high volumetric densities,
  - have low production cost and be dispensable,
  - be autonomous and operate unattended,
  - be adaptive to the environment.

Motes in the DMMS Laboratory

Mica2

- **Processor**: MPR400CB based on Atmel ATmega128L
- **Radio**: 900 MHz multi-channel transceiver
- **Memory**: 4 kB Configuration EEPROM
  128 kB Program Flash Memory
  512 kB Measurement (Serial) Flash
- **Power**: 2 x AA
- **OS**: TinyOS v1.0
- **Weight**: 18g (excluding batteries)
- **Sensors**:
  - Light
  - Temperature
  - Acoustic
- **Actuators**:
  - Sounder

Principles of Sensor Networks

- A large number of low-cost, low-power, multifunctional, and small sensor nodes
- Sensor node consists of sensing, data processing, and communicating components
- A sensor network is composed of a large number of sensor nodes,
  - which are densely deployed either inside the phenomenon or very close to it.
- The position of sensor nodes need not be engineered or pre-determined.
  - sensor network protocols and algorithms must possess self-organizing capabilities.

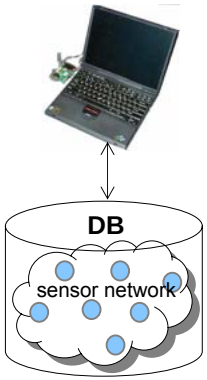
Motes vs. Traditional Computing

- Embedded OS
- Lossy, ad hoc radio communication
- Sensing hardware
- Severe power constraints

Managing Data

- **Purpose of sensor network:** Obtain real-world data
  - Extract and combine data from the network
- **But:** Programming sensor networks is hard!
  - Months of lifetime required from small batteries
  - Lossy, low-bandwidth, short range communication
  - Highly distributed environment
  - Application development
  - Application deployment administration

Data Management Systems for Sensor Networks



Data Management Systems for Sensor Networks

- Motivation:
  - Implement data access
    - Sensor tasking
    - Data processing
    - Possibly support for data model and query language
- Goals:
  - Adaptive
    - Network conditions
    - Varying/unplanned stimuli
  - Energy efficient
    - In-network processing
    - Flexible tasking
    - Duty cycling

Data Management System Challenges

- Routing
- Resource allocation
- Deployment
- Query language, query optimization

DSMS for Sensor Networks

- Aurora & Medusa System
  - System model
  - Workflows
  - Operators
  - Scheduling and QoS
  - Query model
  - Architecture
  - Example application
- TinyDB
  - Architecture
  - Data model
  - Query language
  - Resource efficiency

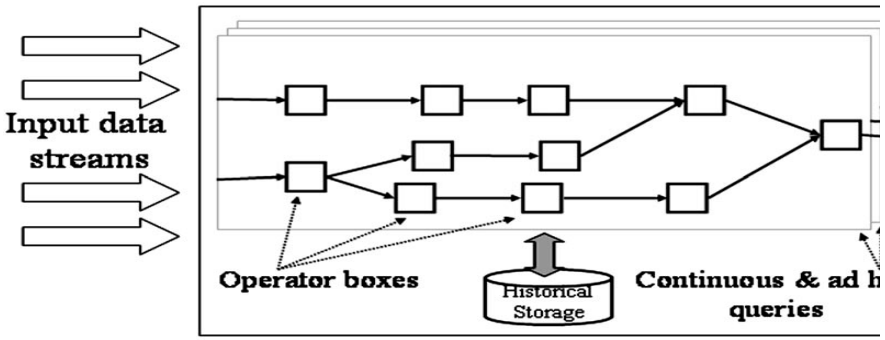
Aurora & Medusa

- Aurora: single-site high performance stream processing engine
- Aurora*: connecting multiple Aurora workflows in a distributed environment
- Medusa: distributed environment where hosts belong to different organizations and no common QoS notion is feasable

Important Aspects of Aurora

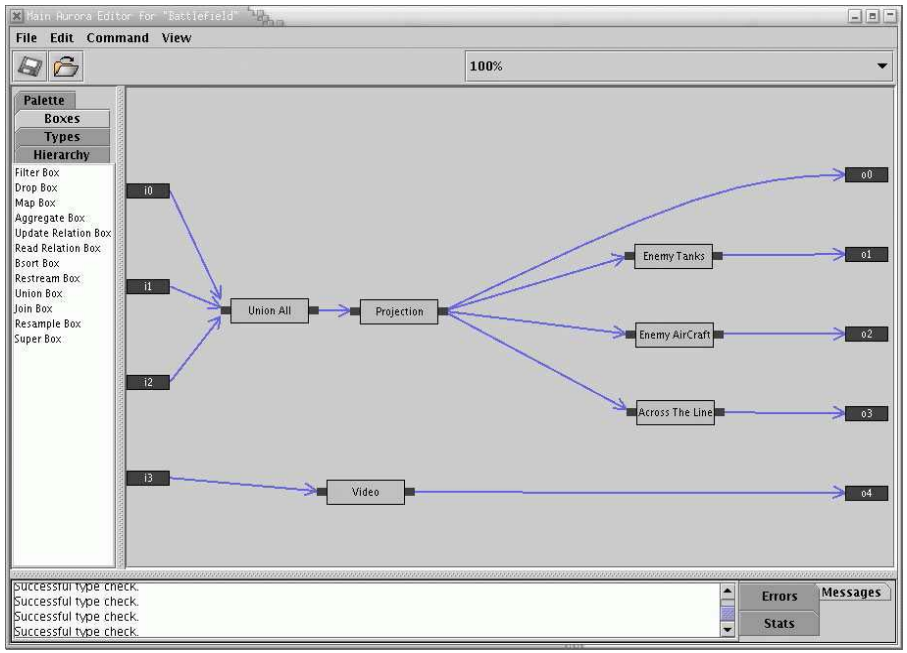
- Workflow orientation
- Operators
- Scheduler
- Quality of Service
- Optimizations

Aurora System Model



Aurora Workflows

- Two reasons to build Aurora as a workflow system:
  - Most monitoring applications contain a component that performs sensor fusion or data cleansing
    - This can be part of DSMS or a different sub-system (less control and more boundary crossings)
  - Query optimization: application designers locate the correct place in workflow diagram to add needed functionality ⇒ reduces complexity of query optimization



Aurora Operators

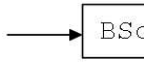
- Windowed operations have timeout capability
- Handling out-of-order messages
- Extendability
- Resample operator

Aurora Operators (cont.)

Filter ($p_1, \ldots, p_m$) (S)

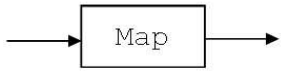
Selection: route tuples to output with 1st predicate match

BSort (Using



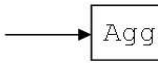
Bounded-Pass
Number of pas

Map ($A_1=f_1, \ldots, A_m=f_m$) (S)



Mapping: Map input tuples to outputs with m fields

Aggregate (F,



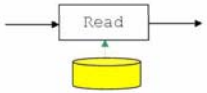
Sliding Windov
Slide by i betw

Aurora Operators (cont.)

Resample (F, Size s, Left Assume O_1, Right Assume O_2) (S_1,S_2)

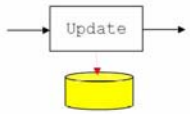
Interpolation: Interpolate missing points in S_2 as determined with S_1

Read (Assume O, SQL Q) (S)



SQL Read: Apply SQL query Q for each input in S. Output as stream.

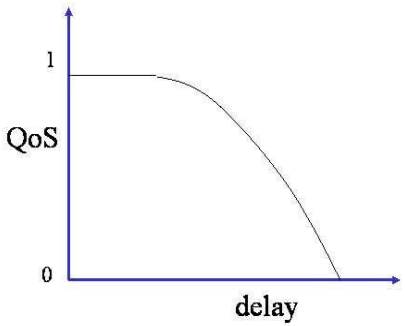
Update (Assume O, SQL U, Report t) (S)



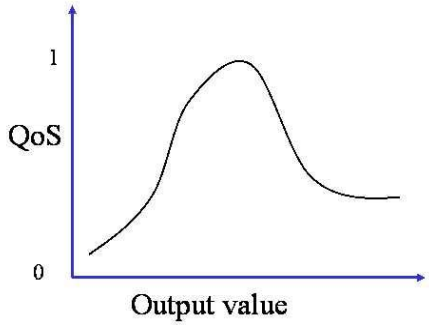
SQL Write: Apply SQL update query U for each input in S. Report t for each update.

Aurora: Quality of Service

- Basic idea: use human specified QoS graphs at Aurora output

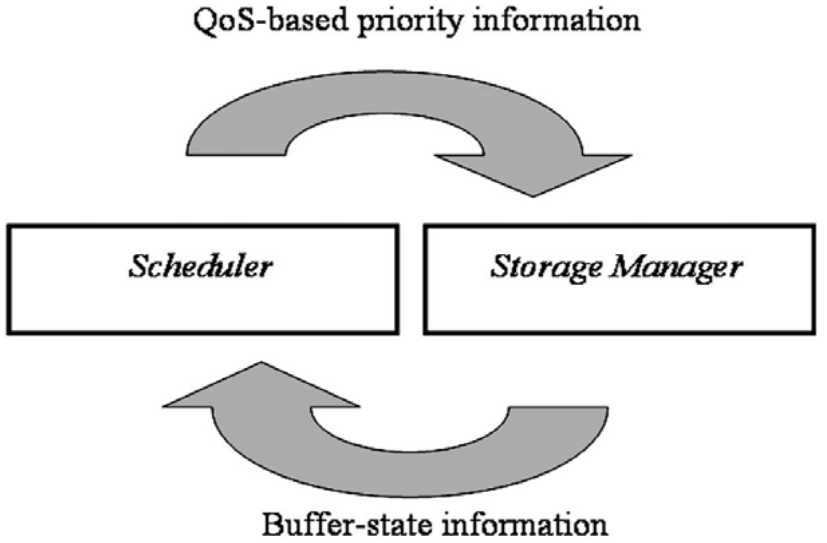


Drop based

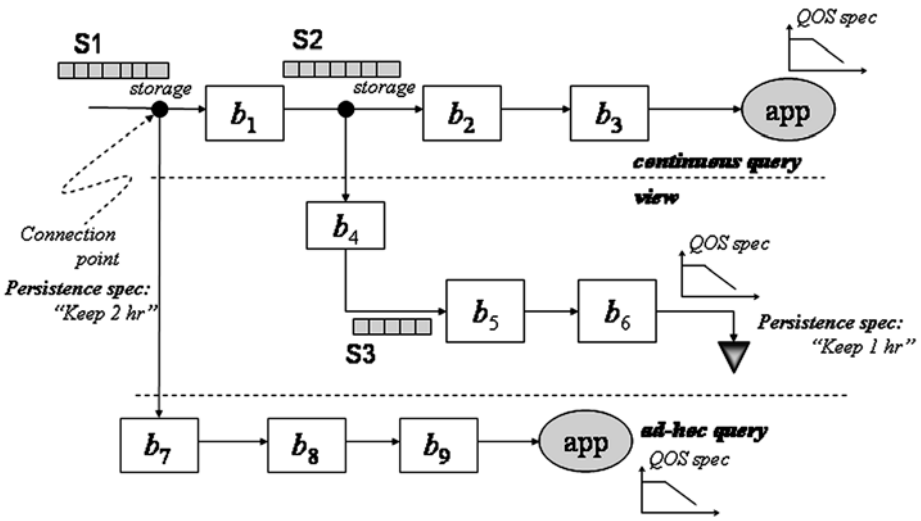


Value-based

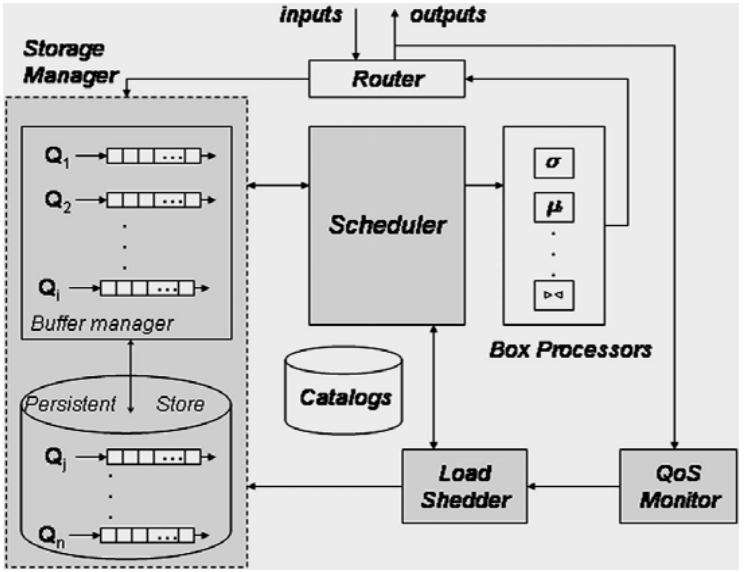
Aurora: Scheduler ↔ Storage



Aurora Query Model

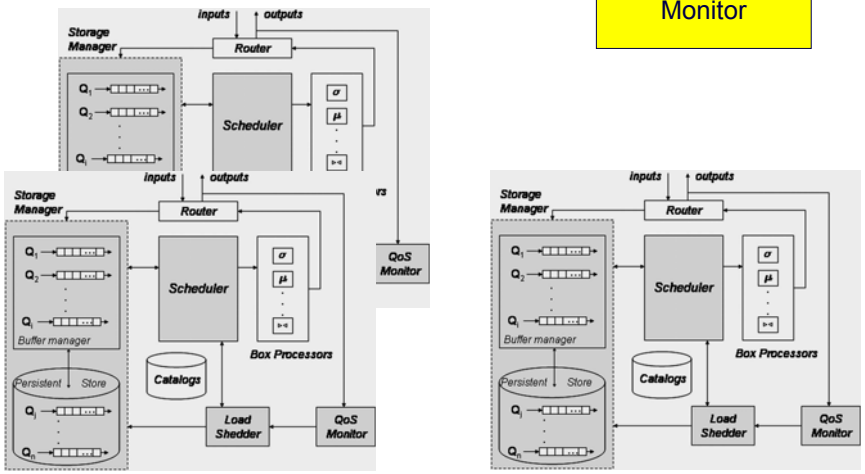


Aurora Architecture



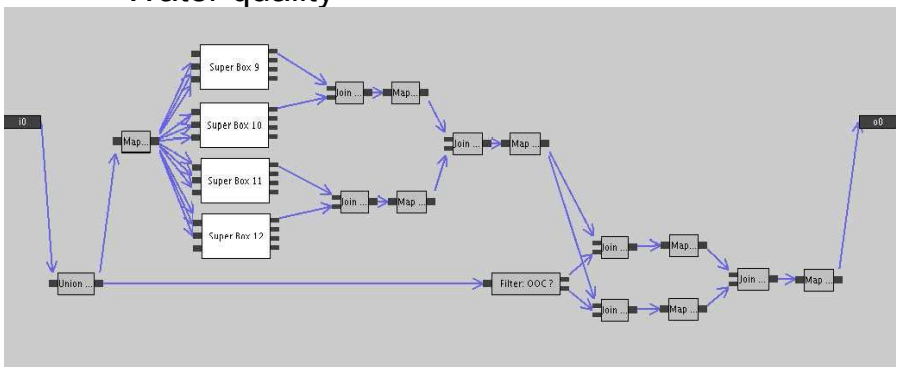
Aurora* Architecture

External QoS Monitor

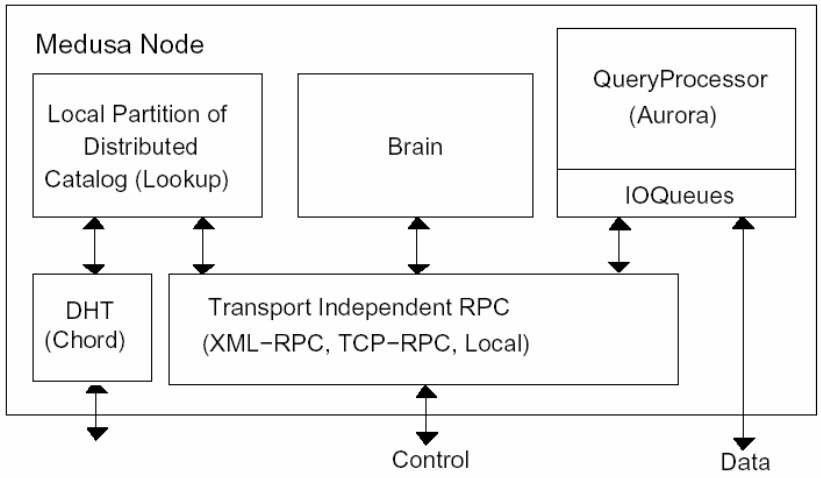


Example: Enviromental Monitoring

- Monitoring toxins in the water
  - Fish behaviour
  - Water quality



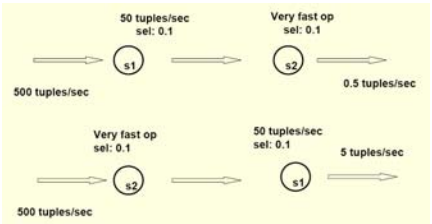
Medusa



Optimization Objectives

- Rate-based optimization:
  - Take into account the rates of the streams in the query evaluation tree during optimization
  - Rates can be known and/or estimated
- Maximize tuple output rate for a query
  - Instead of seeking the least cost plan, seek the plan with the highest tuple output rate

Rate Based Optimization – I



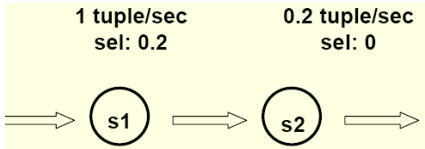
- Output rate of a plan: number of tuples produced per unit time
- Derive expressions for the rate of each operator
- Combine expressions to derive expression r(t) for the plan output rate as a function of time:
  - Optimize for a specific point in time in the execution
  - Optimize for the output production size

Rate Based Optimization – II

- Optimize for resource (memory) consumption
- A query plan consists of interacting operators, with each tuple passing through a sequence of operators
- When streams are bursty, tuple backlog between operators may increase, affecting memory requirements
- Goal: scheduling policies that minimize resource consumption

Operator Scheduling

- When tuple arrival rate is uniform:
  - a simple FIFO scheduling policy suffices
  - let each tuple flow through the relevant operators



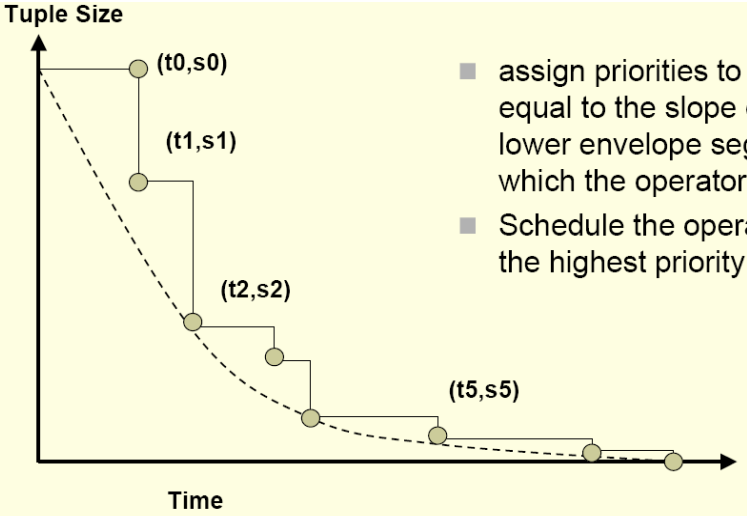
Time	Greedy	FIFO
0	1	1
1	1.2	1.2
2	1.4	2.0
3	1.6	2.2
4	1.8	3.0

Average arrival rate: 0.5 tuples/sec

FIFO: tuples processed in arrival order

Greedy: if tuple before s1 schedule it; otherwise process tuples before s2

Progress Chart: Chain Scheduling



- assign priorities to operators equal to the slope of the lower envelope segment to which the operator belongs
- Schedule the operator with the highest priority

QoS Based Optimization

- Query and operator scheduling based on QoS requirements
- Two-level scheduling policy:
  - Operator batching: superbox selection, superbox traversal based on avg throughput, avg latency, minimizing memory
  - Tuple batching

Optimization Objectives

- Multi-way join techniques proposed:
  - start with a fixed plan
  - moderately adjust it as tuples arrive
- Eddies framework for adaptive query optimization:
  - Continuously adapt the evaluation order as tuples arrive

Load Shedding

- When input stream rate exceeds system capacity a stream manager can shed load (tuples)
- Load shedding affects queries and their answers
- Introducing load shedding in a data stream manager is a challenging problem
- Random and semantic load shedding

Load Shedding in Aurora

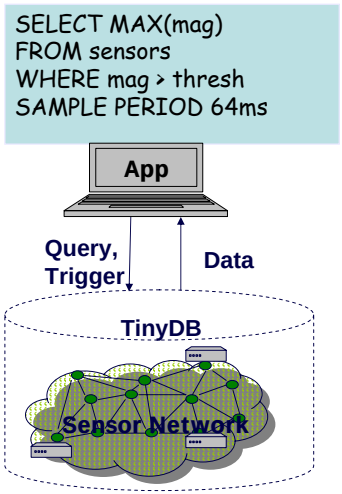
- QoS for each application as a function relating output to its utility
  - Delay based, drop based, value based
- Techniques for introducing load shedding operators in a plan such that QoS is disrupted the least
  - Determining when, where and how much load to shed

TinyDB

- Developed as public-domain system at UC Berkeley
- Literature: Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, **TinyDB: An Acquisitional Query Processing System for Sensor Networks**, *ACM Transactions on Database Systems (TODS),* Volume 30, Issue 1, 2005, available through the ACM Digital Library: http://x-port.uio.no)
- Latest released with TinyOS 1.1 (9/03)
  - Install the task-tinydb package in TinyOS 1.1 distribution
  - First release in TinyOS 1.0 (9/02)
  - Widely used by research groups as well as industry pilot projects
- Successful deployments in Intel Berkeley Lab and redwood trees at UC Botanical Garden
  - Largest deployment: ~80 weather station nodes
  - Network longevity: 4-5 months

TinyDB

- High level abstraction:
  - Data centric programming
  - Interact with sensor network as a whole
  - Extensible framework
- Under the hood:
  - Intelligent query processing: query optimization, power efficient execution
  - Fault Mitigation: automatically introduce redundancy, avoid problem areas



[Source: Sam Madden]

Feature Overview

- Declarative SQL-like query interface
- Metadata catalog management
- Multiple concurrent queries
- Network monitoring (via queries)
- In-network, distributed query processing
- Extensible framework for attributes, commands and aggregates
- In-network, persistent storage

[Source: Sam Madden]

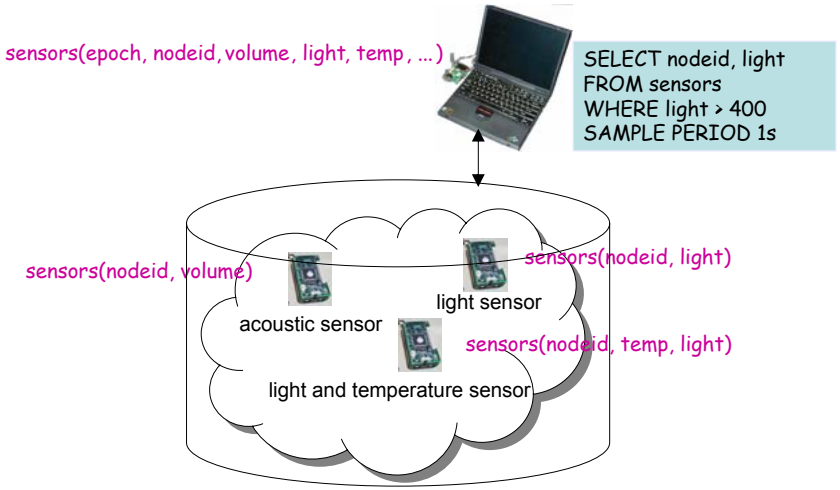
Query Language Essentials

- Declarative queries
  - Simple, SQL-like queries
  - Users specify the data they want and the rate at which data should be refreshed
  - Using predicates, not specific addresses
- TinyDB collects data from motes in the environment, filters it, aggregates it, and routes it out to a PC
- TinyDB does this with power-efficient in-network processing algorithms

Data Model

- Relational model
- Single table **sensors**
  - One column (attribute) per type of value that a device can produce (light, temperature,...)
  - One row (record) per node per instant in time
  - Physically partitioned across all nodes in the network
  - Records are materialized only at need and stored only for a short period or delivered directly to the network
  - Projections and transformations of tuples from **sensors** may be stored in materialization points

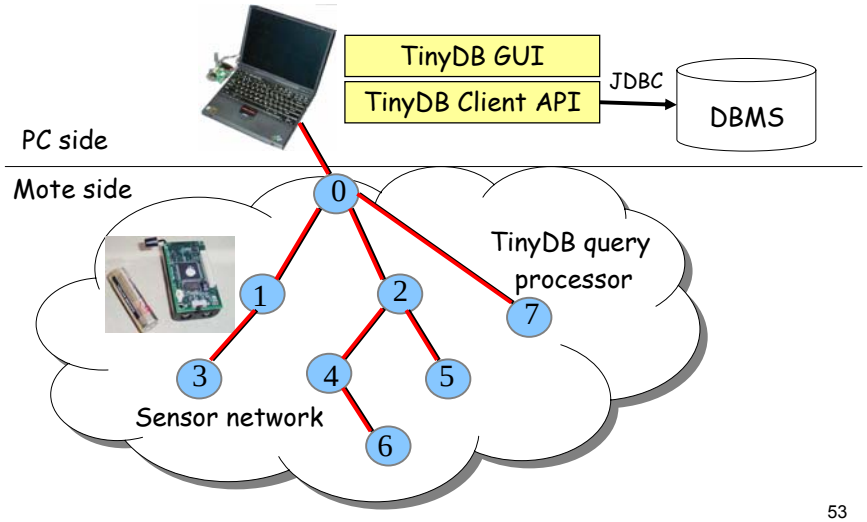
The *sensors* Table



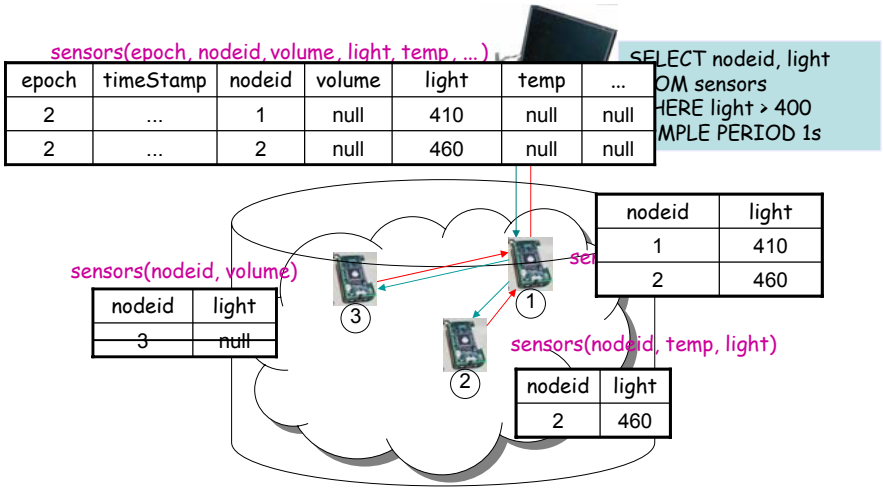
Metadata Management

- Each node maintains a **metadata catalog** containing
  - local attributes
    - name
    - cost: power, sample time
  - events
    - name
    - signature
    - cost: frequency estimate
  - user-defined functions and predicates
- Periodically copied to root for use by query optimizer
- Registered via static linking at compile time using nesC

TinyDB Architecture (Routing Tree)



TinySQL Routing Example



Routing

- A routing tree is established
  - Root is a gateway
  - Spanning tree
  - No multiple paths
  - Tree construction and maintenance
- Query fragments are disseminated down the routing tree
  - Query optimization performed centrally, outside the sensor network, using metadata obtained from the nodes
  - Semantic routing tree to avoid flooding

Routing Tree Creation

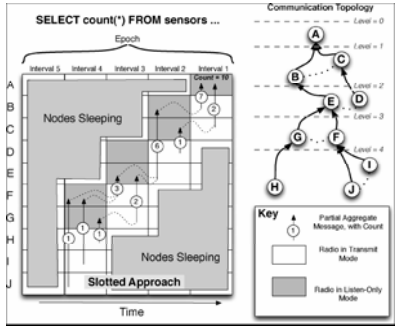
1. One mote is appointed the root (usually used as the interface/gateway of the network)
2. The root broadcasts a message `<ID, distanceFromRoot>` asking motes to organize into a routing tree
3. Any mote without an assigned level that hears this message assign its level as the `received+1` and chooses the sender as its parent through which it will route messages to the root
4. Motes re-broadcast the routing message, inserting their own IDs and levels... and so on

Communication Scheduling

- Data processed up the routing tree
- Sample periods: Sleep; data sampling; receiving; processing; transmitting
  - Sleep period defined based on number of children
    - Awakening just in time to receive results
  - Sampling
    - Expensive!
  - Receiving
  - Processing:
    - Filtering, partial aggregate
  - Network transmission
    - Adaptation to network contention and power consumption

Communication Scheduling

- A mote upon receiving a request to perform a query:
  - awakens
  - synchronizes its clock
  - chooses the sender of the msg as its parent
  - forwards the query, setting the delivery interval for children to be slightly before the time its parent expects to see the partial state record



Sample Period

- Long enough to allow all nodes to report
- Delivery interval length = SamplePeriod/TreeDepth
  - Sets a lower bound to sample period
  - Limits the maximum sample rate of the network
- The sample rate can be increased by pipelining the communication schedule

TinyOS

- Operating system for managing and accessing mote HW
- Characteristics:
  - Energy-efficient
  - Programming model: Components
  - Only one application running at a time
  - No process isolation or scheduling
  - No kernel
  - No protection domains
  - No memory manager
  - No multithreading
- Programming language: nesC

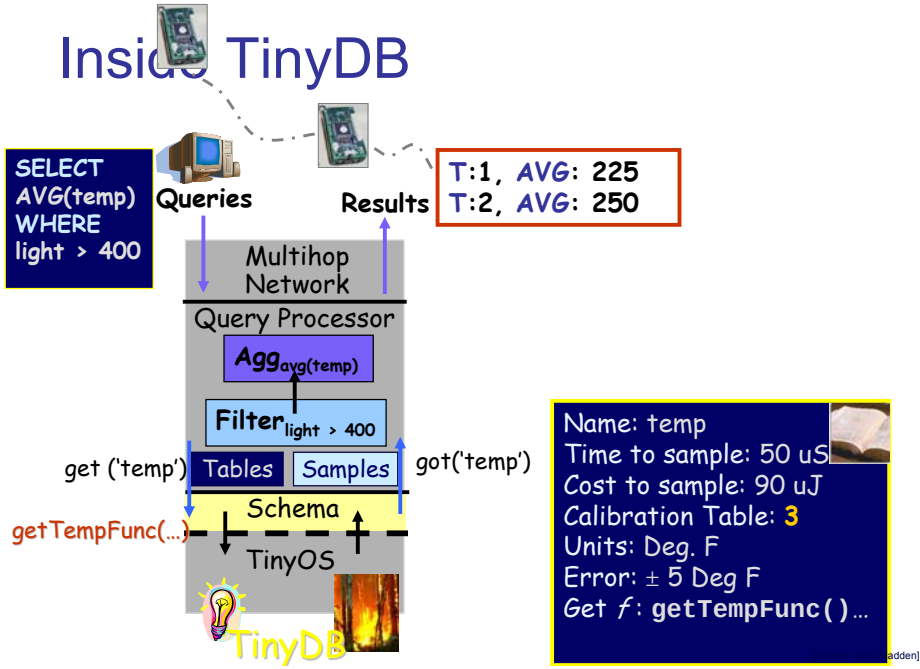
NesC/TinyOS

- NesC: a C dialect for embedded programming
  - Components, "wired together"
  - Quick commands and asynch events
- TinyOS: a set of NesC components
  - hardware components
  - ad-hoc network formation & maintenance
  - time synchronization

Think of the pair as a programming environment

[Source: Sam Madden]

Inside TinyDB



[Source: Sam Madden]

Inside TinyDB (cont.)

- ~10,000 Lines Embedded C Code
- ~5,000 Lines (PC-Side) Java
- ~3200 Bytes RAM (w/ 768 byte heap)
- ~58 kB compiled code
- (3x larger than 2nd largest TinyOS Program)

TinyDB 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
- On server, all attributes are defined in catalog.xml
- Discussion: other alternative data models?

[Source: Sam Madden]

TinySQL

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


[Source: Sam Madden]

TinySQL Examples

"Find the sensors in bright nests."

1

```
SELECT nodeid, nestNo, light
FROM sensors
WHERE light > 400
EPOCH DURATION 1s
```

Sensors

Epoch	Nodeid	nestNo	Light
0	1	17	455
0	2	25	389
1	1	17	422
1	2	25	405

[Source: Sam Madden]

TinySQL Examples (cont.)

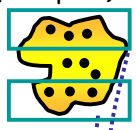
2

```
SELECT AVG(sound)
FROM sensors
EPOCH DURATION 10s
```

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

3

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



Epoch	region	CNT(…)	AVG(…)
0	North	3	360
0	South	3	520
1	North	3	370
1	South	3	520

Regions w/ AVG(sound) > 200

[Source: Sam Madden]

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

[Source: Sam Madden]

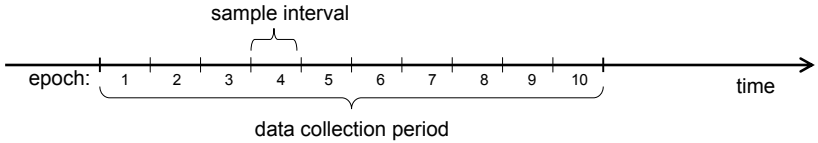
Query over Stored Data

- 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

[Source: Sam Madden]

TinySQL Example 1

- **Sample interval:** Interval during which exactly one tuple of sensors is produced per node for the purpose of executing the query, and during which the query is executed once.
- **Epoch**: Period of time between the start of each sample interval. Numbered consecutively
- **Data collection period**: Period of time over which query is running.

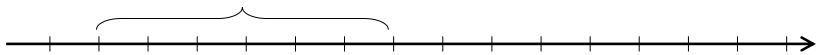


”Report light and temperature readings once per second for 10 seconds.”

```
SELECT nodeid, light, temp
FROM sensors
SAMPLE PERIOD 1 s FOR 10 s
```


TinySQL Example 2

- **Materialization point**: Stored table in the nodes.
  Cf. *materialized views* in traditional RDBSs and *windows* in data stream management systems.



”Store the latest eight light readings, doing one reading every 10 seconds (forever).”

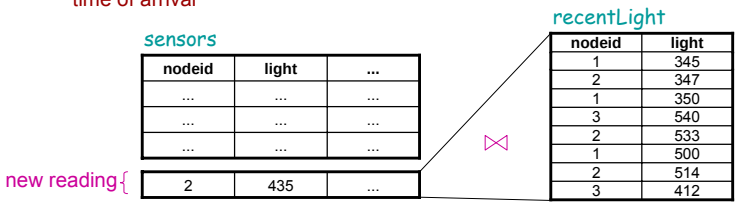
```
CREATE STORAGE POINT recentLight SIZE 8
AS (SELECT nodeid, light
      FROM sensors
      SAMPLE PERIOD 10 s)
```

... later:

```
DROP STORAGE POINT recentLight
```


TinySQL Example 3

- **Joins** are allowed between two materialization points or between a materialization point and the sensors table.
  - New sensors tuples are joined with tuples of the materialization point on their time of arrival



”Count the number of recent light readings (from zero to eight samples in the past) that were brighter than the current reading, each current reading collected during a time span of ten seconds.”

```
SELECT COUNT(*)
FROM sensors AS s, recentLight AS rl
WHERE rl.nodeid = s.nodeid AND s.light < rl.light
SAMPLE PERIOD 10 s
```

TinySQL Example 4

- **Aggregation** can be performed on grouped values as in ordinary SQL. Grouping and aggregation take place over the tuples collected during each sample interval.

”Find the rooms where the average volume is over some threshold (assuming each room can have multiple sensors). Do this every 30 seconds.”

```
SELECT room, AVG(volume)
FROM sensors
WHERE floor = 6
GROUP BY room
HAVING AVG(volume) > threshold
SAMPLE PERIOD 30 s
```

... later:

```
STOP QUERY id
```


TinySQL Example 5

- An **event** can be used for initiating data collection.
  - Generated by another query or by a lower-level part of the OS

”When a bird-detect event occurs, report the average light and temperature levels at sensors near the event’s location. Do this every 2 seconds for a period of 30 seconds (then go to sleep again).”

```
ON EVENT bird-detect(loc):
SELECT event.loc, AVG(light), AVG(temp)
FROM sensors AS s
WHERE dist(s.loc, event.loc) < 10 m
SAMPLE PERIOD 2 s for 30 s
```


TinySQL Example 6

- **Generating an event** from a query:

”Signal the event hot whenever the temperature goes above some threshold. Read the temperature every 10 seconds.”

```
SELECT nodeid, temp
FROM sensors
WHERE temp > threshold
OUTPUT ACTION SIGNAL hot(nodeid, temp)
SAMPLE PERIOD 10 s
```


TinySQL Example 7

- To make sure the network runs for a guaranteed period, users may request a specific query **lifetime**.

”Get the temperature, but space out the readings to make sure that the network will survive at least 30 days.”

```
SELECT nodeid, temp
FROM sensors
LIFETIME 30 days
```

TinySQL Example 8

- **Network health queries** are metaqueries over the network itself.

”Report all sensors whose current battery voltage is less than k.”

```
SELECT nodeid, voltage
FROM sensors
WHERE voltage < k
SAMPLE PERIOD 10 minutes
```


TinySQL Example 9

- **Actuation queries** can be used to perform some physical action in response to a query.

”Turn on the fan if the temperature is rising above a certain level.”

```
SELECT nodeid, temp
FROM sensors
WHERE temp > threshold
OUTPUT ACTION power-on(nodeid)
SAMPLE PERIOD 30 s
```


Conclusions

- Applications of Data Stream Management Systems
  - Near real-time queries and potentially massive data volumes
  - Sensor networks, network monitoring, intrusion detection, ….
- Stream data analysis: rich and largely unexplored field
  - Current research focus in database community: DSMS system architecture, continuous query processing, supporting mechanisms, QoS issues, distribution
- Data mining and OLAP analysis of streams
  - Powerful tools for finding general and unusual patterns
  - Largely unexplored: current studies only touched the surface
- Many challenging technical problems
  - Resource limitations exist, especially at low-level
  - Important to think of the end-to-end architecture
- Some research prototypes are available
  - TelegraphCQ, STREAM, Borealis