6.5830 Lecture 4

Database Design and Normalization

9.19.2022

PS 1 Due Wednesday
Today

• Recap Window Functions
• Schema normalization and normal forms
Window Functions

Compute the value of \text{window\_func} for each row of each partition

\[
\text{SELECT } x, y, \ldots, \text{window\_func}(\text{params}) \text{ OVER (PARTITION BY alist1 ORDER BY alist2)}
\]

Example:

\[
\text{SELECT hour, minute, RANK() OVER (ORDER BY hour, minute) FROM times}
\]

\text{Compute the rank of each row}

\[
\begin{array}{|c|c|}
\hline
\text{hour} & \text{minute} \\
\hline
1 & 15 \\
2 & 00 \\
4 & 30 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
\text{RANK} \\
\hline
1 \\
2 \\
3 \\
\hline
\end{array}
\]
### Window Functions

**Example:**

```sql
SELECT animalid, hour, minute, RANK() OVER (PARTITION BY animalid ORDER BY hour, minute) FROM times
```

**times (hour int, minute int, animalid int)**

<table>
<thead>
<tr>
<th>hour</th>
<th>min</th>
<th>animalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>00</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>00</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
<td>1</td>
</tr>
</tbody>
</table>

**animal | hour | minute**

<table>
<thead>
<tr>
<th>1</th>
<th>4</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
<th>1</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>00</td>
</tr>
</tbody>
</table>

**split by animal, compute the rank of each row**
Other Window Functions

• `cume_dist()` : cumulative position of the row (between 0 and 1) in total ordering

• `lag(value, offset)`: return the value for the record offset records before this one

• `sum() / count() / avg()`: sum / count / average of all rows in partition
  • For these expressions, OVER clause can include a `frame` that defines the subset of the partition to be included (Example on next slide)
Examples

Times with feed quantities

SELECT hour, min, cume_dist() OVER (ORDER BY hour, min) as c FROM times

<table>
<thead>
<tr>
<th>hour</th>
<th>min</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>0.25</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>0.75</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>1</td>
</tr>
</tbody>
</table>

SELECT hour, min, qty, lag(qty,1) OVER (ORDER BY hour, min) as lag FROM times

<table>
<thead>
<tr>
<th>hour</th>
<th>min</th>
<th>qty</th>
<th>lag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>10</td>
<td>40</td>
</tr>
</tbody>
</table>

SELECT hour, min, avg(qty) OVER (ORDER BY hour, min ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS rolling_avg FROM times

<table>
<thead>
<tr>
<th>hour</th>
<th>min</th>
<th>rolling_avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>26.67</td>
</tr>
</tbody>
</table>
• Write a SQL query with window function to compute the difference between sales a week ago and today

<table>
<thead>
<tr>
<th>Date</th>
<th>Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1/2022</td>
<td>5540</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1/8/2022</td>
<td>7000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1/15/2022</td>
<td>9000</td>
</tr>
</tbody>
</table>

Assume 1 row per day

Sales Table

Functions

• `rank()`: rank of items in ordering
• `cume_dist()`: cumulative position of the row (between 0 and 1) in total ordering
• `lag(value, offset)`: return the value for the record offset records before this one
• `sum() / count() / avg()`: sum / count / average of all rows in partition

Queries

SELECT hour, min, cume_dist() OVER (ORDER BY hour, min) as c FROM times

SELECT hour, min, avg(qty) OVER (ORDER BY hour, min ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS rolling_avg FROM times

SELECT hour, min, qty, lag(qty,1) OVER (ORDER BY hour, min) as lag FROM times
Write a SQL query with window function to compute the difference between sales a week ago and today:

```sql
SELECT date, sales, sales - lag(sales,7) OVER (ORDER BY date) difference FROM sales
```

Assume 1 row per day.
**Entity Relationship Modeling**

**Already Saw with Zoo**

Animals have names, ages, species
Keepers have names
Cages have cleaning times, buildings
Animals are in 1 cage; cages have multiple animals
Keepers keep multiple cages, cages kept by multiple keepers
ER Diagrams

Rectangle: Entity

Oval: Attribute

Underline: KEY

Relationship Cardinality

N:1 Each keeper keeps 0 or more cages
Each cage kept by 0 or 1 keeper

N:N Each keeper keeps 0 or more cages
Each cage kept by 0 or more keeper
More ER Modeling

**Recursive relationship**: Each keeper supervises 0 or more other supervisee keepers

**Relationship attributes**:
- Supervises
- Keeps
- Supervisee
- Supervisor
- KeepTime

**Double line**: Total participation
- Each cage kept by exactly one keeper

**Entity attributes**:
- Keeper
  - ID
  - Age
Converting to Relations

Keepers: (ID, age, supervisor REFERENCES Keepers.ID …)
Cages: (keptby NON NULL REFERENCES Keepers.id, keepTime, …)
What if we change the relationships to be N:N? (I.e., employees can have multiple supervisors, cages can be kept by multiple keepers?)
Keeps: (ID, age)
Cages: (CageID, ...)
Keeps: (kid, cid, keepTime)
Supervises: (supervisor_kid, supervisee_kid)
Hobbies Example

- Consider a database about people & their hobbies

**Entity Relationship Diagram**

- People have names and addresses, hobbies have costs.
- People can have multiple hobbies, and hobbies can be practiced by multiple people.
Hobby DB, Attempt 1

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>john</td>
<td>main st</td>
<td>dolls</td>
<td>$</td>
</tr>
<tr>
<td>123</td>
<td>john</td>
<td>main st</td>
<td>bugs</td>
<td>$</td>
</tr>
<tr>
<td>345</td>
<td>mary</td>
<td>lake st</td>
<td>tennis</td>
<td>$$</td>
</tr>
<tr>
<td>456</td>
<td>joe</td>
<td>first st</td>
<td>dolls</td>
<td>$</td>
</tr>
</tbody>
</table>

“Wide” schema
- has redundancy (wasted space)
- anomalies in the presence of updates, inserts, and deletes
+ avoids joins

Table key is Hobby, SSN
Types of Anomalies

• **Update anomaly**
  – E.g., address needs to change in several places
  – Creates possibility for inconsistency

• **Insertion anomaly** - what if we want to add someone with no hobby?
  – Can we use NULLs?
  – Problem: hobby is a part of the key!

• **Solution:** “Normalize”!
Hobby DB Attempt 2

- “Lossy decomposition”
- No redundancy, but we have lost some information (cost of hobbies)
Normalization

• Normalized: a schema that is redundancy free
  – As much as possible
• And that preserves dependencies
  – As much as possible
• Several methods:
  – ER Diagrams
  – Use functional dependencies and normal forms
### Schema From ER Diagram

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
<th>Cost</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>john</td>
<td>main st</td>
<td>dolls</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>john</td>
<td>main st</td>
<td>bugs</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>345</td>
<td>mary</td>
<td>lake st</td>
<td>tennis</td>
<td>$$</td>
<td></td>
</tr>
<tr>
<td>456</td>
<td>joe</td>
<td>first st</td>
<td>dolls</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>

SSN \(ightarrow\) Name, Address

Hobby, SSN \(\rightarrow\) Hobby, SSN

<table>
<thead>
<tr>
<th>Hobby</th>
<th>SSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>dolls</td>
<td>123</td>
</tr>
<tr>
<td>bugs</td>
<td>123</td>
</tr>
<tr>
<td>tennis</td>
<td>345</td>
</tr>
<tr>
<td>dolls</td>
<td>456</td>
</tr>
</tbody>
</table>

Schema is free of anomalies and redundancy

### Entity Relationship Diagram

- **SSN**
- **Address**
- **Name**
- **Hobby**
- **Cost**

- Person \(\text{n:n}\) Hobby

- SSN \(\rightarrow\) Person
- Name \(\rightarrow\) Person
- Address \(\rightarrow\) Person
- Hobby \(\rightarrow\) Person
- Cost \(\rightarrow\) Person
Why Does Redundancy Arise?

- When a subset of attributes are uniquely determined by a subkey
  - E.g., SSN determines name, address
  - Key is SSN, Hobby
  - Each row with same SSN will duplicate data!

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>john</td>
<td>main st</td>
<td>dolls</td>
<td>$</td>
</tr>
<tr>
<td>123</td>
<td>john</td>
<td>main st</td>
<td>bugs</td>
<td>$</td>
</tr>
<tr>
<td>345</td>
<td>mary</td>
<td>lake st</td>
<td>tennis</td>
<td>$$</td>
</tr>
<tr>
<td>456</td>
<td>joe</td>
<td>first st</td>
<td>dolls</td>
<td>$</td>
</tr>
</tbody>
</table>
Functional Dependencies

• $X \rightarrow Y$
• Attributes $X$ uniquely determine $Y$
  – i.e., for every pair of instances $x_1, x_2$ in $X$, with $y_1, y_2$ in $Y$, if $x_1 = x_2$, $y_1 = y_2$

• For Hobbies, we have:
  1. SSN, Hobby $\rightarrow$ Name, Addr, Cost
  2. SSN $\rightarrow$ Name, Addr
  3. Hobby $\rightarrow$ Cost

• 2 & 3 imply 1, by union ("Armstrong’s Axioms")
  – F1: $A \rightarrow B$, F2: $C \rightarrow D$, $F1 \cup F2 = A \cup C \rightarrow B \cup D$
FDs are a Property of the Application, Not the Data

- Can’t necessarily tell the FDs by looking at the data
- Given the FDs, can verify that the data satisfies them!
- Example: Is cost property of hobby or person?

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>john</td>
<td>main st</td>
<td>dolls</td>
<td>$</td>
</tr>
<tr>
<td>123</td>
<td>john</td>
<td>main st</td>
<td>bugs</td>
<td>$</td>
</tr>
<tr>
<td>345</td>
<td>mary</td>
<td>lake st</td>
<td>tennis</td>
<td>$$</td>
</tr>
<tr>
<td>456</td>
<td>joe</td>
<td>first st</td>
<td>dolls</td>
<td>$</td>
</tr>
</tbody>
</table>
Boyce-Codd Normal Form (BCNF)

- For a relation R, with FDs of the form $X \rightarrow Y$, it is in BCNF iff
  
  Every FD is either:
  1) Trivial (e.g., Y contains X) ($SSN \rightarrow SSN, Name$)
  2) X is a key of the table

- If an FD violates 2), multiple rows with same X value may occur
  - Indicates redundancy, as rows with given X value all have same Y value
  - E.g., $SSN \rightarrow Name, Addr$ in non-decomposed hobbies schema
    - $SSN$ is not a key of the whole table
    - $Name, Addr$ repeated for each appearance of a given $SSN$

- To put a schema into BCNF, create subtables of form $XY$
  - E.g., tables where key is left side (X) of one or more FDs
  - Repeat until all tables in BCNF
  - Effectively eliminates redundancy, while preserving (most) dependencies
BCNFify

BCNFify(schema \(R\), functional dependency set \(F\)):

\[D = \{(R,F)\}\]  // \(D\) is set of output relations

while there is a (schema ,FD set) pair \((S,F')\) in \(D\) not in BCNF, do:

given \(X \rightarrow Y\) as a BCNF-violating dependency in \(F'\)

replace \((S,F')\) in \(D\) with

\[S_1 = (XY,F_1)\] and
\[S_2 = ((S-Y) \cup X, F_2)\]

where \(F_1\) and \(F_2\) are the FDs in \(F'\) over \(XY\) or \((S-Y) \cup X\), respectively

End

return \(D\)
BCNFify Example for Hobbies

Iter 1
S = SSN, H = Hobby, N = Name, A = Addr, C = Cost

Did we lose S,H → N,A,C? No!
S,H → N, A, C is implied by H→C and S→N,A

No, using union rule from Armstrong’s Axioms,

S+H -> N,A + C

Iter 2

3 remaining tables are same as in ER decomposition
Account, Client, Office

• FD’s
  – Client, Office $\rightarrow$ Account
    • Each client/office has only one account
  – Account $\rightarrow$ Office
    • All records for a given account at same office

<table>
<thead>
<tr>
<th>Account</th>
<th>Client</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>joe</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>mary</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>john</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>joe</td>
<td>2</td>
</tr>
</tbody>
</table>
Account, Client, Office

• FD’s
  – Client, Office → Account
    • Each client/office has only one account
  – Account → Office
    • All records for a given account at same office

<table>
<thead>
<tr>
<th>Account</th>
<th>Client</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>joe</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>mary</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>john</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>joe</td>
<td>2</td>
</tr>
</tbody>
</table>

Redundancy!
A Dilemma

- This is not in BCNF
- Put account, office in separate tables?

<table>
<thead>
<tr>
<th>ao_id</th>
<th>Account</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ao_id</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>joe</td>
</tr>
<tr>
<td>2</td>
<td>mary</td>
</tr>
<tr>
<td>1</td>
<td>john</td>
</tr>
<tr>
<td>3</td>
<td>joe</td>
</tr>
</tbody>
</table>

BCNF, but can’t check CO → A without a join

FD’s
- Client, Office → Account
- Account → Office
BCNF vs 3NF

- BCNF is not “dependency preserving”
- 3rd Normal Form (3NF) eliminates as much redundancy as possible while preserving all dependencies
  - We will skip the details
- Neither form is “better”
  - You can choose either dependency preservation (3NF) or redundancy-free (BCNF)
Study Break

• Patient database
• Want to represent patients at hospitals with doctors
• Patients have names, birthdates
• Doctors have names, specialties
• Hospitals have names, addresses
• One doctor can treat multiple patients, each patient has one doctor
• Each patient in one hospital, hospitals have many patients

1) Draw an ER diagram

2) What are the functional dependencies

3) What is the normalized schema? Is it redundancy free?
Solution

Patients have names, birthdates
Doctors have names, specialties
Hospitals have names, addresses
One doctor can treat multiple patients, each patient has one doctor
Each patient in one hospital, hospitals have many patients

Pid → Pname, Bday, P_Did, P_Hid
Hid → Hname, Addr
Did → Dname, Specialty

Patients (Pid, Pname, Bday, P_Did, P_Hid)
Hospitals (Hid, Hname, Addr)
Doctors (Did, Dname, Specialty)
Recap

• Properly normalized schemas avoid redundancy and preserve dependencies
• Functional dependencies and normal forms (e.g., BCNF) give us a formal way to reason about these concepts
• In practice, people use ER modeling to derive a schema in BCNF rather than the BCNFify algorithm