Advanced Database Management Systems

Data Stream Management

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Data Streams Defined and Motivated

Data Streams v. Stored Data

Windows on Data Streams

Query Syntax and Semantics in DSMSs

Data Streams (1)
What are they?

- A **stream** is a continuous, potentially unbounded, potentially voluminous, real-time, sequence of data elements (e.g., tuples).
- Often, an item in a stream can be seen as the notification that an event has occurred.
- There two major kinds of sources, viz.:
  - **transactional streams**, in which case an item conveys a notification that an interaction between entities has taken place;
  - **monitoring streams**, in which case an item conveys a notification that some entity has changed (i.e., that its state has evolved).

Data Streams (2)
Why do they matter? (1)

Examples of transactional streams include:

- credit card purchases by consumers from merchants
- phone calls by callers to dialled parties
- web accesses by client of resources held by servers
- inter-organizational interactions (e.g., purchase of supplies, delivery of good, payment for services provided, etc.)
- intra-organizational interactions (e.g., movement of parts from warehouses to production lines, from production lines to delivery vehicles, from vehicles to loading bays, from loading bays to shelves, etc.)

Data Streams (3)
Why do they matter? (2)

Examples of monitoring streams include:

- price movements in financial and commodity markets
- traffic levels in networks
- physical parameters (such as temperature, pressure, etc.) of physical phenomena (such as oceans, the atmosphere, etc.)

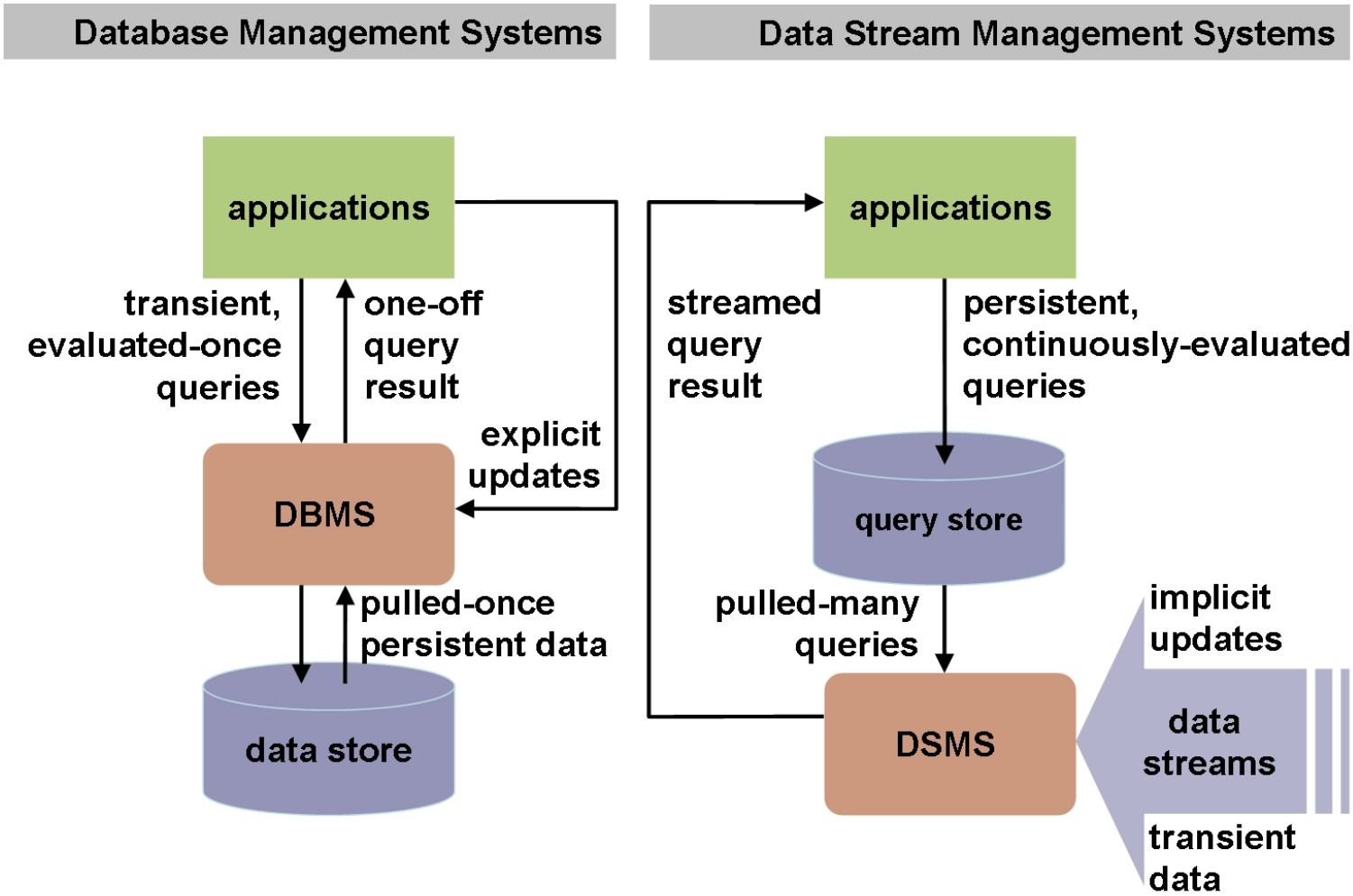
Data Stream v. Data Management Systems (1)

Contrasts (1)

	DBMSs	DSMSs
Data	Persistent Tuple set/bag Bounded size	Transient Tuple sequence Unbounded size
Updates	Explicit Modify in place	Implicit Append only
Access	Random Multi-pass	Sequential One-pass only
Queries	One-off, transient Exact answer	Continuous, persistent Approximate answer
Execution	Pro-active Pulled data Fixed plan	Reactive Pushed data Adaptive plan

Data Stream v. Data Management Systems (2)

Contrasts (2)



Data Stream v. Data Management Systems (3)

Challenges Arising

- Operator argument have unbounded cardinality, so one cannot hope to scan them in their entirety.
- Blocking operators (like sort, join, duplicate removal, etc.) have no defined semantics over unbounded streams.
- Data is pushed onto the system, so the data stream management system (DSMS) must keep state to receive arriving data.
- If arrival rates are higher than the achievable throughput, the DSMS may not be able to keep up.
- Since many queries are registered for continuous (or periodic) evaluation, sharing execution load is possible and beneficial.
- Since queries are long-lived, query execution plans (QEPs) must adapt over their lifetime.

Data Stream v. Data Management Systems (4)

Responses

- A window over a stream acts as a stream-to-relation conversion operator: it generates a bounded region in the stream, thereby allowing blocking operators to retain their classical semantics.
- Sliding such a window allows operators to produce/revise answers at each window slide.
- Operators interact via queues, which bind operators with one another and with source streams.
- If the DSMS is not be able to keep up, tuples are shed according to some policy.
- Multi-query optimization techniques are used to share work (e.g., when here are several QEPs in the query store, they may share subplans).
- Adaptive query processing techniques (e.g., special, adaptive operators) are used to respond to changing conditions.

Windows (1)
Why? What kinds? (1)

- A window is a mechanism to superimpose a region of definite cardinality over a stream whose cardinality is unknown.
- The main kinds of windows are:

  time-based : keep the items that have arrived in the last k time units

  count-based : keep the last k items to have arrived

  punctuated : keep all items until a marker k has arrived
- Punctuated windows are particularly useful for streams of un- or semistructured data, like text or XML documents.
- Time-based windows are particularly useful for structured data in transactional or in monitoring streams.
- Count-based windows are useful irrespective of the degree of structure in the data.

Windows (2)
How?

- A window, by default, changes as data arrives and the operators that use it re-evaluate as it does so.
- Time- and count-based windows have specified scope in terms of a number of units, i.e., time units or tuples.
- It is often convenient, in these cases, to specify a **slide**, i.e., a number of units (either time units or tuples) that must come to pass (or arrive) to trigger re-evaluation.
- As time passes and more items arrive, some items in the window will be removed because they have fallen out of the window scope.
- Thus, at every change or slide, the conditions that have thus far justified the inclusion of some items in the window may now have become invalid.
- If so, these items are said to have expired and are removed.
- Continuous query evaluation takes into account valid items only (i.e., those that have not expired yet).

Query Syntax in DSMSs (1)
Language Extensions (1)

- The simplest language extensions make use of SQL OLAP syntax.
- In the FROM clause, if a name denotes a stream it may have a window specification placed upon it.
- Time-based windows are specified as having a given RANGE, i.e., a certain width in time units.
- Count-based windows are specified as holding a given number of ROWs.

Query Syntax in DSMSs (2)
Language Extensions (2)

Example

- Every minute, report items with class higher than A whose price has moved in the last two minutes.

```
SELECT *
  FROM Prices
       [RANGE 2 min SLIDE 1 min]
 WHERE class > ‘‘a’’
```

- After every price movement, report how many items with class higher than A appeared in the last three movements.

```
SELECT COUNT(*)
  FROM Prices [ROWS 3 SLIDE 1]
 WHERE class > ‘‘a’’
```

- When new information arrives in S1 or in S2, report those items whose prices have coincided within the last three minutes.

```
SELECT *
  FROM S1 [RANGE 3 min],
       S2 [RANGE 3 min]
 WHERE S1.price = S2.price
```

Query Semantics in DSMSs (1)
Contrasts

- While the syntax may be SQL-like, the semantics is rather different.
- In a stream QL, tuples have an ordering attribute (typically a timestamp, always implicitly retained), but not in SQL.
- In SQL, the answer is a table; in a stream QL, it is a stream or a materialized view.
- Intuitively, the answer to a stream query is the answer of the corresponding SQL query (i.e., with the window specifications removed) over the current state of the input streams/windows.
- This means that the answer changes over time as the windows slide forward, i.e., whenever a new tuple arrives or an old tuple leaves the window.
- To represent additions and deletions from the result, positive and negative tuples can be generated or else the answer can be recomputed from scratch on every change.

Query Semantics in DSMSs (2)

Some Examples (1)

Example

```
SELECT *
  FROM S [RANGE 2 min SLIDE 1 min]
 WHERE class > ‘‘a’’
```

`SLIDE 1 min` Every minute

`FROM S` take the S stream, consider only those tuples

`RANGE 2 min` that were timestamped in the last two minutes and

`WHERE class > ‘‘a’’` that have class above “a”,

`SELECT *` report them.

Query Semantics in DSMSs (3)

Some Examples (2)

Example

```
SELECT *
  FROM S1 [RANGE 3 min],
       S2 [RANGE 3 min]
 WHERE S1.price = S2.price
```

- Whenever data arrives in S1 or S2,

`FROM S1` take the S1 stream, consider only those tuples

`RANGE 3 min` that were timestamped in the last three minutes,

`S2` take the S2 stream, consider only those tuples

`RANGE 3 min` that were timestamped in the last three minutes,

- form the Cartesian product of the S1 and S2 tuples that are in scope,

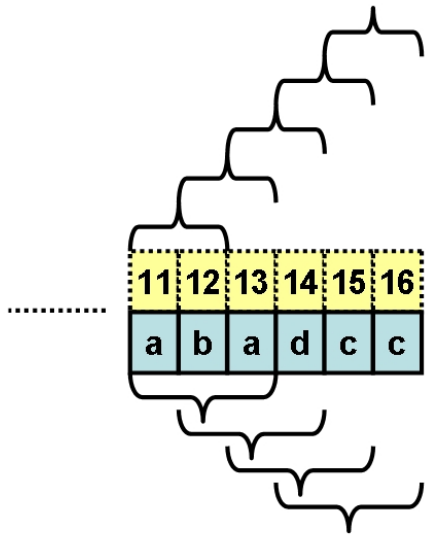
`WHERE S1.price = S2.price` keep those who have identical prices,

`SELECT *` report them.

Query Semantics in DSMs (4)

Two Examples

```
SELECT *
  FROM S [RANGE 2 min SLIDE 1 min]
 WHERE class > "a"
```




```
@11: {<10, f>}
@12: {<12, b>}
@13: {<12, b>}
@14: {<14, d>}
@15: {<14, d>, <15, c>}
@16: {<15, c>, <16, c>}
```

Window Queries: Time-Based/ Row-Based

```
SELECT COUNT(*)
  FROM S [ROWS 3 SLIDE 1]
 WHERE class > "a"
```

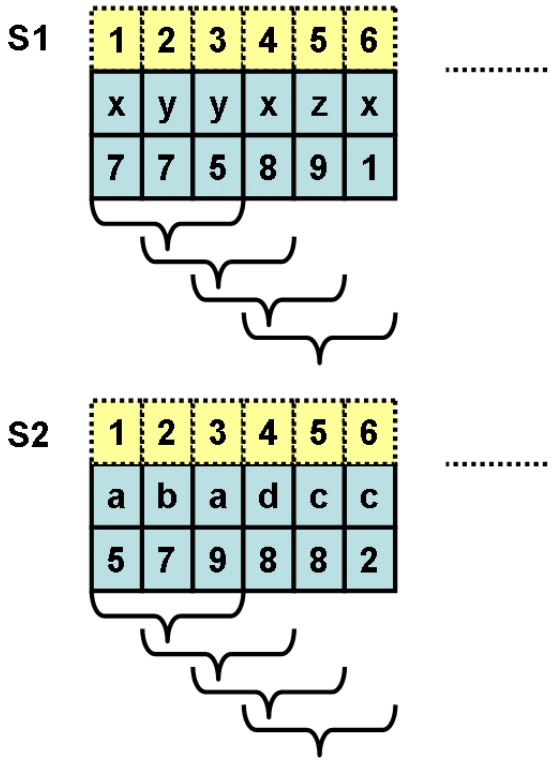
```
@11: { 1 }
@12: { 0 }
@13: { 1 }
@14: { 2 }
@15: { 2 }
@16: { 3 }
```

Query Semantics in DSMSs (5)

A Time-Based Window Join

Example Time-Based Window Join

```
SELECT *
  FROM  S1 [RANGE 3 min],
        S2 [RANGE 3 min]
 WHERE S1.price = S2.price
```




```
@1: -
@2: -
@3: {<1, x, 2, b, 7>,
     <2, y, 2, b, 7>,
     <3, y, 1, a, 5>}
@4: {<2, y, 2, b, 7>,
     <4, x, 4, d, 8>}
@5: {<4, x, 4, d, 8>,
     <4, x, 5, c, 8>,
     <5, z, 3, a, 9>}
@6: {<4, x, 4, d, 8>,
     <4, x, 5, c, 8>}
```

Query Semantics in DSMSs (6)
Selection, Projection

- Selection is non-blocking, so there is no need for windows.
- Likewise, for projection under bag semantics (retaining the ordering attribute).
- Under set semantics, duplicate removal requires that we superimpose a window on the stream.

Query Semantics in DSMSs (7)
Joins (1)

- The simplest is a **binary join** over sliding windows.
- It is inspired by symmetric hash join.
- The informal semantics of a sliding window join between two streams S_1 and S_2 is as follows.
- When a new tuple arrives in one of the operands, say S_1:
  1. Scan the window on S_2 to find any matching tuples and propagate the concatenations into the answer.
  2. Insert the new arrival in the window on S_1.
  3. Invalidate all the tuples in the window on S_1 that have expired as a consequence.
- The process is symmetrical when a new tuple arrives in the other operand.

Query Semantics in DSMSs (8)
Aggregation

- Distributive aggregation functions (e.g., `COUNT`, `SUM`, `MAX`, `MIN`) only require that we hold on to the last answer we emitted and update it at each new arrival before emitting the new answer.
- Algebraic aggregation functions (e.g., `AVG`) require that we hold on to the terms (e.g., `COUNT`, `SUM`) used to compute the last answer we emitted and update them at each new arrival before computing and emitting the new answer.
- Holistic aggregation functions (e.g., `MEDIAN`, `COUNT DISTINCT` require that we superimpose a window on the stream.
- Group-by aggregation can, as usual, be done using hash-based techniques to hold the partitions, in this case updating them for new arrivals works much as has been described for binary join.

Summary
Data Stream Management

- Data stream management is widely believed to be a growth area for the deployment of database technology.
- Many modern organizations have as part of their competitive strategy the ability to respond in real time to external events.
- Data stream management systems are very well placed to perform the kind of complex event processing that such organizations require.
- However, the challenges posed by data streams to classical DBMS technology are unprecedented, ranging from foundational issues, through query semantics and optimization, to adaptive query processing.

Advanced Database Management Systems

Data Stream Query Processing

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

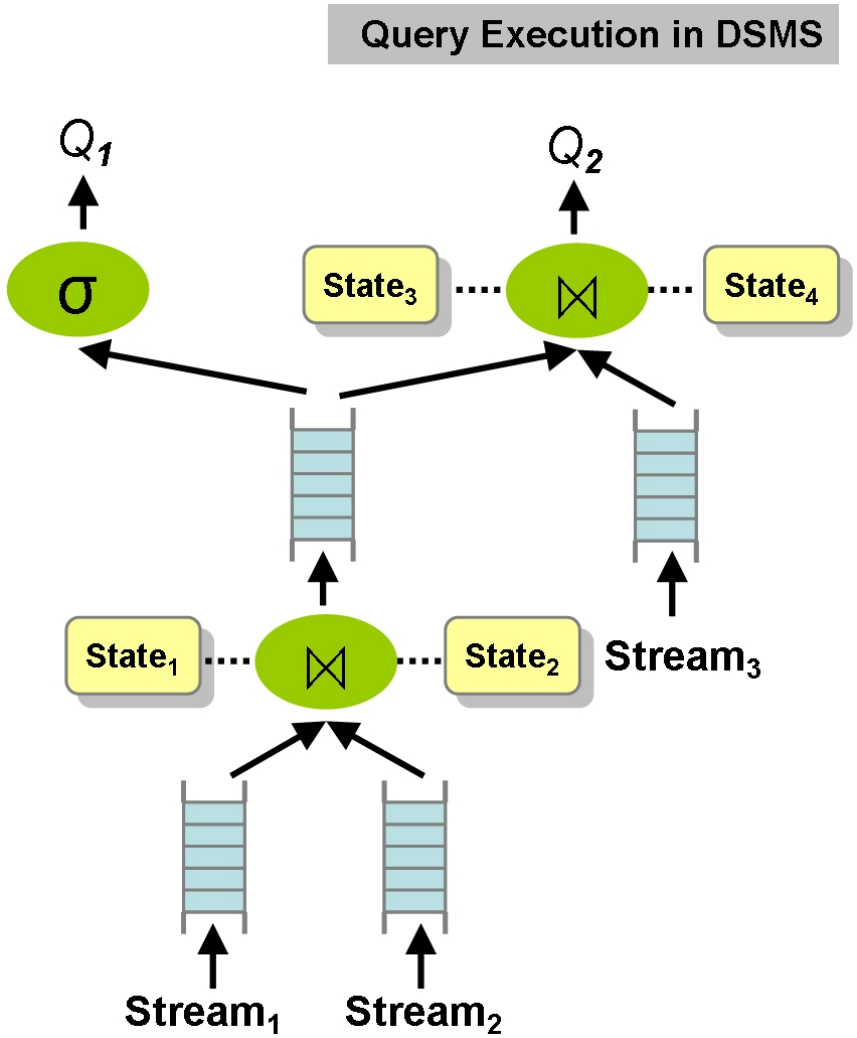
Query Optimization in DSMSs

Example DSMSs

Query Execution in DSMSs

A Typical Picture

- When a continuous query Q_n is registered, a QEP P_n is generated for Q_n.
- The new plan is merged with the collection of existing plans $P_1, \ldots, P_{n-1}$.
- At any point in time, the registered queries form a graph: individual queries share inputs and outputs.
- The collection of QEPs comprises:
  - Operators
  - Queues, both input and inter-operator ones
  - State (e.g., windows, previous results, etc.)
- A global scheduler oversees which operators evaluate in response to what.



Query Optimization in DSMSs (1)
General Framework

- The general idea is still to generate candidate query plans by rewriting, e.g..:selections and time-based windows commute, but selections and count-based windows do not.
- While we still want to reduce sizes, since operators keep state, evaluation is in main-memory mostly, so disk I/O not as major a cost concern.

Query Optimization in DSMSs (2)

Issues Regarding Multi-Query Execution

- In a DBMS, a query is issued and runs as if in isolation; in a DSMS, many queries are likely to be executing together for potentially long periods at any one time.
- If so, there are opportunities for sharing, e.g.:
  - Same SELECT and WHERE clauses but different window scope in the FROM clauses.
  - Same SELECT and FROM clauses but different predicates in the WHERE clauses.
- It is also possible, e.g., to generate indexes from a list of predicates that are active and, when a new tuple t arrives, find which predicates it satisfies, thereby allowing the scheduler control over which queries and operators to trigger.

Query Optimization in DSMSs (3)
Issues Regarding Operator Scheduling

- Possible overall strategies include:
  - Many tuples at a time: each operator gets a time-slice and the tuples in its input queue(s).
  - Many operators at a time: each tuple is processed by all the operators in a path in pipelined fashion.
- The choice of scheduling strategy depends upon the optimization goal:
  - Is it to minimize end-to-end latency? A tuple should take the least amount of time possible from its arrival to being reflected in the result.
  - Is it to maximize tuple output rate for the query? Given an arrival rate and two operators O and O' that are neighbours in a pipeline and commute, if they have the same selectivity, the one with faster output rate should execute earlier.

Query Optimization in DSMSs (4)
Issues Regarding Adaptivity

- System conditions can change throughout the lifetime of a persistent query, e.g.:
  - The overall workload can change as the QEP collection changes.
  - Stream arrival rates can change.
- One option is to adapt the plan on-the fly, e.g., change from a symmetric to an asymmetric binary join.
- Another is to use adaptive operators from the start, e.g., collapse a join sequence into a single operator (known as an **eddy** [Avnur and Hellerstein, 2000]) and thread each tuple through all the joins but decide the route dynamically, in response to the observed output rate in each join.
- Eddies implement dynamically changing join ordering strategies.

Query Optimization in DSMSs (5)
Issues Regarding Load Shedding (1)

- When the system is overwhelmed because the scheduler cannot produce tuple output rates that contend with the tuple arrival rates being experienced, there is a need for strategies to shed load.
- These include:
  - Randomly drop a fraction of arriving tuples: in monitoring streams, in certain contexts (e.g., where sampling is acceptable), this may be sound.
  - Examine the contents of a tuple before deciding whether or not to drop it, on the assumption that some tuples may have more value than others (e.g., in detection contexts, a single possibly rare event is valued more highly than confirmation of normality).

Query Optimization in DSMSs (6)
Issues Regarding Load Shedding (2)

- Rather than dropping tuples, we can also:
  - Spill over to disk and pick up to process during quieter times
  - Narrow the scope of the windows, perhaps progressively.
- One guiding optimization goal in load shedding is to minimize the impact on accuracy or the approximation error.

Data Stream Management Systems (7)
Aurora/Borealis

- Aurora [Abadi et al., 2003] is geared towards monitoring applications (streams, triggers, imprecise data, real time requirements).
- Rather than as declarative queries, Aurora tasks are specified as a connected data flow graph where nodes are operators.
- Optimization is over this data flow graph
- Aurora supports three query modes: continuous, which is classical for streams; ad-hoc, which allows a query to be placed from now until explicitly terminated, and view, which allows for results to persist.
- Aurora accepts QoS specifications and attempts to optimize QoS for the outputs produced
- It performs real-time scheduling and load shedding

Data Stream Management Systems (8)
Gigascope

- Gigascope [Cranor et al., 2003] specializes in network applications (a consequence of its origins in AT&T).
- It has a declarative language, GSQL, that is a pure stream query language (i.e., all inputs and outputs are streams).
- It uses ordering attributes to turn blocking operators into non-blocking ones through a *merge* operator that is an order-preserving union of two streams.
- Rather than interpret QEPs, it generates executables (and pushes computation as low as possible, e.g., into network adapters).
- It provides for foreign functions to allow for escaping the pure stream model (and perform more complex joins, e.g.).

Data Stream Management Systems (9)
STREAM

- STREAM [Arasu et al., 2004] is a general purpose data stream management system.
- It has a declarative language, CQL, that uses stream-to-relation, and relation-to-stream converters in order to retain the classical semantics of relational-algebraic operators.
- It aggressively shares state and computation among registered queries and carefully considers resource allocations and use through its scheduler.
- It performs continuous self-monitoring and re-optimization.
- It tries to approximate gracefully if necessary.

Data Stream Management Systems (10)
TelegraphCQ

- TelegraphCQ [Chandrasekaran et al., 2003] supports continuous queries over a mixture of relations and streams.
- It allows for both sliding and landmark windows to be defined (the latter has a fixed older end and a newer end that moves forward as new tuples arrive in the stream).
- The language used is SQL-like but window specification is much more expressive than the means inherited from SQL OLAP.
- TelegraphCQ query execution is focussed on adaptivity and on multi-query optimization opportunities.

Data Stream Management Systems (11)
Related Areas

- Publish-subscribe (pub-sub) systems process a very large number of simple conditions against a stream of events, while DSMS execute more complex queries.
- Sensor network applications are stream systems that, when deployed in isolation from power sources and communication sinks, have concern themselves with energy-efficient query plans to save battery power, and with in-network processing and storage.
- Approximate query processors for computing on-line aggregates in limited space work by summarizing a stream (e.g., maintaining a sample) and running queries over the summary.
- On-line data stream mining is used for incremental clustering and classification as well as subsequence matching, among others.

Summary
Data Stream Query Processing

- Stream query processing is motivated by emerging data-intensive applications that monitor an environment as it evolves.
- Many novel problems arise in all of data modelling, query syntax and semantics, query optimization and processing.
- Central to the challenges is the unbounded nature of streams and the data-driven, rather than query-driven, nature of query execution.
- In DBMSs, data persists while queries are transient. In contrast, in stream query processing, data is transient and queries persist.

Advanced Database Management Systems

Sensor Network Data Management

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Sensor Network Data Management

Sensor Networks as a Distributed Computing Platform

SNDM Desiderata

Sensor Networks as a Hardware/Software Platform

How Did We Get Here?

- Sensor network data management (SNDM) is yet another consequence of the ascendancy of distributed computing as the dominant computing paradigm.
- In the database area, SNDM builds not only on previous work on distributed and parallel DBMS but also on P2P query processing (QP) and on stream QP.
- Like P2PQP engines, sensor network QP (SNQP) engines implement an overlay network.
- Like stream QP engines, SNQP engines process data streams.
- In comparison with P2P and stream data management, there are fewer fundamental challenges in SNDM.
- Novel challenges abound but they cluster around the extremely constrained nature of the platform.

Sensor Networks (1)
What Are They?

- A typical sensor network (SN) comprises 10^1 to 10^2 sensor nodes, often referred to as **motes**.
- A mote is
  - (typically) small
  - battery-powered
  - endowed with limited computing capabilities
  - capable of sensing the physical environment
  - capable of forming links with other nodes by means of wireless radio communication.

Sensor Networks (2)
What Are They For?

- The major application areas so far have been:
  environmental data collection e.g.:
  - of natural phenomena such as floods, fires, volcanic eruptions, etc.
  - of natural habitats such as bird colonies, forests, glaciers, etc.
  - of civil structures such as bridges, buildings, etc..

  entity tracking e.g.:
  - of animals in natural environments,
  - of vehicles in built environments,
  - of goods in organizations, etc..

  event detection e.g.:
  - of risk hazards such as rising pressure in utility pipes, rising water levels in river basins, etc.
  - of intruders, patients in risk, etc..

Sensor Networks (3)
What Do They Do?

- Each sensor in a SN takes time-stamped measurements of physical phenomena, e.g. temperature, light, sound, air pressure. etc..
- Sensed data is annotated at source, e.g., with the id, location, and type of the sensor node that obtained it.
- Sensor nodes go beyond producing data: they are responsible for computing, storage and communication.

Sensor Network Data Management (1)
Basics

- Each sensor node can be seen as a processing and storage element in a distributed, shared-nothing architecture with a wireless interconnect.
- From a database viewpoint, a SNQP engine allows a SN to be viewed as a distributed database that obtains data by sensing the physical environment and over which we can run declarative continuous queries.

Sensor Network Data Management (2)
Contrasts with Existing DBMS Technology (1)

- The network replaces the storage and the buffer manager: data transfers are from data in node memory as opposed to data blocks on disks.
- Node memory is limited by cost and energy considerations, unlike disk storage, which is relatively inexpensive.
- As with P2P approaches, the system is highly volatile (nodes may be depleted, links may go down): the system should provide the illusion of a stable environment.
- Unlike stream QP, SNQP engines are said to be **acquisitional**, insofar as the rate in which data enters the system is typically specified as a quality-of-service (QoS).

Sensor Network Data Management (3)
Contrasts with Existing DBMS Technology (2)

- Nodes typically only have depletable energy stocks, which are often hard to replenish.
- Classical qualities of service, e.g., response time, are less important.
- SNQP must optimize for low energy consumption in order to maximize longevity.
- Since the energy cost of communication may be up to an order of magnitude larger than that of processing, doing as much in-network processing as possible tends to be advantageous.
- Query processing tends to become highly aware of and very closely coupled to the networking layer.

Sensor Network Data Management (4)
Contrasts with Existing DBMS Technology (3)

- Limited storage on nodes along with high communication costs prevent offloading, so persistent data must be subject to compression, summarization and deletion policies, typically based on aging if queries about the past are to be supported.
- Since data is discarded, answers may be approximate.
- Since sensed data consists of measurements from the physical world, errors (e.g. noise) are inevitable, so support for range (instead of exact) and probabilistic queries is important.

Sensor Network Data Management (5)
Desirable Characteristics

persistence : data stored must remain available to queries, despite sensor node failures and changes in the network topology

consistency : a query must be routed correctly to a node where the data are stored

controlled access to data : different update operations must not undo one another's work, queries must always see a valid state of the DB

scalability : as the number of nodes increases, the total storage capacity should increase, and the communication cost should not grow unduly

balance : storage should not unduly burden any node, nor should a node become a hotspot of communication

topological generality : DB architecture should work well on broad range of network topologies

Sensor Network Data Management (6)

Example Performance Metrics (1): Network

total network traffic : the sum total of bytes sent, which is an indicator of probable longevity

per-node network traffic : this indicates whether there are hotspots, which when they fail may cause the network to become disconnected before it is depleted of energy

Sensor Network Data Management (7)

Example Performance Metrics (2): Storage

available space : some SNQP engines hog persistent memory, leaving less room for measurements to be held

data longevity : the average amount of time a data item is accessible in storage (with variants for its having been summarized, approximated, etc.)

data access time : as with P2P networks, distributed data structures (e.g., geographic hash tables) may or may not deliver performance

Sensor Network Data Management (8)

Example Performance Metrics (3): Processing

delivery time : the amount of time taken for a measurement to become part of the answer

acquisition rate : the frequency with which measurements are obtained

output rate : the frequency with which answers are produced

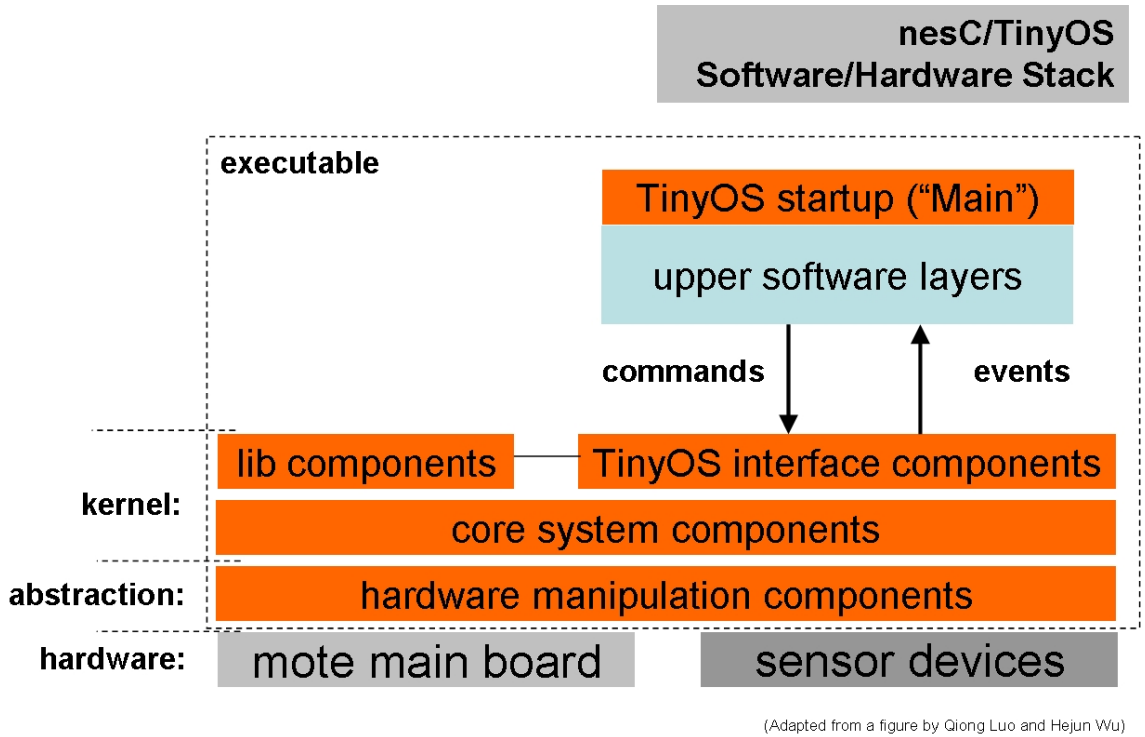
Sensor Network Platforms (1)
A Typical Mote: MICA (by Crossbow)

- 2.25 x 1.25 by 0.25 inches (5.7 x 3.18 x.64 centimeters), two AA batteries
- 8-bit 4 MHz Atmel ATmega 128L (as much as the original 1982 IBM PC)
- But it only consumes 8 milliamps when running, and 15 microamps when sleeping.
- 512 KB of flash memory
- 10-bit A/D converter for temperature, acceleration, light, sound and magnetic sensors
- 40 Kbps, 10^2m-range radio, 10 milliamps receiving, 25 milliamps transmitting
- Like most sensor nodes, MICA motes run nesC/TinyOS executables.

Sensor Network Platforms (1)

nesC/TinyOS: *de facto* Standard HW/SW Abstraction Layer for SNs

- TinyOS [Hill et al., 2000] is a component-based, event-driven runtime environment designed for wireless SNs.
- nesC [Gay et al., 2003] is a C-based language for writing programs over a library of TinyOS components.
- The figure shows how upper software layers written in nesC generate mote-level executables that rely on several kinds of TinyOS components.

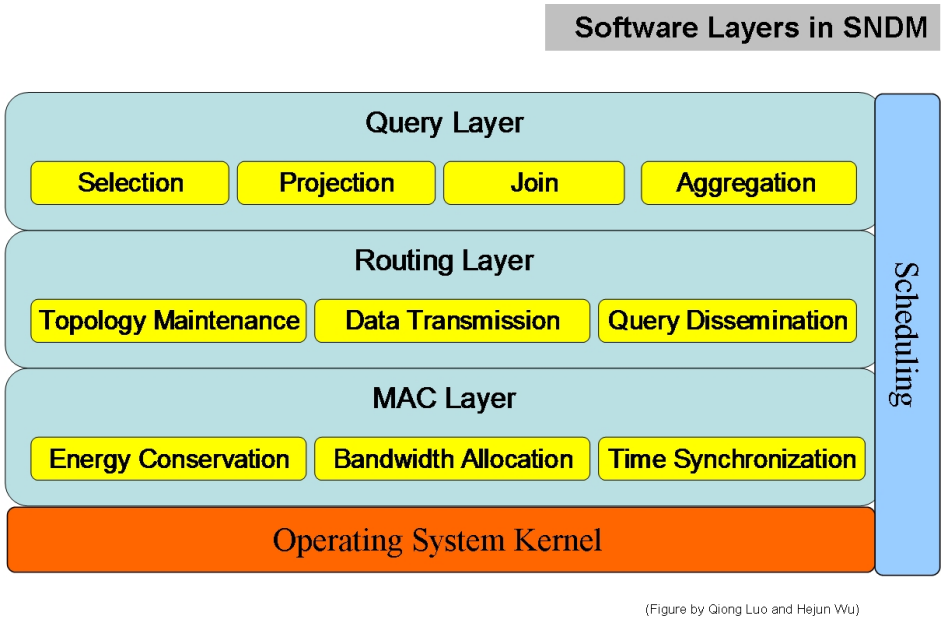


(Adapted from a figure by Qiong Luo and Hejun Wu)

Sensor Network Platforms (1)

Upper Software Layers

- There is great diversity in the upper software layers.
- The figure shows a conceptually plausible division of labour between software layers in the case of SNQP engines:
  - The topmost layer implements query execution functionality.
  - It relies on a routing layer that implements the overlay network required to carry the data flows that make up a query.
  - The routing layer relies on the medium-access control (MAC) layer that implements the radio-level protocols required.
-
  - Scheduling tasks cuts vertically across software layers.



(Figure by Qiong Luo and Hejun Wu)

Summary
Sensor Network Data Management

- SNDM has emerged as another form of distributed computing: it share with P2P the idea of overlay networks and with data streams the goal of processing events, in this case, grounded on physical reality.
- A SN is a distributed computing platforms, albeit an extremely resource-constrained one.
- From such constraints there emerge desiderata and performance metrics that make SNDM platforms distinct from any other DBMS technology.
- Fully-functional hardware and software platforms are available from sensor nodes to system-level programming platforms.

Advanced Database Management Systems

Sensor Network Querying

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Sensor Network Queries

Sensor Network Querying with TinyDB

Sensor Network Queries (1)

- When a SN is construed as a data management platform, SQL-like declarative queries can be used to retrieve information from it.
- This is a very significant advance on the alternative of programming data retrieval tasks directly, because the very low level at which the hardware/software infrastructure is cast makes the software engineering task extremely difficult and costly.
- It is orders of magnitude more convenient and cost-effective to pose a declarative query and have the SNQP map that to an interpretable/executable program that can retrieve the desired data.

Sensor Network Queries (2)

- Consider SNs that act as flood warning systems.
- Consider the needs of an emergency management agency to monitor the consequences of heavy rainfall in a region (e.g., Hull).
- An example SN query in this context might be:

  Every 10 minutes for the next 3 hours, report the maximum rainfall level in stations in Hull, provided that it is greater than 3.0 inches.

```
      select max(rainfall_level), station
        from Sensors
       where area = 'Hull'
    group by station
      having max(rainfall_level) > 3.0
    duration [now, now + 180 min]
sampling period 10 min
```

Sensor Network Queries (3)

- In the example just used (but not in general), the query was expressed over one table comprising all sensors in the SN, with each sensor corresponding to a column in the table.
- This example assumed (as is usual) that there is metadata describing schemas and the execution environment available at the point of compilation (often referred to as the **base station**).
- If the SN is implemented as P2P (rather than a client-server) system, the query may originate from any node, and metadata may need to be retrieved by querying too (or be broadcast to every node).

Sensor Network Queries (4)

- Monitoring queries are long-running, continuously-evaluated queries.
- In the example, the `duration` clause stipulates the period during which data is to be collected.
- The `sampling period` clause stipulates the frequency at which the sensors acquire data (and, by default, results are delivered).
- The desired outcome is a set of notifications of system activity (periodic or triggered by special situations)

Sensor Network Queries (5)

- Some SN queries need to aggregate sensed data over time windows, e.g.,

  Every ten minutes, return the average temperature measured over the last ten minutes.

- Other need to correlate data produced simultaneously by different sensor nodes, e.g.,

  Report an alert whenever 2 sensor nodes within 10 meters of each other simultaneously detect an abnormally high temperature.

- Many queries contain predicates on the sensor nodes involved (e.g., it is common to refer to geographical locations), as is to be expected since SNs are grounded in the physical world.

Sensor Network Queries (6)

With respect to the time dimension, the major types of SN queries are:

- long-running, continuous queries: report results over an sliding time window, e.g.

  For the next 3 hours, every 10 minutes, retrieve the rainfall level in Hull stations.

- snapshot queries: retrieve sensed data the network at a given point in time (typically now), e.g.,

  Retrieve the current rainfall level in Hull stations.

- historical queries: retrieve past sensed data, e.g.,

  Retrieve the average rainfall level at all sensor nodes for the last 3 months of the previous year.

SN Querying with TinyDB (1)
The TinyDB SNDM System

- TinyDB is the seminal SNDM: it single-handedly delineated the research topic of SNQP.
- TinyDB is a nesC-coded distributed query processor that runs on MICA motes over TinyOS.
- It has had several successful deployments, mostly for environmental data collection, the largest consisting of around 80 nodes.
- It is now not actively developed any more but still constitutes the benchmark for more recent SNQP systems.
- Cougar was another influential SNDM platform but was never as fully developed as TinyDB and its influence has correspondingly diminished recently.

SN Querying with TinyDB (2)
TinyDB Query Cycle

- TinyDB assumes the existence of a base station (that is assumed to be a normal computer, say a PC).
- The base station parses and optimizes a query.
- The resulting QEP is injected into the SN.
- This starts a dissemination process as a result of which a routing tree is formed, with the QEP being installed in the sites that comprise it and then started.
- Results flow back up the routing tree.

SN Querying with TinyDB (3)
Query Language Features

- The TinyDB query language (called TinyQL) is a declarative SQL-like query language supporting selection, (limited kinds of) join, projection, and aggregation.
- It is a continuous QL, so it supports windows.
- It is an acquisitional QL, so it supports sampling rates.
- TinyQL views the entire collection of sensors as a single, unbounded universal relation, with attributes for all the modalities (e.g., temperature, pressure, etc.) for which there is a sensor.
- Each modality is modelled as a distinct attribute in the universal relation.
- Tuples are tagged with metadata, i.e., node id, location, etc.

SN Querying with TinyDB (4)
Example TinyDB Queries: Select/Project

Every second, for 10 seconds, return node id, light and temperature readings provided the temperature is above 10.

```
        select nodeid, light, temp
          from Sensors
         where temp > 10
sample interval 1s
           for 10s
```

- This query generates a stream at the base station, where it may be logged or output to the user.
- The stream is a sequence of tuples, each tuple including a timestamp.

SN Querying with TinyDB (5)
Example TinyDB Queries: Materialized Views

- In TinyQL, because of the design choice for a universal relation `Sensors`, windows cannot be specified as in stream QLs.
- Instead, windows are specified as materialized views over streams.
- The following materializes the last eight light readings taken 10s apart:

```
create storage point recentlight size 8
    as (          select nodeid, light
                    from Sensors
         sample interval 10s)
```

SN Querying with TinyDB (6)

Example TinyDB Queries: Joins

- In TinyDB, joins are only allowed between two storage points on the same node, or between a storage point and the `Sensors` relation.
- For example, the following is an example of what the TinyDB papers refer to as a **landmark query**.

  Every 10s, from now on, return the number of recent light readings that were brighter than the current reading.

```
select count(*)
  from Sensors s,
       recentLight r
 where r.nodeId = s.nodeId
   and r.light > s.light
sample interval 10s
```

SN Querying with TinyDB (7)
Example TinyDB Queries: Aggregation

- TinyDB supports aggregations over time intervals using sliding windows.
- For example:

  Every 5 seconds, sampling once per second, return the average volume over the last 30 seconds.

```
        select winavg(volume, 30s, 5s)
          from Sensors
sample interval 1s
```

SN Querying with TinyDB (8)
Example TinyDB Queries: Event-Based (1)

- TinyQL allows data collection to be initiated by event occurrences.
- Events are generated explicitly, either by another query or by the operating system.
- Event occurrences have attributes that bind parameters of an event-based query.

SN Querying with TinyDB (9)
Example TinyDB Queries: Event-Based (1)

- The following (rather naïve) query raises a `bird-detect` event by detecting a high temperature in a nest:

```
                select nodeid,loc
                 where temp > 5
output action signal bird-detect(loc)
       sample period 10s
```

- Then, the following query responds to `bird-detect` events raised:

  When a bird has been detected in a nest, report the average light and temperature at sensors near the nest.

```
        on event bird-detect(loc):
          select avg(light), avg(temp), event.loc
            from Sensors s
           where dist(s.loc, event.loc) < 10m
sample interval 2s for 30s
```

SN Querying with TinyDB (10)
Example TinyDB Queries: Lifetime-Based

- Instead of an explicit `sample interval` clause, users may request a specific query `lifetime`, i.e., a duration in days, weeks, or months

  For at least 30 days, report light and acceleration by sampling at as fast a rate as possible.

```
  select nodeId, light, accel
    from Sensors
lifetime 30 days
```

Summary
Sensor Network Querying

- Using declarative queries to retrieve data from a SN has significant practical and economical benefits.
- TinyDB exemplifies the functionality that is capable of being supported.
- Complex queries and event detection are expressible, accompanied by quality-of-service expectations regarding lifetime and sampling intervals.

Advanced Database Management Systems

Sensor Network Query Processing

Alvaro A A Fernandes

School of Computer Science, University of Manchester

Outline

Query Processing in TinyDB

Query Processing in SNEE

Query Processing in TinyDB (1)
SNQP Engines as Autonomous Systems

- The main goal in SNQP (on battery-powered motes) is to reduce energy consumption.
- Deploying new sensor nodes in the field, or physically replacing or recharging batteries is time consuming and expensive, since deployment sites of interest tend to be remote, isolated and sometimes hazardous.
- This means that query optimization aims to generate QEPs that allow the SN to perform autonomously, i.e., the QEP controls where, when, and how often data is physically acquired (i.e. sampled), processed and delivered.
- TinyDB is an example of this class of SNQP engine.

Query Processing in TinyDB (2)
Duty Cycling as a Means to Save Energy

- Most motes can transition their hardware components between states with different energy consumption rates.
- Typical states are:

  Snoozing : the processor and radio are idle, waiting for either a timer- or an an external event to wake the device.

  Processing : after being woken up, the processor is doing local processing.

  Transmitting after being woken up, the radio is delivering results (either locally-obtained or relayed) to its parent in the routing tree.

  Receiving : after being woken up, the radio is receiving results from its children in the routing tree.
- Duty cycling is vital for longevity, and, therefore, the ability to spend time in lower-energy states is an important performance metric for SNQP engines.

Query Processing in TinyDB (3)

Networking: Short Ranges, Multi-Hops, Relays

- The current range for low-power wireless radios goes from a few feet to around 100 feet.
- Such short ranges imply the need for multi-hop communication where intermediate nodes act as relays (either purely or in combination with their sensing and processing duties).
- It is desirable that SNs be low maintenance and easy to deploy from a network management viewpoint.

Query Processing in TinyDB (4)
Networking: Time Synchronization

- Clock drift is likely in such limited hardware, leading to synchronization issues as to whether the target is in a receiving state when the source is in a transmitting one.
- There are different time synchronization protocols.
- The one used by TinyDB is simple and (seems to be) effective in practice:
  - All messages are sent with a 5-byte timestamp indicating node time in millisecs.
  - $system\_time := node\_time$
  - When a node receives a message it sets its node time to the $system\_time := timestamp\_received$.
  - All nodes agree that the waking period begins when $system\_time \bmod epoch = 0$, where *epoch* is the period between the start of each sampling activity.
  - The latter is known from the query text.

Query Processing in TinyDB (5)
Networking: Network Formation (1)

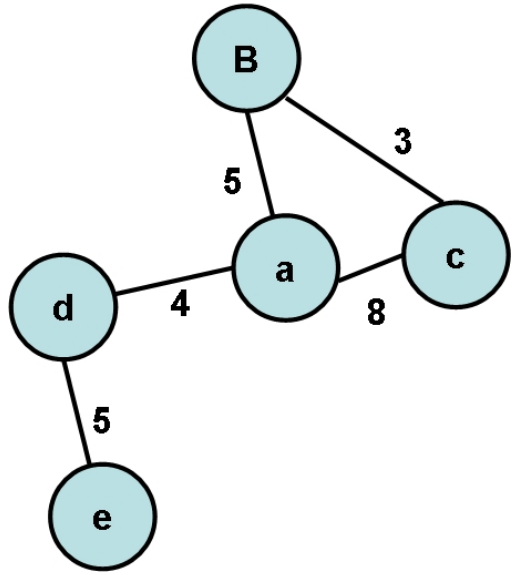
- TinyDB does not assume the communication topology to be known.
- Instead, it instruments nodes to form it in an ad-hoc manner.
- It uses a flooding algorithm similar to the one we alluded to when discussing P2PQP, as follows:
  1. The root broadcasts a request.
  2. All nodes that hear this request process it, and forward it on to their children, and so on, until the entire network has heard the request.
  3. This establishes a network topology (with undirected edges).
  4. A communication topology (i.e., one with directed edges) can then be chosen: nodes pick a parent node (with the most reliable connection to the root, i.e. highest link quality).
  5. This parent is then responsible for forwarding the node's (and its children's) messages to the base station.

Query Processing in TinyDB (6)

Networking: Network Formation (2)

- In the example network topology, vertices denotes nodes named by the corresponding label, with B denoting the base station.
- Edges denote that the nodes involved have a communication link between them (i.e., are within communication range of one another).
- Edge labels denote the quality (the higher, the better) of the communication link.

Network Topology with Base Station



Query Processing in TinyDB (7)

Networking: Network Formation (3)

- Given a network topology such as described, the selection of a communication topology is as follows: follows:
  1. Given nodes N and N', if N transmits and N' hears with quality Q, then a candidate routing edge $N' \rightarrow_Q N$ is proposed iff there is no already existing proposal of a candidate routing edge $N \rightarrow_Q N'$.
  2. If there is more than one candidate routing edge outgoing from the same node, i.e., if there are edges $N' \rightarrow_{Q_1} N_1, \ldots, N' \rightarrow_{Q_n} N_n$ then the one with the highest Q_i is chosen (or one is chose arbitrarily is there is a tie).

Query Processing in TinyDB (8)

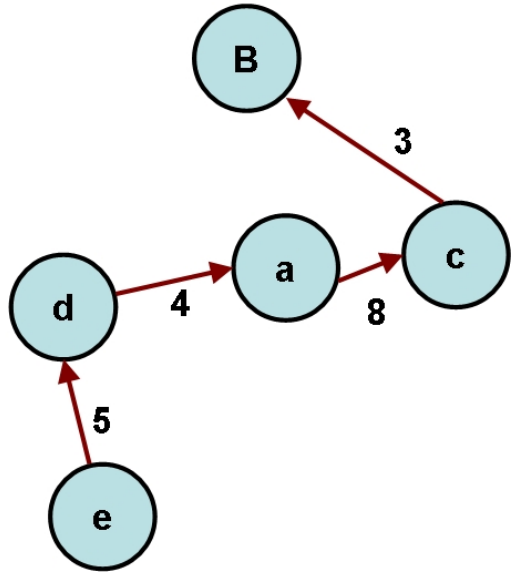
Networking: Network Formation (4)

- Given the previous network topology, the following are the derivation steps (with underlined candidate edges being discarded ones) which compute the communication topology:

1. $a \rightarrow_5 B$
2. $c \rightarrow_3 B$
3. $\underline{B \rightarrow_5 c}$
4. $a \rightarrow_8 c$
5. $\underline{B \rightarrow_5 a}$
6. $\underline{c \rightarrow_8 a}$
7. $d \rightarrow_4 a$
8. $\underline{a \rightarrow_4 d}$
9. $e \rightarrow_5 d$
10. $\underline{d \rightarrow_5 e}$

- There is one case of more than one candidate routing edge outgoing from the same node, viz., $\{a \rightarrow_5 B, a \rightarrow_8 c\}$, in which case, we choose $a \rightarrow_8 c$ because it has the highest quality.

Communication Topology with Base Station



Query Processing in TinyDB (9)
Query-Specific Routing

- Over a given communication topology, we can select the paths along it that data flows will follow in a QEP.
- When TinyDB disseminates the QEP (i.e., sends it to be installed at nodes) it computes what it calls a semantic routing tree (SRT), by which is meant that it takes into account the predicates used in the query to determine which nodes need to participate in the computation.
- This means that TinyDB also establishes the route for data flows (from leaves to root) as it decides (from root to leaves) which nodes will have the QEP installed and executing.

Query Processing in TinyDB (10)
Query-Specific Routing

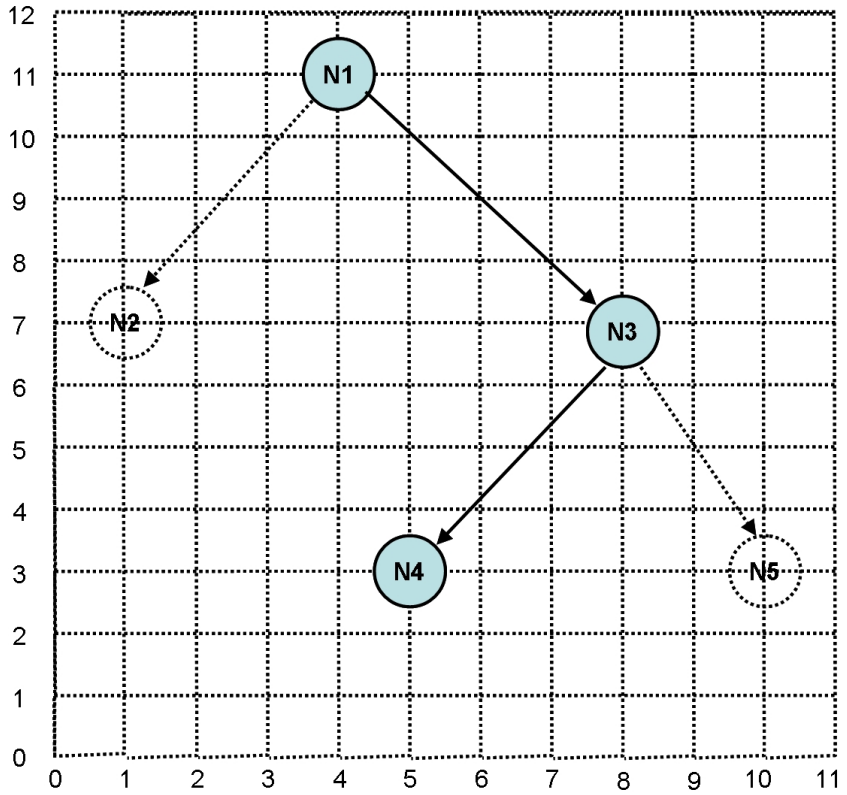
- An SRT is especially useful for a query in which the predicates in the WHERE clause define a geographical extent through an attribute A (e.g., the x-coordinate of a node).
- In a TinyDB SRT, each node stores a single unidimensional interval denoting the range of A values beneath each of its children.
- Then, the decision as to whether a node n must be involved in processing a QEP q with a predicate over A is taken as follows:
  1. When a QEP q with a predicate over A arrives in node n, if q applies locally to n, n participates in the execution of q, therefore n starts executing q.
  2. If the A-value of any n-child n' overlaps with the A-value in q (which condition n can verify from the A-range it holds), then n prepares to receiving results from any such n' and forwards q to them.
  3. If there is no overlap, then q is not forwarded from n, results.

Query Processing in TinyDB (11)
An Example TinyDB SRT

- Let the query be

```
SELECT light
  FROM Sensors
 WHERE x > 3 AND x < 7
```

- If so, $N1$ knows it can exclude $N2$, and $N3$ knows it can exclude $N5$.
- In this way, only the shaded nodes in the figure, i.e., those in the desired x-range, receive and execute the QEP



Query Processing in TinyDB (12)
Event Influence in TinyDB QP

- Events allow the nodes to snooze until some external condition occurs, instead of continually polling or blocking on an iterator waiting for some data to arrive.
- The benefit is significant reduction in energy consumption.
- When a query is issued, it is assigned an id that can be used to stop a query via a `stop query id` command,
- Queries can be limited to run for a specified lifetime via a `FOR` clause, or include a stopping condition that is an event occurrence.
- TinyDB can performs lifetime estimation if it is not stipulated: it uses a cost model relates sampling and transmission rate to energy consumption.

Query Processing in TinyDB (13)
Energy-Aware Optimization in TinyDB (1)

- Consider the following metadata about sensor hardware:

Sensor	Power	Sample time	Sample energy
Light, Temp	0.9	0.1	90
Magnetometer	15	0.1	1500
Accelerometer	1.8	0.1	180

- The table shows that sampling is expensive in terms of and varies between different modalities, e.g., the magnetometer consumes an order of magnitude more energy than other sensors.

Query Processing in TinyDB (14)
Energy-Aware Optimization in TinyDB (2)

- Note that a sample from a sensor s must be taken in order to evaluate any predicate over the attribute `Sensors.s`.
- If a predicate discards a tuple of the `Sensors.s` table, then subsequent predicates need not examine the tuple, and the expense of sampling any attributes in those predicates can be avoided.
- Thus, ordering the predicates in such a way that those that consume less energy are sampled first is often a good strategy.
- Now, consider the following example query Q:

```
          select accel, mag
            from Sensors
           where accel > 5
             and mag > 10
sample interval 1s
```

Query Processing in TinyDB (15)
Energy-Aware Optimization in TinyDB (3)

- There are three possible strategies to evaluate Q:
  1. the magnetometer and the accelerometer are sampled before either predicate is evaluated;
  2. the magnetometer is sampled, the predicate on it is evaluated then the accelerometer is sampled and the predicate on it is evaluated;
  3. the same as the previous but with the sampling order reversed.
- The first is always more energy-expensive than the latter two.
- The last is better than the second given that sampling accelerometer is cheaper unless, `mag` $>$ 10 is much more selective than `accel` $>$ 5.

Query Processing in TinyDB (16)
Processing TinyDB QEPs

- Once a query has been optimized and disseminated, the query processor executes it.
- Roughly (i.e., ignoring communication), the node:
  1. sleeps then
  2. wakes up then
  3. samples the sensor then
  4. processes both the data just obtained and that received from children then
  5. put the result in a queue for delivery to its parent then

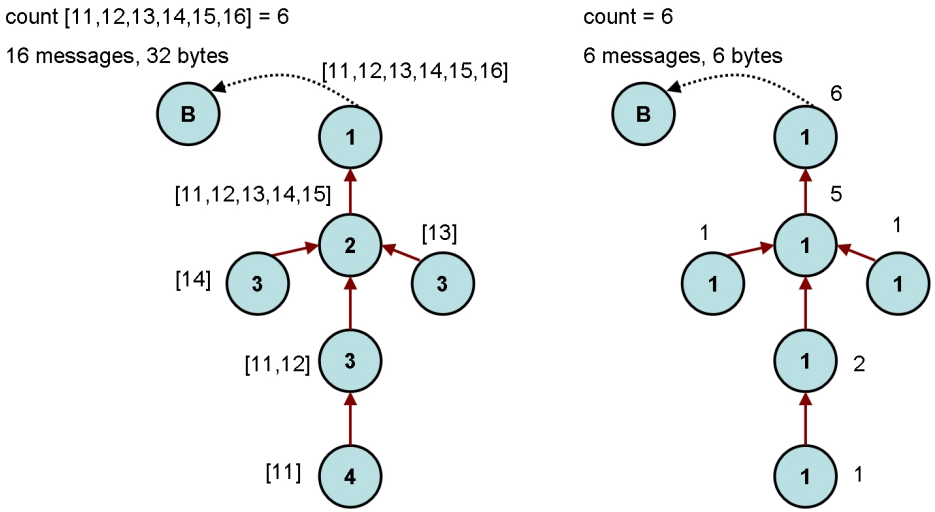
Query Processing in TinyDB (17)
Load Shedding in TinyDB

- When there is no contention, the queue can be drained faster than results arrive in it.
- When the opposite is the case, prioritizing data delivery is necessary.
- TinyDB uses three 3 simple prioritization schemes:
  1. naïve: a tuple is dropped if the queue cannot accept it.
  2. winavg: the first two results are averaged into one to make room at the tail.
  3. delta: each tuple is marked with to indicate how different it is from the last transmitted result, so that when there is a need to make room in the queue, the least different is dropped.

Query Processing in TinyDB (18)
Tree-Staged Aggregation in TinyDB

- TinyDB makes use of the tree-staged aggregation discussed in the context of P2P systems.
- In the case of SNs, though, the reduction in bandwidth is even more important because of the energy cost of radio communication.
- For example, consider the 3-hop routing tree in the figure and a `COUNT` query.
- If all data is sent to the base station, 16 messages and 32 bytes are transmitted..
- If sites perform partial aggregation on the way, 6 messages and 6 bytes are transmitted.

Tree-Staged Aggregation in SNs



Query Processing in SNEE (1)
SNQP as Distributed QP

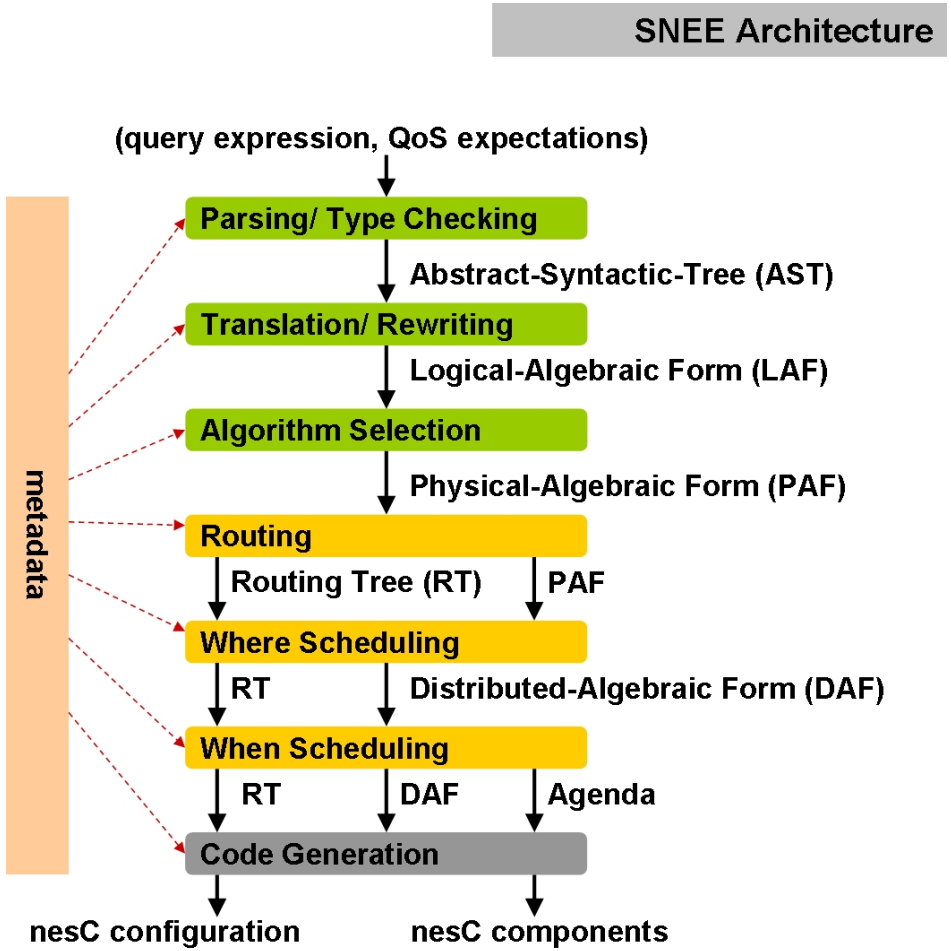
- TinyDB sends the same QEP to all participating nodes and expects a query engine to be running in every node.
- It can be seen as not being economical with memory.
- TinyDB also sends every tuple produces as soon as it is produced.
- It can be seen as not being careful to pack bytes when transmitting and receiving and therefore may find it harder to amortize the fixed per-message cost.
- It could be argued that construing a SN as distributed computing platform in a strict sense can overcome this and other shortcomings.
- SNEE is a SNQP developed in Manchester that takes this approach.
- For more detail on the remainder of these notes, see the assigned reading [Galpin et al., 2007].

Query Processing in SNEE (2)
SNEE v. TinyDB

- SNEE comes short of TinyDB in not supporting
  - specification of event-based queries
  - materialization of results
- SNEE matches TinyDB in
  - allowing the user to stipulate how often data is acquired and processed
  - performing in-network, tree-staged aggregation
  - allowing selection and projection
  - correlating data across time
- SNEE goes beyond TinyDB in
  - supporting application-specific relations
  - allowing windows on the past
  - supporting joins without materialization
  - allowing the specification of which sites and which sensed data to include in a query
  - correlating data across sites and times

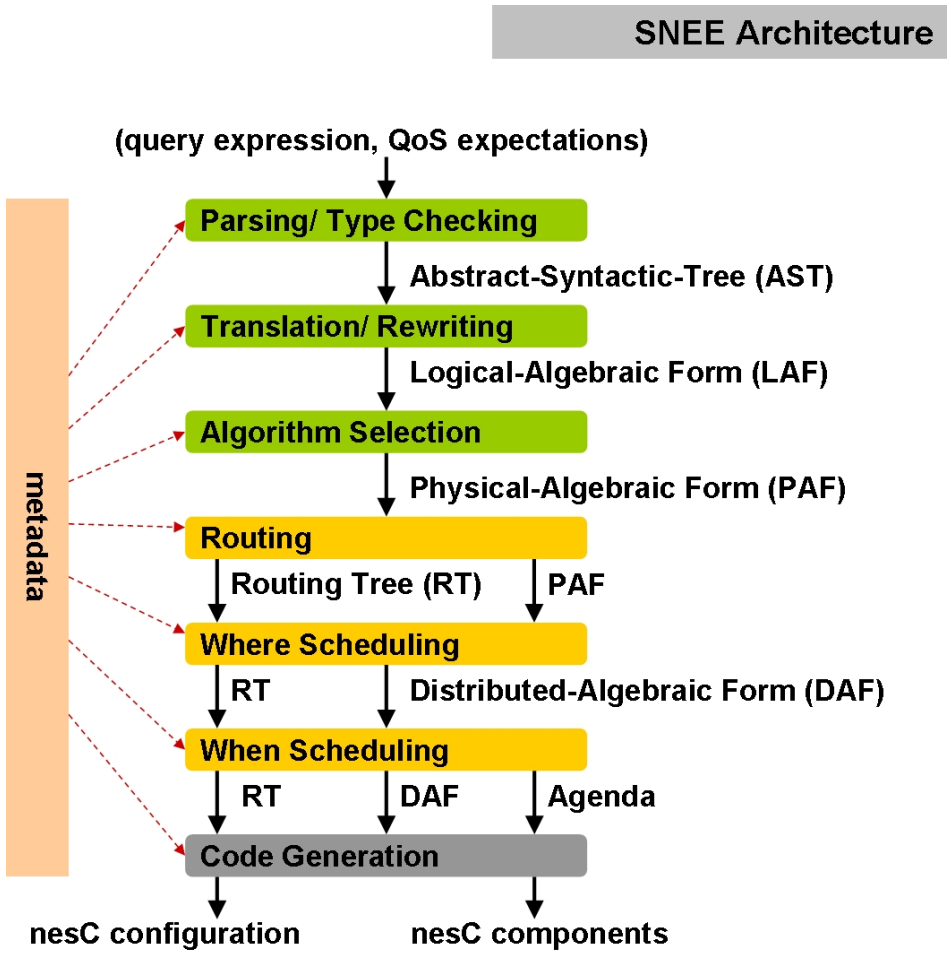
Query Processing in SNEE (3)
SNQP as Distributed QP (1)

- SNEE uses the well-known two-phase optimization approach of parallel/distributed DBMSs.
- In the figure, the three first stages are classical (except that SNEE does not perform much logical rewriting).
- SNEE introduces the notion of **routing** which is required in the case of SNs (because wireless, ad-hoc networking) but not in classical distributed DBMSs (because there is no need to specify an overlay network).



Query Processing in SNEE (4)
SNQP as Distributed QP

- The partitioning/scheduling phase is broken down in SNEE into what is referred to as **where scheduling** (deciding where QEP fragments run) and **when scheduling** (deciding when the QEP fragments run).
- Finally, unlike TinyDB, SNEE generates not interpretable QEPs, but nesC/TinyOS source code, which when compiled and deployed in sensor nodes constitutes an executing QEP.



Query Processing in SNEE (5)
SNQP as Distributed QP

- The first example query shows that SNEEql (the SNEE query language), inspired by STREAM's CQL, allows the definition of logical extents (e.g., `Inflow`, `Outflow`) in the `FROM` clause.
- The second shows that SNEEql allows windows on streams and supports aggregation.
- The third shows that SNEEql can specify windows on the past (e.g., data seen one minute ago) and express joins without materialization points.

```
Q1 == SELECT *
      FROM Outflow
      WHERE pressure > 24

Q2 == SELECT AVG(pressure)
      FROM Outflow
        [FROM NOW 10 secs TO NOW-5 secs]
      WHERE temperature < 10

Q3 ==
  SELECT Outflow.time,
         Inflow.pressure,
         Outflow.presure
    FROM Outflow [NOW],
         Inflow [AT NOW-1 Min]
   WHERE Inflow.pressure > 500
     AND Outflow.pressure > Inflow.pressure
```

Query Processing in SNEE (6)
Some QoS and Some Metadata Used by SNEE

- Users can specify some QoS expectations, e.g.:
  - acquisition rate
  - maximum delivery time
- The SNEE compiler/optimizer expects metadata such as:
  - the schema of each sensed stream
  - which sensor nodes sense which attributes
  - the network connectivity graph (NGP)

Query Processing in SNEE (7)
Some SNEE Optimization Techniques

- Recall that TinyDB does not presume to know the NGP and hence first derives it by flooding and then computes a routing tree that is data-sensitive.
- SNEE assume the NGP to have been asserted in the metadata and computes the routing tree as a minimum spanning tree that minimizes the total energy cost of routing.

Query Processing in SNEE (8)
Cost Models in SNEE

- SNEE uses cost models more extensively than TinyDB and hence is able to statically compute worst-case bounds on space and time for all QEP fragments.
- This allows it to derive a strict agenda for execution.
- Timers fire to wake up, acquire/receive, process, transmit/deliver data at specific time slots, which repeat with a periodicity that is determined by a buffering factor.
- The use of buffering makes SNEE QEPs more energy-efficient than TinyDB.
- The amount of buffering is computed from the QoS expectations regarding acquisition rate, maximum delivery time and the memory available/required..

Query Processing in SNEE (9)

Routing Tree and Distributed Algebraic Form for Example SNEE Query Q3 (1)

- Assume that Q3 (above) is posed with the following QoS expectations:

```
    Delivery time: Within 2.5 seconds
Acquisition rate: Every 2 seconds
         Lifetime: 3 months
```

- Assume further that the metadata describing where data is to be acquired and delivered is as follows:

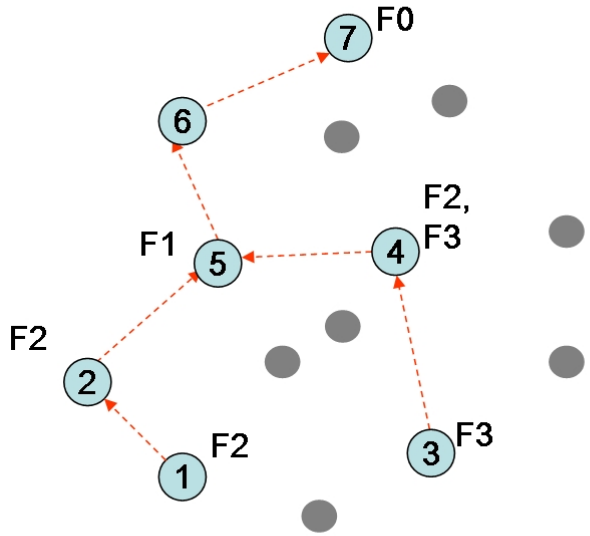
```
   Outflow at sites: 1,2,4
    Inflow at sites: 3, 4
Destination at site: 7
```

- The next figure shows the routing tree and the distributed algebraic form computed for Q3.

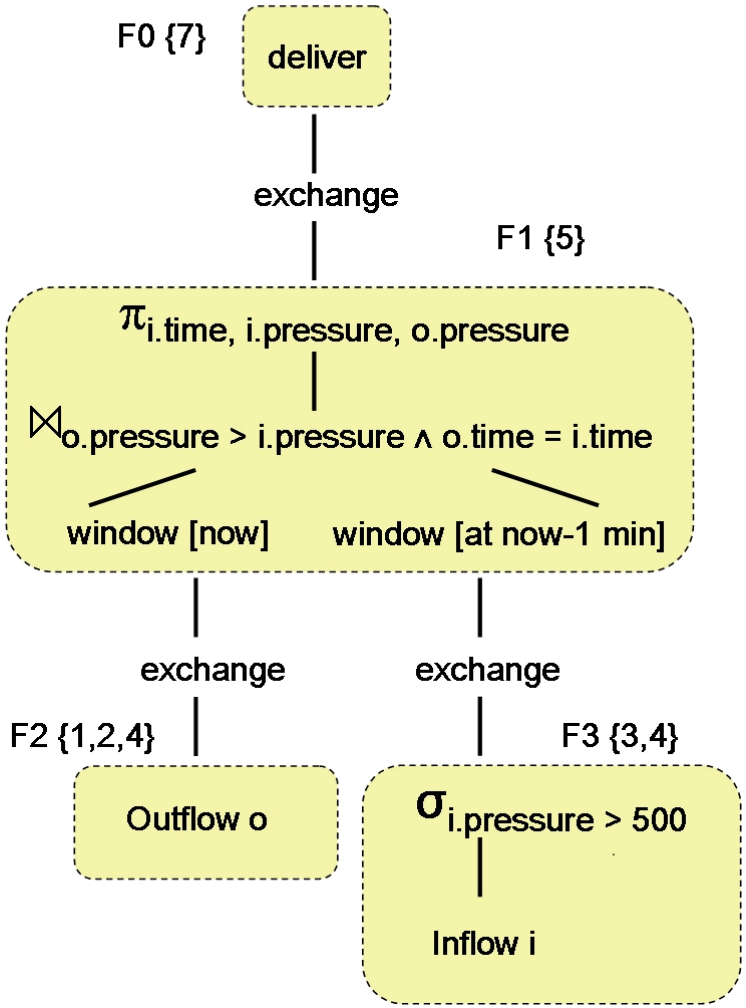
Query Processing in SNEE (10)

Routing Tree and Distributed Algebraic Form for Example SNEE Query Q3 (2)

Example RT for Q3



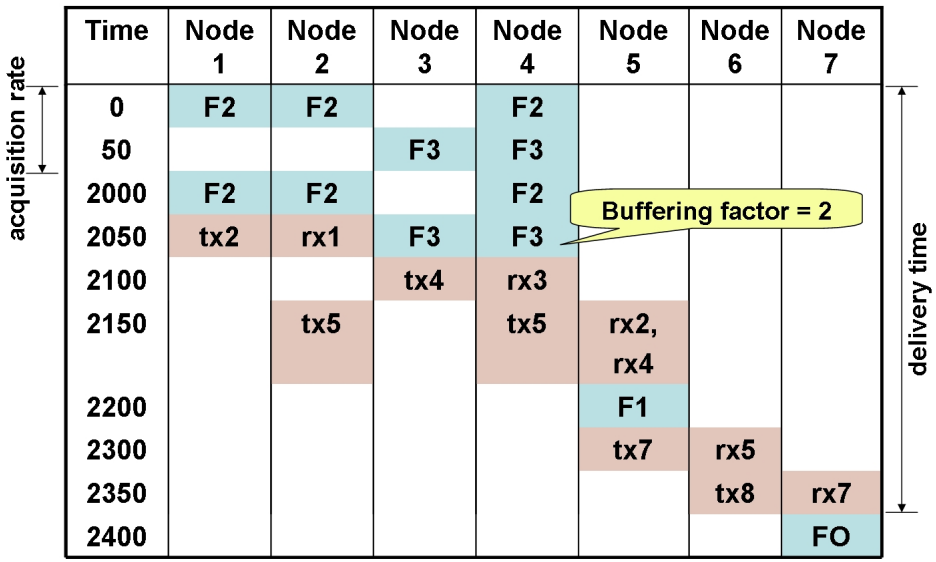
Example DAF for Q3



Query Processing in SNEE (11)
Execution Agenda for Example SNEE Query Q3

- The agenda computed for Q3 given the QoS expectations and the metadata above is given in the figure.
- Rows are identified by relative time, columns by sensor nodes/sites in the routing tree for the query.
- A cell indicates which action is performed:
  - *Fn* indicates that the denoted fragment executes at that time in that node.
  - *txn* indicates that that node at that time transmits data to node *n*, while *rxn* indicates that that node at that time receives data from node *n*.
  - The acquisition rate is reflected in the buffering factor (e.g., there is time to acquire twice and buffer before the first transmission).
- The delivery time can be read as the span of time it takes for data to be transmitted to the delivery node.

Example Agenda for Q3



Time	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7
0	F2	F2		F2			
50			F3	F3			
2000	F2	F2		F2			
2050	tx2	rx1	F3	F3			
2100			tx4	rx3			
2150		tx5		tx5	rx2, rx4		
2200					F1		
2300					tx7	rx5	
2350						tx8	rx7
2400							FO

Summary
Sensor Network Querying

- Query optimization in SNDM is quite distinct both in more clearly having multiple objectives and in the prominence of energy cost (as a prerequisite for longevity).
- TinyDB, the seminal first-generation SNQP engine, pioneered many ideas and insights on query optimization and execution in SNs.
- SNEE, a second-generation SNQP engine, differs from TinyDB in many respects, most fundamentally in viewing a SN as a distributed computing platform with all the implications this viewpoint suggests.
- The SNDM field is still being actively developed and the landscape is likely to change at a fast rate in the next few years.

Acknowledgements

The material presented mixes original material by the author as well as material adapted from tutorials and presentations by M. Tamer Özsu, Nick Koudas, Divesh Srivastava, Jennifer Widom, Lina Al-Jadir, Qiong Luo, Hejun Wu, Wei Hong, and Samuel Madden.
The author gratefully acknowledges the work of the authors cited while assuming complete responsibility any for mistake introduced in the adaptation of the material.

References

Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. B. (2003).
Aurora: a new model and architecture for data stream management.
VLDB J., 12(2):120–139.

Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., and Widom, J. (2004).
Stream: The stanford data stream management system.
`http://dbpubs.stanford.edu/pub/2004-20`.

Avnur, R. and Hellerstein, J. M. (2000).
Eddies: Continuously adaptive query processing.
In Chen, W., Naughton, J. F., and Bernstein, P. A., editors, *SIGMOD Conference*, pages 261–272. ACM.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. A. (2003).
Telegraphcq: Continuous dataflow processing for an uncertain world.
In *CIDR*.

Cranor, C. D., Johnson, T., Spatscheck, O., and Shkapenyuk, V. (2003).
Gigascope: A stream database for network applications.
In Halevy, A. Y., Ives, Z. G., and Doan, A., editors, *SIGMOD Conference*, pages 647–651. ACM.

Gay, D., Levis, P., von B ehren, J. R., Welsh, M., Brewer, E. A., and Culler, D. E. (2003).
The nesC language: A holistic approach to net worked embedded systems.
In *PLDI*, pages 1–11. ACM.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. E., and Pister, K. S. J. (2000).
System Architecture Directions for Networked Sensors.
In *ASPLOS*, pages 93–104.