Outline

- Relational Algebra (6.1)
- E/R Model (7.2 - 7.4)
- E/R Diagrams (7.5)
- Reduction to Schema (7.6)
- Relational Database Design (7.7)
- Functional Dependencies (8.1 – 8.4)
- Normalization (8.5 – 8.7)

A Movie Industry Schema
Relational Database Design

- Where did we come up with the schema that we used?
  - E.g. why not store the actor names with movies?

- If from an E-R diagram, then:
  - Did we make the right decisions with the E-R diagram?

- Goals:
  - Formal definition of what it means to be a “good” schema.
  - How to achieve it.

Relational Schemas and Redundancy

- movies(name, year, len)
- stars(name, addr, gender, birthdate)
- execs(name, cert#)
- studios(stud_name, address)

- in(star_name, movie_name, movie_year)
- made_by(movie_name, movie_year)
- produced_by(movie_name, movie_year, cert#)
- helmed_by(cert#, stud_name)
Relational Schemas and Redundancy

- movies(name, year, len)
- stars(name, addr, gender, birthdate)
- execs(name, cert#)
- studios(stud_name, address, pres#)
- in(star_name, movie_name, movie_year)
- made_by(movie_name, movie_year)
- produced_by(movie_name, movie_year, cert#)

Relational Schemas and Redundancy

- movies(name, year, len, prod#)
- stars(name, addr, gender, birthdate)
- execs(name, cert#)
- studios(stud_name, address, pres#)
- in(star_name, movie_name, movie_year)
- made_by(movie_name, movie_year)
Relational Schemas and Redundancy

- movies(name, year, len, prod#, studio_name)
- stars(name, addr, gender, birthdate)
- execs(name, cert#)
- studios(stud_name, address, pres#)
- in(star_name, movie_name, movie_year)
Relational Database Design

or

“Troubles With Schemas“

Outline

- Relational Algebra (6.1)
- E/R Model (7.2 - 7.4)
- E/R Diagrams (7.5)
- Reduction to Schema (7.6)
- Relational Database Design (7.7)
- Functional Dependencies (8.1 – 8.4)
- Normalization (8.5 – 8.7)
### Movie Table:

<table>
<thead>
<tr>
<th>Title</th>
<th>Year</th>
<th>Length</th>
<th>inColor</th>
<th>StudioName</th>
<th>prodC#</th>
<th>StarName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star wars</td>
<td>1977</td>
<td>121</td>
<td>Yes</td>
<td>Fox</td>
<td>128</td>
<td>Hamill</td>
</tr>
<tr>
<td>Star wars</td>
<td>1977</td>
<td>121</td>
<td>Yes</td>
<td>Fox</td>
<td>128</td>
<td>Fisher</td>
</tr>
<tr>
<td>Star wars</td>
<td>1977</td>
<td>121</td>
<td>Yes</td>
<td>Fox</td>
<td>128</td>
<td>H. Ford</td>
</tr>
<tr>
<td>King Kong</td>
<td>2005</td>
<td>187</td>
<td>Yes</td>
<td>Universal</td>
<td>150</td>
<td>Watts</td>
</tr>
<tr>
<td>King Kong</td>
<td>1933</td>
<td>100</td>
<td>no</td>
<td>RKO</td>
<td>20</td>
<td>Fay</td>
</tr>
</tbody>
</table>

### Issues:

1. **Redundancy** ➔ higher storage, inconsistencies ("anomalies")
   
   *update anomalies, insertion anomalies*

2. **Need nulls**
   
   Unable to represent some information without using nulls
   
   *How to store movies w/o actors (pre-productions etc)*?

### Movie Table with Star Names:

<table>
<thead>
<tr>
<th>Title</th>
<th>Year</th>
<th>Length</th>
<th>inColor</th>
<th>StudioName</th>
<th>prodC#</th>
<th>StarNames</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star wars</td>
<td>1977</td>
<td>121</td>
<td>Yes</td>
<td>Fox</td>
<td>128</td>
<td>{Hamill, Fisher, H. Ford}</td>
</tr>
<tr>
<td>King Kong</td>
<td>2005</td>
<td>187</td>
<td>Yes</td>
<td>Universal</td>
<td>150</td>
<td>Watts</td>
</tr>
<tr>
<td>King Kong</td>
<td>1933</td>
<td>100</td>
<td>no</td>
<td>RKO</td>
<td>20</td>
<td>Fay</td>
</tr>
</tbody>
</table>

### Issues:

3. **Avoid sets**
   
   - Hard to represent
   
   - Hard to query
Smaller schemas always good ????

Split Studio(*name*, *address*, presC#) into:
   Studio1 (*name*, presC#),
   Studio2(*name*, *address*)

<table>
<thead>
<tr>
<th>Name</th>
<th>presC#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fox</td>
<td>101</td>
</tr>
<tr>
<td>Studio2</td>
<td>101</td>
</tr>
<tr>
<td>Universial</td>
<td>102</td>
</tr>
</tbody>
</table>

This process is also called “decomposition”

**Issues:**
4. Requires more joins (w/o any obvious benefits)
5. Hard to check for some dependencies
   - What if the “address” is actually the presC#’s address?
   - No easy way to ensure that constraint (w/o a join).

Smaller schemas always good ????

Decompose StarsIn(*movieTitle*, *movieYear*, *starName*) into:
   StarsIn1(*movieTitle*, *movieYear*)
   StarsIn2(*movieTitle*, *starName*)

<table>
<thead>
<tr>
<th>movieTitle</th>
<th>movieYear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star wars</td>
<td>1977</td>
</tr>
<tr>
<td>King Kong</td>
<td>1933</td>
</tr>
<tr>
<td>King Kong</td>
<td>2005</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>movieTitle</th>
<th>starName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star Wars</td>
<td>Hamill</td>
</tr>
<tr>
<td>King Kong</td>
<td>Watts</td>
</tr>
<tr>
<td>King Kong</td>
<td>Faye</td>
</tr>
</tbody>
</table>

**Issues:**
6. “joining” them back results in more tuples than what we started with
   - (King Kong, 1933, Watts) & (King Kong, 2005, Faye)
   - This is a “lossy” decomposition
     - We lost some constraints/information
     - The previous example was a “lossless” decomposition.
Desiderata

- No sets
- Correct and faithful to the original design
  - Avoid lossy decompositions
- As little redundancy as possible
  - To avoid potential anomalies
- No “inability to represent information”
  - Nulls shouldn’t be required to store information
- Dependency preservation
  - Should be possible to check for constraints

Not always possible.
We sometimes relax these for:

* simpler schemas, and fewer joins during queries.*

Functional Dependencies!
Outline

- Relational Algebra (6.1)
- E/R Model (7.2 - 7.4)
- E/R Diagrams (7.5)
- Reduction to Schema (7.6)
- Relational Database Design (7.7)
- Functional Dependencies (8.1 – 8.4)
- Normalization (8.5 – 8.7)

Outline

- Mechanisms and definitions to work with FDs
  - Closures, candidate keys, canonical covers etc...
  - Armstrong axioms
- Decompositions
  - Loss-less decompositions, Dependency-preserving decompositions
- BCNF
  - How to achieve a BCNF schema
- BCNF may not preserve dependencies
- 3NF: Solves the above problem
- BCNF allows for redundancy
- 4NF: Solves the above problem
Approach

1. We will encode and list all our knowledge about the schema
   - Functional dependencies (FDs)
     - $\text{SSN} \rightarrow \text{name}$ (means: SSN “implies” (“determines”) length)
     - If two tuples have the same “SSN”, they must have the same “name”
     - $\text{movietitle} \rightarrow \text{length}$ ??? Not true.
     - But, $(\text{movietitle, movieYear, movieDirector}) \rightarrow \text{length}$ --- True.

2. We will define a set of rules that the schema must follow to be “good”
   - “Normal forms”: 1NF, 2NF, 3NF, BCNF, 4NF, ...
   - A normal form specifies constraints on the schemas and FDs

3. If not in a “normal form”, we modify the schema

FDs: Example

<table>
<thead>
<tr>
<th>Course ID</th>
<th>Course Name</th>
<th>Dept Name</th>
<th>Credits</th>
<th>Section ID</th>
<th>Semester</th>
<th>Year</th>
<th>Building</th>
<th>Room No.</th>
<th>Capacity</th>
<th>Time Slot ID</th>
</tr>
</thead>
</table>

Functional dependencies:

- course_id ➔
- building, room_number ➔
- course_id, section_id, semester, year ➔
Functional Dependencies

- Let $r(R)$ be a relation schema and
  \[ \alpha \subseteq R \text{ and } \beta \subseteq R \]
- The \textit{functional dependency}
  \[ \alpha \rightarrow \beta \]
  \text{holds on } R \text{ iff for any legal relations } r(R), \text{ whenever two tuples } t_1 \text{ and } t_2 \text{ of } r \text{ have same values for } \alpha, \text{ they have same values for } \beta.
  \[ t_1[\alpha] = t_2[\alpha] \implies t_1[\beta] = t_2[\beta] \]
- Example:

- On this instance, $A \rightarrow B$ does NOT hold, but $B \rightarrow A$ does hold.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Functional Dependencies

- \textbf{Difference between holding on an instance and holding on all legal relations}

<table>
<thead>
<tr>
<th>Title</th>
<th>Year</th>
<th>Length</th>
<th>inColor</th>
<th>StudioName</th>
<th>prodC#</th>
<th>StarName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Star wars</td>
<td>1977</td>
<td>121</td>
<td>Yes</td>
<td>Fox</td>
<td>128</td>
<td>Hamill</td>
</tr>
<tr>
<td>Star wars</td>
<td>1977</td>
<td>121</td>
<td>Yes</td>
<td>Fox</td>
<td>128</td>
<td>Fisher</td>
</tr>
<tr>
<td>Star wars</td>
<td>1977</td>
<td>121</td>
<td>Yes</td>
<td>Fox</td>
<td>128</td>
<td>H. Ford</td>
</tr>
<tr>
<td>King Kong</td>
<td>1933</td>
<td>100</td>
<td>no</td>
<td>RKO</td>
<td>20</td>
<td>Fay</td>
</tr>
</tbody>
</table>

- \textit{Title} $\rightarrow$ \textit{Year} \text{ holds on this instance}
- \textit{Is this a true functional dependency? No.}
  - \textit{Two movies in different years can have the same name.}
- Can’t draw conclusions based on a single instance
  - Need \textit{domain knowledge to decide which FDs hold}
FDs and Redundancy

- Consider a table: \( R(A, B, C) \):
  - With FDs: \( B \rightarrow C \), and \( A \rightarrow BC \)
  - So “A” is a Key, but “B” is not
- So: there is a FD whose left hand side is not a key
  - Leads to redundancy

Since B is not unique, it may be duplicated
Every time B is duplicated, so is C

Not a problem with \( A \rightarrow BC \)
A can never be duplicated

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>a2</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>a3</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>a4</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>a5</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>a6</td>
<td>b3</td>
<td>c3</td>
</tr>
<tr>
<td>a7</td>
<td>b4</td>
<td>c1</td>
</tr>
</tbody>
</table>

Not a duplication → Two different tuples just happen to have the same value for C

Functional Dependencies

- Functional dependencies and keys:
  - A key constraint is a specific form of a FD.
  - E.g. if \( \alpha \) is a superkey for \( R \), then:
    \[ \alpha \rightarrow R \]
  - Similarly for candidate keys and primary keys.

- Deriving FDs
  - A set of FDs may imply other FDs
    - e.g. If \( A \rightarrow B \), and \( B \rightarrow C \), then clearly \( A \rightarrow C \)
    - We will see a formal method for inferring this later
Definitions

1. A relation instance $r$ satisfies a set of functional dependencies, $F$, if the FDs hold on the relation.
2. $F$ holds on a relation schema $R$ if no legal (allowable) relation instance of $R$ violates it.
3. A functional dependency, $\alpha \rightarrow \beta$, is called trivial if:
   - $\alpha$ is a superset of $\beta$
   - e.g. Movieyear, length $\rightarrow$ length
4. Given a set of functional dependencies, $F$, its closure, $F^+$, is all the FDs that are implied by FDs in $F$.

Approach

1. We will encode and list all our knowledge about the schema
   - Functional dependencies (FDs)
   - Also:
     - Multi-valued dependencies (briefly discuss later)
     - Join dependencies etc...
2. We will define a set of rules that the schema must follow to be considered good
   - “Normal forms”: 1NF, 2NF, BCNF, 3NF, 4NF, ...
   - A normal form specifies constraints on the schemas and FDs
3. If not in a “normal form”, we modify the schema
BCNF: Boyce-Codd Normal Form

- A relation schema $R$ is “in BCNF” if:
  - Every functional dependency $\alpha \rightarrow \beta$ that holds on it is **EITHER**:
    1. Trivial **OR**
    2. $\alpha$ is a superkey of $R$

- **Why is BCNF good?**
  - Guarantees that there can be no redundancy because of a functional dependency
  - Consider a relation $r(A, B, C, D)$ with functional dependency
    - $A \rightarrow B$ and two tuples: $(a_1, b_1, c_1, d_1)$, and $(a_1, b_1, c_2, d_2)$
      - $b_1$ is repeated because of the functional dependency
      - BUT this relation is not in BCNF
        - $A \rightarrow B$ is neither trivial nor is $A$ a superkey for the relation

BCNF and Redundancy

- **Why does redundancy arise?**
  - Given a FD, $\alpha \rightarrow \beta$, if $\alpha$ is repeated $(\beta - \alpha)$ has to be repeated
    1. If rule 1 is satisfied, $(\beta - \alpha)$ is empty, so not a problem.
    2. If rule 2 is satisfied, then $\alpha$ can’t be repeated, so this doesn’t happen either

- Hence no redundancy because of FDs in BCNF
  - Redundancy may exist because of other types of dependencies
    - Higher normal forms used for that (specifically, 4NF)
  - Data may naturally have duplicated/redundant data
    - We can’t control that unless a FD or some other dependency is defined
**Approach**

1. We will encode and list all our knowledge about the schema:
   - Functional dependencies (FDs); Multi-valued dependencies; Join dependencies etc...

2. We will define rules the schema must follow to be “good”
   - “Normal forms”: 1NF, 2NF, 3NF, BCNF, 4NF, ...
   - A normal form specifies constraints on the schemas and FDs

3. If not in a “normal form”, we modify the schema
   - Through lossless decomposition (splitting)
   - Or direct construction using the dependencies information

**BCNF**

- What if the schema is not in BCNF?
  - *Decompose (split) the schema into two pieces.*

- From the previous example: split the schema into:
  - \( r_1(A, B), r_2(A, C, D) \)
  - The first schema is in BCNF, the second one may not be (and may require further decomposition)
  - No repetition now: \( r_1 \) contains \((a1, b1)\), but \( b1 \) will not be repeated

- Careful: you want the decomposition to be **lossless**
  - *No information should be lost*
    - The above decomposition is lossless
  - We will define this more formally later

---

\( r(A, B, C, D) \) with \( A \rightarrow B \) and:
- \((a1, b1, c1, d1)\), and \((a1, b1, c2, d2)\)
1. Closure of Functional Dependencies

- Given a set of functional dependencies, \( F \), its closure, \( F^+ \), is all FDs that are implied by FDs in \( F \).
  - *e.g.* If \( A \rightarrow B \), and \( B \rightarrow C \), then clearly \( A \rightarrow C \)

- We can find \( F^+ \) by applying **Armstrong’s Axioms**:  
  - if \( \beta \subseteq \alpha \), then \( \alpha \rightarrow \beta \)  
    (reflexivity)
  - if \( \alpha \rightarrow \beta \), then \( \gamma \alpha \rightarrow \gamma \beta \)  
    (augmentation)
  - if \( \alpha \rightarrow \beta \), and \( \beta \rightarrow \gamma \), then \( \alpha \rightarrow \gamma \)  
    (transitivity)

- These rules are
  - sound (generate only functional dependencies that actually hold)
  - complete (generate all functional dependencies that hold)
Additional rules (not Armstrong’s axioms)

- If $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$, then $\alpha \rightarrow \beta \gamma$ (union)
- If $\alpha \rightarrow \beta \gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ (decomposition)
- If $\alpha \rightarrow \beta$ and $\gamma \beta \rightarrow \delta$, then $\alpha \gamma \rightarrow \delta$ (pseudotransitivity)

- The above rules can be inferred from Armstrong’s axioms.

Example (only Armstrong’s axioms)

- $F = \{ A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
- Some members of $F^+$
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ and $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with $G$, to get $AG \rightarrow CG$
    - and then transitivity with $AG \rightarrow CG \rightarrow I$
  - $CG \rightarrow HI$
    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$
    - and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$
    - and then transitivity: $CG \rightarrow CGI \rightarrow HI$

DONE