A Database Primer

Jun Yang

CPS 296.1, Spring 2007

Sensor Data Processing

Announcements (Jan. 18)

- A note on paper reviews
  - Announced via email and posted on Web ~1 week in advance
  - 10% of your total grade
  - ~1 paper per lecture
  - Mini-review due before class; submit to Jun via email
    - Plain text only; use "[CPS296.1] review for …" in subject line
  - 3-2-1 format:
    - ≥ 3 important points the paper makes
    - ≥ 2 interesting points (e.g., non-obvious pitfall, neat insight…)
    - ≥ 1 thing you do not like
- Next Tuesday: TinyDB article by Madden et al.
- Verify mailing list membership

Today’s agenda

- 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 sensor 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 $(id, timestamp, light, temperature, …)$, 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? 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?
- Lots of tricks and alternatives
- 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 collection of relations (or tables)
- Each relation has a list of attributes (or columns)
- Each relation contains a set of tuples (or rows)

```
<table>
<thead>
<tr>
<th>Nodes 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>

<table>
<thead>
<tr>
<th>Readings id</th>
<th>time</th>
<th>light</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>1</td>
<td>3.14</td>
<td>26.1</td>
</tr>
<tr>
<td>N2</td>
<td>1</td>
<td>2.27</td>
<td>24.5</td>
</tr>
<tr>
<td>N3</td>
<td>2.97</td>
<td>24.9</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Key = \{id\}

(Okay, this is a grossly simplified design)
```

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 \( C \)
- 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: \( \rho_{A_1, A_2, ...} R \) or \( \rho_{(A_1, A_2, ...)} R \)
  - Rename a table and/or its columns
- Join: \( R \bowtie S = \sigma_p (R \times S) \)
- Natural join: \( R \bowtie S \)
  - Equate common columns and keep one in output

Example query

- Nodes(id, x, y), Readings(id, time, light, temp, ...)
- Find nodes in rectangular region \( D(xl, yl, xh, yh) \), where temperature at time \( t \) is higher than 40

\[
\sigma_{id \leq x \leq xh \text{ and } yl \leq y \leq yh} (\sigma_{time = t \text{ and } temp > 40} \pi_{id} \text{ Nodes Readerings})
\]
SQL

- Structured Query Language: standard language spoken by most commercial DBMS
- Simplest form:

\[
\begin{align*}
\text{SELECT } & A_1, A_2, \ldots, A_n \\
\text{FROM } & R_1, R_2, \ldots, R_m \\
\text{WHERE } & \text{condition};
\end{align*}
\]

- \(A_j\)'s can be expressions in general
- Same as \(\pi_{A_1, A_2, \ldots, A_n}(\sigma_{\text{condition}}(R_1 \times R_2 \times \ldots \times R_m))\)
- Except SQL preserves duplicates
- Also called an SPJ (select-project-join) query

More SQL features

```sql
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 \(\sigma_{\text{where_condition}}\)
- GROUP BY: group result tuples according to `list_of_group_by_columns`
- HAVING: apply \(\sigma_{\text{having_condition}}\) 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`

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.:
    - B-tree index on `Nodes(id, 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?
Query processing and optimization

Physical plan operators

Query optimization

Back to sensor data processing…