The Relational Model

Mine eye hath play’d the painter and hath stell’d
Thy beauty’s form in table of my heart.

Shakespeare, Sonnet XXIV

Why Study the Relational Model?

- Most widely used model.
  - Vendors: IBM, Informix, Microsoft, Oracle, Sybase, etc.
- "Legacy systems" in older models
  - e.g., IBM’s IMS
- Object-oriented concepts have recently merged in
  - object-relational model
    - Informix, IBM DB2, Oracle 8i
    - Early work done in POSTGRES research project at Berkeley

Relational Database: Definitions

- Relational database: a set of relations.
- Relation: made up of 2 parts:
  - Schema: specifies name of relation, plus name and type of each column.
    - E.g. Students(sid: string, name: string, login: string, age: integer, gpa: real)
  - Instance: a table, with rows and columns.
    - #rows = cardinality
    - #fields = degree/arity
- Can think of a relation as a set of rows or tuples.
  - i.e., all rows are distinct

Ex: Instance of Students Relation

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@cs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

- Cardinality = 3, arity = 5, all rows distinct
- Do all values in each column of a relation instance have to be distinct?

SQL - A language for Relational DBs

- SQL* (a.k.a. “Sequel”), standard language
- Data Definition Language (DDL)
  - create, modify, delete relations
  - specify constraints
  - administer users, security, etc.
- Data Manipulation Language (DML)
  - Specify queries to find tuples that satisfy criteria
  - add, modify, remove tuples

SQL Overview

- CREATE TABLE <name> ( <field> <domain>, ... )
- INSERT INTO <name> ( <field names> ) VALUES ( <field values> )
- DELETE FROM <name> WHERE <condition>
- UPDATE <name>
  - SET <field name> = <value> WHERE <condition>
- SELECT <fields>
  - FROM <name> WHERE <condition>

*Structured Query Language
Creating Relations in SQL

• Creates the Students relation.
  --Note: the type (domain) of each field is specified, and enforced by the DBMS whenever tuples are added or modified.

```
CREATE TABLE Students
(sid CHAR(20),
name CHAR(20),
login CHAR(10),
age INTEGER,
gpa FLOAT)
```

Table Creation (continued)

• Another example: the Enrolled table holds information about courses students take.

```
CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2))
```

Adding and Deleting Tuples

• Can insert a single tuple using:

```
INSERT INTO Students (sid, name, login, age, gpa)
VALUES ('53666', 'Smith', 'smith@cs', 18, 3.2)
```

• Can delete all tuples satisfying some condition (e.g., name = Smith):

```
DELETE
FROM Students S
WHERE S.name = 'Smith'
```

Powerful variants of these commands are available; more later!

Keys

• Keys are a way to associate tuples in different relations

• Keys are one form of integrity constraint (IC)

```
Enrolled

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53650</td>
<td>15-112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>15-103</td>
<td>B</td>
</tr>
<tr>
<td>53668</td>
<td>15-101</td>
<td>C</td>
</tr>
</tbody>
</table>

Students

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53650</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53666</td>
<td>Smith</td>
<td>smith@cs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53668</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>
```

Primary Keys

• A set of fields is a **superkey** if:
  -- No two distinct tuples can have same values in all key fields

• A set of fields is a **key** for a relation if:
  -- It is a superkey
  -- No subset of the fields is a superkey

• what if >1 key for a relation?
  -- one of the keys is chosen (by DBA) to be the **primary key**. Other keys are called **candidate** keys.

• E.g.
  -- sid is a key for Students.
  -- What about name?
  -- The set (sid, gpa) is a superkey.

Primary and Candidate Keys in SQL

• Possibly many **candidate keys** (specified using UNIQUE), one of which is chosen as the **primary key**.
  -- Keys must be used carefully!
  -- “For a given student and course, there is a single grade.”

```
CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid, cid))

CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid),
UNIQUE (cid, grade))
```

"Students can take only one course, and no two students in a course receive the same grade."
Foreign Keys, Referential Integrity

- **Foreign key**: Set of fields in one relation that is used to ‘refer’ to a tuple in another relation.
  - Must correspond to the primary key of the other relation.
  - Like a ‘logical pointer’.
- If all foreign key constraints are enforced, **referential integrity** is achieved (i.e., no dangling references.)

Enforcing Referential Integrity

- Consider Students and Enrolled; sid in Enrolled is a foreign key that references Students.
- What should be done if an Enrolled tuple with a non-existent student id is inserted? (Reject it)
- What should be done if a Students tuple is deleted?
  - Also delete all Enrolled tuples that refer to it?
  - Disallow deletion of a Students tuple that is referred to?
  - Set sid in Enrolled tuples that refer to it to a default sid?
  - (In SQL, also: Set sid in Enrolled tuples that refer to it to a special value null, denoting ‘unknown’or ‘inapplicable’)
- Similar issues arise if primary key of Students tuple is updated.

Foreign Keys in SQL

Example: Only students listed in the Students relation should be allowed to enroll for courses.
- sid is a foreign key referring to Students:

  ```sql
  CREATE TABLE Enrolled
  (sid CHAR(20), cid CHAR(20), grade CHAR(2),
   PRIMARY KEY (sid,cid),
   FOREIGN KEY (sid) REFERENCES Students )
  ```

<table>
<thead>
<tr>
<th>Enrolled</th>
<th>Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>cid</td>
</tr>
<tr>
<td>53666</td>
<td>15-101</td>
</tr>
<tr>
<td>53666</td>
<td>19-203</td>
</tr>
<tr>
<td>53650</td>
<td>15-112</td>
</tr>
</tbody>
</table>

Integrity Constraints (ICs)

- **IC**: condition that must be true for *any* instance of the database; e.g., **domain constraints**.
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.
- A legal instance of a relation is one that satisfies all specified ICs.
  - DBMS should not allow illegal instances.
- If the DBMS checks ICs, stored data is more faithful to real-world meaning.
  - Avoids data entry errors, too!

Where do ICs Come From?

- ICs are based upon the semantics of the real-world that is being described in the database relations.
- We can check a database instance to see if an IC is violated, but we can **NEVER** infer that an IC is true by looking at an instance.
  - An IC is a statement about *all possible* instances!
  - From example, we know name is not a key, but the assertion that sid is a key is given to us.
- Key and foreign key ICs are the most common; more general ICs supported too.

Relational Model: Summary

- A tabular representation of data.
- Simple and intuitive, currently the most widely used
  - Object-relational variant gaining ground
- Integrity constraints can be specified by the DBA, based on application semantics. DBMS checks for violations.
  - Two important ICs: primary and foreign keys
  - In addition, we always have domain constraints.
- Mapping from ER to Relational is (fairly) straightforward.

- NEXT: FILES< STORAGE, BUFFERS, DISKS...
- READ CHAPTER 9!
Nobody realizes that some people expend tremendous energy merely to be normal.

---

**Schema Refinement and Normalization**

**Functional Dependencies (FDs)**
- A functional dependency \( X \rightarrow Y \) holds over relation schema \( R \) if, for every allowable instance \( r \) of \( R \):
  \[
  t_1 \in r, \ t_2 \in r, \ \pi_Y(t_1) = \pi_Y(t_2) \quad \text{implies} \quad \pi_Y(t_1) = \pi_Y(t_2)
  \]
  (where \( t_1 \) and \( t_2 \) are tuples; \( X \) and \( Y \) are sets of attributes)
- In other words: \( X \rightarrow Y \) means
  Given any two tuples in \( r \), if the \( X \) values are the same, then the \( Y \) values must also be the same. (but not vice versa)
- Can read “\( \rightarrow \)” as “determines”

---

**FD’s Continued**
- An FD is a statement about all allowable relations.
  - Must be identified based on semantics of application.
  - Given some instance \( r \) of \( R \), we can check if \( r \) violates some FD \( f \), but we cannot determine if \( f \) holds over \( R \).
- Question: How related to keys?
- If “\( K \rightarrow \) all attributes of \( R \)” then \( K \) is a superkey for \( R \)
  (does not require \( K \) to be minimal)
- FDs are a generalization of keys.

---

**Normal Forms**
- Back to schema refinement...
- Q1: is any refinement needed??!
- If a relation is in a normal form (BCNF, 3NF etc.):
  - we know that certain problems are avoided/minimized.
  - helps decide whether decomposing a relation is useful.
- Role of FDs in detecting redundancy:
  - Consider a relation \( R \) with 3 attributes, ABC.
    - No (non-trivial) FDs hold: There is no redundancy here.
    - Given \( A \rightarrow B \): If \( A \) is not a key, then several tuples could have the same \( A \) value, and if so, they’ll all have the same \( B \) value!
- 1\textsuperscript{st} Normal Form – all attributes are atomic
- 1\textsuperscript{st} \( \supseteq \) 2\textsuperscript{nd} (of historical interest) \( \supset 3\textsuperscript{rd} \) Boyce-Codd \( \supseteq \) ...

---

**Boyce-Codd Normal Form (BCNF)**
- Reln \( R \) with FDs \( F \) is in BCNF if, for all \( X \rightarrow Y \) in \( F \):
  - \( A \in X \) (called a trivial FD), or
  - \( X \) is a superkey for \( R \).
- In other words: “\( R \) is in BCNF if the only non-trivial FDs over \( R \) are key constraints.”
- If \( R \) in BCNF, then every field of every tuple records information that cannot be inferred using FDs alone.
  - Say we know \( RX \rightarrow A \) holds this example relation:
    | X | Y | A |
    |---|---|---|
    | x | y1 | a |
    | x | y2 | ? |
  - Can you guess the value of the missing attribute?
  - Yes, so relation is not in BCNF

---

**Decomposition of a Relation Scheme**
- If a relation is not in a desired normal form, it can be decomposed into multiple relations that each are in that normal form.
- Suppose that relation \( R \) contains attributes \( A_1 \ldots A_n \). A decomposition of \( R \) consists of replacing \( R \) by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of \( R \), and
  - Every attribute of \( R \) appears as an attribute of at least one of the new relations.
Example (same as before)

SNLWRH has FDs S \rightarrow SNLWRH and R \rightarrow W

Q: Is this relation in BCNF?

No. The second FD causes a violation: W values repeatedly associated with R values.

Decomposing a Relation

Easiest fix is to create a relation RW to store these associations, and to remove W from the main schema:

<table>
<thead>
<tr>
<th>S</th>
<th>N</th>
<th>L</th>
<th>R</th>
<th>W</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-22-3666</td>
<td>Atishoo</td>
<td>48</td>
<td>8</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>231-31-5368</td>
<td>Smiley</td>
<td>22</td>
<td>8</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>131-24-3650</td>
<td>Smethurst</td>
<td>35</td>
<td>5</td>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>434-26-3751</td>
<td>Guldu</td>
<td>35</td>
<td>5</td>
<td>7</td>
<td>32</td>
</tr>
<tr>
<td>612-67-4134</td>
<td>Madayan</td>
<td>35</td>
<td>8</td>
<td>10</td>
<td>40</td>
</tr>
</tbody>
</table>

Hourly_Emps

<table>
<thead>
<tr>
<th>S</th>
<th>N</th>
<th>L</th>
<th>R</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-22-3666</td>
<td>Atishoo</td>
<td>48</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>231-31-5368</td>
<td>Smiley</td>
<td>22</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>131-24-3650</td>
<td>Smethurst</td>
<td>35</td>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>434-26-3751</td>
<td>Guldu</td>
<td>35</td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>612-67-4134</td>
<td>Madayan</td>
<td>35</td>
<td>8</td>
<td>40</td>
</tr>
</tbody>
</table>

Wages

Hourly_Emps2

Q: Are both of these relations now in BCNF?

Decompositions should be used only when needed.

Q: Potential problems of decomposition?

Problems with Decompositions

There are three potential problems to consider:
1) May be impossible to reconstruct the original relation! (Lossiness)
   • Fortunately, not in the SNLWRH example.
2) Dependency checking may require joins.
   • Fortunately, not in the SNLWRH example.
3) Some queries become more expensive.
   • e.g., How much does Guldu earn?

Tradeoff: Must consider these issues vs. redundancy.

Lossless Decomposition (example)

Lossy Decomposition (example)
Lossless Join Decompositions

- Decomposition of R into X and Y is **lossless-join** w.r.t. a set of FDs F if, for every instance r that satisfies F:

  \[ \pi_X(r) \bowtie \pi_Y(r) = r \]

- It is always true that \( r \subseteq \pi_X(r) \bowtie \pi_Y(r) \)

  - In general, the other direction does not hold! If it does, the decomposition is lossless-join.

- Definition extended to decomposition into 3 or more relations in a straightforward way.

- It is essential that all decompositions used to deal with redundancy be lossless! *(Avoids Problem #1)*

More on Lossless Decomposition

- The decomposition of R into X and Y is **lossless with respect to F** if and only if the closure of F contains:

  \[ X \cap Y \rightarrow X, \text{ or } X \cap Y \rightarrow Y \]

  - in example: decomposing ABC into AB and BC is lossy, because intersection (i.e., “B”) is not a key of either resulting relation.

- **Useful result:** If W \( \rightarrow Z \) holds over R and W \( \cap Z \) is empty, then decomposition of R into R-Z and WZ is loss-less.

Lossless Decomposition (example)

\[
\begin{array}{ccc}
A & B & C \\
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 2 & 8 \\
\end{array}
\quad \rightarrow \quad
\begin{array}{ccc}
A & C \\
1 & 3 \\
4 & 6 \\
7 & 8 \\
\end{array}
\quad \bowtie \\
\begin{array}{ccc}
B & C \\
2 & 3 \\
5 & 6 \\
2 & 8 \\
\end{array}
\]

A \( \rightarrow \) B; C \( \rightarrow \) B

But, now we can’t check A \( \rightarrow \) B without doing a join!

Dependency Preserving Decomposition

- **Dependency preserving decomposition** (Intuitive):

  - If R is decomposed into X, Y and Z, and we enforce the FDs that hold individually on X, Y and Z, then all FDs that were given to hold on R should also hold. *(Avoids Problem #2 on our list.)*

- **Projection of set of FDs F**: If R is decomposed into X and Y the projection of F on X (denoted \( F_X \)) is the set of FDs U \( \rightarrow \) V in \( F^* \) (closure of \( F \), not just \( F \) ) such that all of the attributes U, V are in X. *(same holds for Y of course)*

Dependency Preserving Decompositions (Cont.)

- **Definition:** Decomposition of R into X and Y is **dependency preserving** if \( F_X \cup F_Y = F^* \)

  - i.e., if we consider only dependencies in the closure \( F^* \) that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in \( F^* \).

  - **Important to consider \( F^* \) in this definition:**

    - ABC, A \( \rightarrow \) B, B \( \rightarrow \) C, C \( \rightarrow \) A, decomposed into AB and BC.
    - Is this dependency preserving? Is C \( \rightarrow \) A preserved????

      * note: \( F^* \) contains \( F \cup (A \rightarrow C, B \rightarrow A, C \rightarrow B) \), so...

    - FAB contains A \( \rightarrow \) B and B \( \rightarrow \) A? FBC contains B \( \rightarrow \) C and C \( \rightarrow \) B

    - So, \( (FAB \cup FBC)^* \) contains C \( \rightarrow \) A

    - Definition: Decomposition of R into X and Y is **dependency preserving** if \( F_X \cup F_Y = F^* \)

      - i.e., if we consider only dependencies in the closure \( F^* \) that can be checked in \( F^* \) without considering X, and in Y without considering Y, these imply all dependencies in \( F^* \).

      - **Important to consider \( F^* \) in this definition:**

        - ABC, A \( \rightarrow \) B, B \( \rightarrow \) C, C \( \rightarrow \) A, decomposed into AB and BC.
        - Is this dependency preserving? Is C \( \rightarrow \) A preserved????

          * note: \( F^* \) contains \( F \cup (A \rightarrow C, B \rightarrow A, C \rightarrow B) \), so...

          - FAB contains A \( \rightarrow \) B and B \( \rightarrow \) A? FBC contains B \( \rightarrow \) C and C \( \rightarrow \) B

          - So, \( (FAB \cup FBC)^* \) contains C \( \rightarrow \) A

Decomposition into BCNF

- Consider relation R with FDs F. If X \( \rightarrow \) Y violates BCNF, decompose R into X, Y and XY (guaranteed to be loss-less).

  - Repeated application of this idea will give us a collection of relations that are in BCNF: lossless join decomposition, and guaranteed to terminate.

  - e.g., CSDPVQ, key C, \( JP \rightarrow C, SD \rightarrow P, J \rightarrow S \)

    - (contracted; supplier, project, depot, part, qty, value)

    - To deal with SD \( \rightarrow \) P, decompose into SDP, CSDPVQ.

    - To deal with J \( \rightarrow \) S, decompose CSDPVQ into JS and CJDQV

    - So we end up with: SDP, JS, and CJDQV

- **Note:** several dependencies may cause violation of BCNF. The order in which we “deal with” them could lead to very different sets of relations!
BCNF and Dependency Preservation

- In general, there may not be a dependency preserving decomposition into BCNF.
  - e.g., CSZ, CS \rightarrow Z, Z \rightarrow C
  - Can’t decompose while preserving 1st FD; not in BCNF.
- Similarly, decomposition of CSJDPQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs JP \rightarrow P and J \rightarrow S).
  - \{contractid, supplierid, projectid, deptid, partid, qty, value\}
  
- However, it is a lossless join decomposition.
- In this case, adding JPC to the collection of relations gives us a dependency preserving decomposition.
  - but JPC tuples are stored only for checking the f.d. (Redundancy?)

What Does 3NF Achieve?

- If 3NF violated by X \rightarrow A, one of the following holds:
  - X is a subset of some key K ("partial dependency")
    - We store (X, A) pairs redundantly.
    - e.g., Reserves SBD (C is for credit card) with key SBD and S \rightarrow C
  - X is not a proper subset of any key. ("transitive dep.")
    - There is a chain of FDs K \rightarrow X \rightarrow A, which means that we cannot associate an X value with a K value unless we also associate an A value with an X value (different K's, same X implies same A!).
    - problem with initial SNLBBNH example.
  - But: even if R is in 3NF, these problems could arise.
    - e.g., Reserves SBD (note: "C" is for credit card here), S \rightarrow C, C \rightarrow S is in 3NF (why?), but for each reservation of sailor S, same (S, C) pair is stored.
  - Thus, 3NF is indeed a compromise relative to BCNF.

Decomposition into 3NF

- Obviously, the algorithm for lossless join decompose BCNF can be used to obtain a lossless join decompose into 3NF (typically, can stop earlier) but does not ensure dependency preservation.

- To ensure dependency preservation, one idea:
  - If X \rightarrow Y is not preserved, add relation XY.
  - Problem is that XY may violate 3NF! e.g., consider the addition of CJP to ‘preserve’ JP \rightarrow C. What if we also have J \rightarrow C?
- Refinement: Instead of the given set of FDs F, use a minimal cover for F.

Minimal Cover for a Set of FDs

- Minimal cover G for a set of FDs F:
  - Closure of F = closure of G.
  - Right hand side of each FD in G is a single attribute.
  - If we modify G by deleting an FD or by deleting attributes from an FD in G, the closure changes.

- Intuitively, every FD in G is needed, and `as small as possible’ in order to get the same closure as F.
  - e.g., A \rightarrow B, ABCD \rightarrow E, EF \rightarrow GH, ACD \rightarrow EG has the following minimal cover:
    - A \rightarrow B, ACD \rightarrow E, EF \rightarrow G and EF \rightarrow H
  - M.C. implies Lossless-Join, Dep. Pres. Decomp!!!
  - (in book)

Summary of Schema Refinement

- BCNF: each field contains information that cannot be inferred using only FDs.
  - ensuring BCNF is a good heuristic.
- Not in BCNF? Try decomposing into BCNF relations.
  - Must consider whether all FDs are preserved!
- Lossless-join, dependency preserving decompose into BCNF impossible? Consider 3NF.
  - Same if BCNF decom is unsuitable for typical queries
  - Decompositions should be carried out and/or re-examined while keeping performance requirements in mind.
- Note: even more restrictive Normal Forms exist (we don’t cover them in this course, but some are in the book.)