Introduction to Wireless Sensor Network

Peter Scheuermann and Goce Trajcevski

Dept. of EECS
Northwestern University
A Database Primer
A leap in history

A brief overview/review of databases (DBMS’s)
- Aren’t they for payrolls, inventories, and transactions?

Databases are about providing a **declarative interface** to data processing/management
- Hides complexity and increases flexibility

Programming sensors is hard—can databases help?
- After all, it’s all about data
- But, as we will see, traditional databases don’t work well in this setting (good for us: lots of research problems!)
Two important questions

### What’s the right interface?
- Data model: How is data structured conceptually?
- Query language: How do users specify data processing/management tasks?

### How do you support this interface efficiently?
- Physical data organization: Store and index data in smart ways to speed up access
- Query processing and optimization: Figure out the most efficient method to carry out a given task
A simplified example

Data
- Nodes are uniquely identified by their ids
- They are deployed at fixed locations
- Each node generates readings (e.g., light, temperature, humidity) from the environment periodically, over time

Query
- Find nodes in a rectangular region $D$, where the temperature reading at time $t$ is higher than 40
- As a simplification, assume for now the base station has already collected all data
Without DBMS…

- Deployment configuration file
  - One node per line \((id, x, y)\), sorted by \(id\)

- Data log file
  - Each line contains \((id, timestamp, light, temperature, \ldots)\), sorted by \(timestamp\)

- To answer the query, write a program
  - In configuration file, find ids in \(D\), and remember them
  - Search log file for section for timestamp \(t\)
  - Scan the section for lines with qualified ids and temperature higher than 40
Tricks and Alternatives

- Indexes, e.g., on $t$ and on $temperature$?
- Change evaluation order, e.g., find temperature readings higher than 40 first?
  - When does this work better?
- Best choice may not be known in advance
- Problems with imperative programming
  - Burden on programmer to figure out right tricks/alternatives
  - To keep up with runtime characteristics, you need to reprogram your apps constantly!
Physical data independence

- Apps should not need to worry about how data is physically organized
- Apps should work with a logical data model and a declarative query language
- Specify what you want, not how to get it
- Leave implementation and optimization to DBMS!
Relational data model

- A database is a collection of relations (or tables)
- Each relation has a list of attributes (or columns)
- Each relation contains a set of tuples (or rows)

**Readings**

| id | time | light | temp |...
|----|------|-------|------|-----
| N1 | 1    | 3.14  | 26   |...
| N2 | 1    | 3.27  | 27   |...
| N3 | 1    | 2.97  | 26   |...
| N4 | 2    | 3.17  | 26   |...
| N4 | 2    | 2.99  | 25   |...
| N4 | 2    | 3.02  | 26   |...
| N4 | 2    | 3.02  | 26   |...

**Nodes**

<table>
<thead>
<tr>
<th>id</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>14.2</td>
<td>8.5</td>
</tr>
<tr>
<td>N2</td>
<td>7.1</td>
<td>-4.2</td>
</tr>
<tr>
<td>N3</td>
<td>-0.4</td>
<td>1.9</td>
</tr>
<tr>
<td>N4</td>
<td>3.1</td>
<td>-4.1</td>
</tr>
</tbody>
</table>

Key = \{id\}

Key = \{id, time\}
Key Constraint

Two rules for Key constraints:

- Two distinct tuples in a legal instance cannot have identical values in all columns of keys (unique)
- No subset of the set of fields in a key is a unique identifier for a tuple (maximal)

Example:

- No two nodes can have the same id
- No two measurements can have the same id and time

CORE IDEA: Minimal subset of columns of the relation that uniquely identify the tuple.
Relational algebra

A language for querying relational databases based on operators:

- Core set of operators: selection, projection, cross product, union, difference, and renaming
- Additional, derived operators: join, natural join, etc.
- Compose operators to make complex queries
Relational algebra operators

- **Selection**: $\sigma_p R$
  - Return only rows that satisfy selection condition $p$

- **Projection**: $\pi_C R$
  - Return all rows, but only with columns in $C$ (*eliminate duplicates!*)

- **Cross product**: $R \times S$
  - For every pair of rows from $R$ and $S$, return the concatenation

- **Union and difference**: $R \cup S$ and $R - S$

- **Rename**: $\rho_S R$, $\rho_{(A_1, A_2, \ldots)} R$ or $\rho^{S(A_1, A_2, \ldots)} R$
  - Rename a table and/or its columns

- **Join**: $R \bowtie_p S = \rho_p (R \times S)$

- **Natural join**: $R \bowtie S$
  - Equate common columns and keep one in output
Example query

**Given:**
- **Nodes**($id$, $x$, $y$), **Readings**($id$, $time$, light, temp, …)

**Find nodes in rectangular region** $D(x_l, y_l, x_h, y_h)$, where temperature at time $t$ is higher than 40

<table>
<thead>
<tr>
<th>$x_l \leq x \leq x_h$ and $y_l \leq y \leq y_h$</th>
<th>$\Sigma_{time = t}$ and $temp &gt; 40$</th>
</tr>
</thead>
</table>

\[
\pi_{id} \quad \Sigma_{x_l \leq x \leq x_h \text{ and } y_l \leq y \leq y_h} \quad \sigma_{time = t \text{ and } temp > 40}
\]
Structured Query Language: standard language spoken by most commercial DBMS

Simplest form:

```
SELECT A_1, A_2, ..., A_n
FROM R_1, R_2, ..., R_m
WHERE condition;
```

- $A_i$’s can be expressions in general

- Same as $\pi_{A_1, A_2, ..., A_n} (\sigma_{condition} (R_1 \times R_2 \times \ldots \times R_m))$
  - Except SQL preserves duplicates

- Also called an SPJ (select-project-join) query
**Same query in SQL**

- **Nodes** \((id, x, y)\), **Readings** \((id, time, light, temp, \ldots)\)
- Find nodes in rectangular region \(D(x_l, y_l, x_h, y_h)\), where temperature at time \(t\) is higher than 40

```sql
SELECT Nodes.id
FROM Nodes, Readings
WHERE x_l <= x AND x <= x_h
AND y_l <= y AND y <= y_h
AND time = t
AND temp > 40
AND Nodes.id = Readings.id;
```

*Compare this with an imperative program!*
More SQL features

```
SELECT [DISTINCT] list_of_output_exprs
FROM list_of_tables
WHERE where_condition
GROUP BY list_of_group_by_columns
HAVING having_condition
ORDER BY list_of_order_by_columns
```

Operational semantics

- **FROM**: take the cross product of `list_of_tables`
- **WHERE**: apply $\sigma_{where\_condition}$
- **GROUP BY**: group result tuples according to `list_of_group_by_columns`
- **HAVING**: apply $\sigma_{having\_condition}$ to groups
- **SELECT**: evaluate `list_of_output_exprs` for each output group
- **DISTINCT**: eliminate duplicates in output
- **ORDER BY**: sort output by `list_of_order_by_columns`
Aggregation example

- $\text{Nodes}(id, x, y)$, $\text{Readings}(id, \text{time}, \text{light}, \text{temp}, \ldots)$
- Average light over time, by nodes
  - SELECT id, AVG(light) FROM Readings GROUP BY id;

Compute GROUP BY: group rows according to the values of GROUP BY columns

<table>
<thead>
<tr>
<th>id</th>
<th>time</th>
<th>light</th>
<th>temp</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>1</td>
<td>3.14</td>
<td>26</td>
<td>...</td>
</tr>
<tr>
<td>N1</td>
<td>2</td>
<td>3.17</td>
<td>26</td>
<td>...</td>
</tr>
<tr>
<td>N2</td>
<td>1</td>
<td>3.27</td>
<td>27</td>
<td>...</td>
</tr>
<tr>
<td>N2</td>
<td>2</td>
<td>2.99</td>
<td>25</td>
<td>...</td>
</tr>
<tr>
<td>N3</td>
<td>1</td>
<td>2.97</td>
<td>26</td>
<td>...</td>
</tr>
<tr>
<td>N3</td>
<td>1</td>
<td>2.97</td>
<td>26</td>
<td>...</td>
</tr>
<tr>
<td>N1</td>
<td>2</td>
<td>3.17</td>
<td>26</td>
<td>...</td>
</tr>
<tr>
<td>N2</td>
<td>2</td>
<td>2.99</td>
<td>25</td>
<td>...</td>
</tr>
<tr>
<td>N3</td>
<td>2</td>
<td>3.02</td>
<td>26</td>
<td>...</td>
</tr>
</tbody>
</table>

Compute SELECT for each group

<table>
<thead>
<tr>
<th>id</th>
<th>avg_light</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>3.155</td>
</tr>
<tr>
<td>N2</td>
<td>3.13</td>
</tr>
<tr>
<td>N3</td>
<td>2.995</td>
</tr>
</tbody>
</table>
Summary of the relational interface

How is data structured conceptually?
- Simple tables (no order by design!)
- Rows “linked” by key values

How do users specify data processing/management tasks?
- Relational algebra: data flow of operators
- SQL: easier to write; even more declarative

Next: How do we support this interface efficiently?
Physical data organization

- Lay out data in various ways, e.g.:
  - Store *Nodes* hashed by *id*
  - Store *Readings* sorted by *time, id*

- Use auxiliary data structures
  - Index data to provide alternative access paths, e.g.:
    - R-tree index on *Nodes*(x, y)
    - B-tree index on *Readings*(light)
  - Materialize views of data, e.g.:
    - All temperatures higher than 40
      
      ```sql
      SELECT id, time, temperature FROM Readings WHERE temperature > 40;
      ```

- Basic trade-off?
B+ Tree Indexes

- Index leaf pages contain *data entries*, and are chained (prev & next)
- Index non-leaf pages have *index entries*; only used to direct searches:

![Diagram of B+ Tree Indexes]

- Non-leaf Pages
- Leaf Pages (Sorted by search key)

- Index leaf pages contain *data entries*, and are chained (prev & next)
- Index non-leaf pages have *index entries*; only used to direct searches:
Example B+ Tree

Find: 29*, 28*, All > 15* and < 30*

Insert/delete: Find data entry in leaf, then change it.

Note how data entries in leaf level are sorted.
Query processing and optimization

**SQL query**

```
SELECT x, y, time, light FROM Nodes, Readings WHERE Nodes.id = Readings.CID;
```

**Parse tree**

```
<Query>
    <SFW>
    <select-list>
        <from-list>
        <table>
        <table>
            Nodes
            Readings
    ...
```

**Logical plan**

- \( \pi \) \( x, y, time, light \)
- \( \sigma \) \( Nodes.id = Readings.id \)

**Physical plan**

- PROJECT \( (x, y, time, light) \)
- MERGE-JOIN \( (id) \)
- SORT \( (id) \)
- SCAN \( (Nodes) \)
- SCAN \( (Readings) \)

**Result**

**What you want**

**How to get it**
**Database Parameters**

- $|R|, |S| = \text{Number of pages in relations } R \text{ and } S \text{ respectively}$
- $||R||, ||S|| = \text{Number of tuples in relations } R \text{ and } S \text{ respectively}$
- $K = \text{no. of tuples per page}$
- $JS = \text{Join Selectivity Factor}$

- $JS = ||R \bowtie S|| / (||R|| \times ||S||)$
- $V(A, R) = \text{number of distinct values that appear in relation } R \text{ for attribute } A$
Estimating the Selectivity of Selection and Join

σ = Selection selectivity factor of relation R

- σ(A = value) = 1 / V(A, R)
- σ(A>value) = (max(A) – value) / (max(A) - min(A))
  - Max(A) (Min(A)) is the largest (smallest) value of A in R

||R θ S|| = min (||R||x||S|| / V(A, R), ||R||x||S|| / V(A, S))

For each tuple t ∈ R there are on the average
||S|| / V(A, S) tuples in S matching it
Join Techniques: $R \bowtie S$

- **Nested-loop Join Algorithm**
  
  For each block $b_r$ in $R$ do /* read blocks*/
  
  For each block $b_s$ in $S$ do
  
  For each tuple $r \in b_r$ do
  
  For each tuple $s \in b_s$ do
  
    if $r.a = s.b$ then output $r \bowtie s$

- **Cost of Method**

  $T_R = $ Number of Reads $= |R| + |R| \times |S|$

  $T_W = $ Number of Writes $= [J_s \times |R| \times |S| / K]$

  Cost can be lowered if index is available on $R$ and / or $S$
Physical plan operators

- One logical plan operator can be implemented in many different ways (physical plan operators)

Example: $R \bowtie_{R.A = S.B} S$

- Nested-loop join: for each tuple of $R$, and for each tuple of $S$, join
- Index nested-loop join: for each tuple of $R$, use the index on $S.B$ to find joining $S$ tuples
- Sort-merge join: sort $R$ by $R.A$, sort $S$ by $S.B$, and merge-join
- Hash join: partition $R$ and $S$ by hashing $R.A$ and $S.B$, and join corresponding partitions
- And many more…
Query optimization

- One query, many alternative physical plans
  - With different access methods, join order, join methods, etc.
  - With dramatically different costs too!

- Query optimization
  - Enumerate candidate plans
    - Query rewrite: transform queries or query plans into equivalent ones
  - Estimate costs of plans
    - Estimate result sizes using statistics such as histograms
  - Pick a plan with reasonably low cost
    - Dynamic programming
    - Randomized search
**Active Databases**

ECA = Basic Paradigm of the Reactive Behavior: *Triggers (Active Rules)*

```
ON       EVENT
IF       CONDITION
THEN     ACTION
```

seemingly straightforward, but incorporates an interplay of many *semantic dimensions*
Example of Triggers Execution

Assume that the Average Salary of the employees in a given enterprise should not exceed 65,000.

The *Database Modifications* that could cause a change of the value(s) of the Average Salary are:
- Insertions; (of new employees with high salary)
- Deletions; (deletions of employees with low salary)
- Update of the salaries of current employees

An example of the SQL statement specifying a trigger that would automatically correct the database state (if needed) after an *UPDATE* has been executed is:

```
CREATE TRIGGER Update-Salary-Check
ON UPDATE OF Employee.Salary
IF (SELECT AVG Employee.Salary) > 65,000
UPDATE Employee
  SET Employee.Salary = 0.95*Employee.Salary
```
Assume that it was decided to increase the salary of every employee in the “Maintenance” department by 10%. Following is the SQL statement:

```
UPDATE Employee
SET Employee.Salary = 1.10*Employee.Salary
WHERE Employee.Department = 'Maintenance'
```

Assume that there are 3 employees: Bob, Sam and Tom. Below is an example of two execution scenarios:
Back to sensor data processing…

- Does data really live in a few big, flat tables?
- Are data signals or symbols?
- Is SQL really enough?
- What would an index look like?
- What would a physical plan look like?
- How would the optimizer define “cost”??