Database Systems
Relational Model (most popular DB model):

- Store information in tables
- Each table is a relation
- Each column is named with an attribute
- Each row is a tuple
- Example relation named Accounts:

<table>
<thead>
<tr>
<th>accountNo</th>
<th>balance</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>1000.0</td>
<td>checking</td>
</tr>
<tr>
<td>67890</td>
<td>123456.00</td>
<td>savings</td>
</tr>
</tbody>
</table>
Structured Query Language (SQL) Preview

Accounts

<table>
<thead>
<tr>
<th>accountNo</th>
<th>balance</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>1000.00</td>
<td>checking</td>
</tr>
<tr>
<td>67890</td>
<td>123456.00</td>
<td>savings</td>
</tr>
</tbody>
</table>

SELECT balance
FROM Accounts
WHERE accountNo = 67890;

SELECT accountNo
FROM Accounts
WHERE type = 'savings'
AND balance < 1000;
Chapter 2:
The Relational Model of Data
Define schema using SQL

Example: Movies ( title, year, length, genre, studioName, ProducerC# )

```
CREATE TABLE Movies (  
title CHAR(100),  
year INT,  
length INT,  
genre CHAR(10),  
studioName CHAR(30),  
producerC# INT  
);
```
How to modify schema

- We can delete a relation $R$ by the SQL statement:
  \[
  \text{DROP TABLE } R;
  \]

- We can modify the schema of an existing relation by a statement that begins with the keywords \texttt{ALTER TABLE} and the name of the relation. The options include:
  - \texttt{ADD} followed by an attribute name and its data type;
    
    Example:
    \[
    \text{ALTER TABLE MovieStar ADD phone CHAR(16);}
    \]
  - \texttt{DROP} followed by an attribute name.
    
    Example:
    \[
    \text{ALTER TABLE MovieStar DROP birthdate;}
    \]
Every table needs a key
How to declare keys

CREATE TABLE MovieStar (  
    name CHAR(30) PRIMARY KEY,  
    address VARCHAR(255),  
    gender CHAR(1),  
    birthdate DATE  
)  

CREATE TABLE MovieStar (  
    name CHAR(30),  
    address VARCHAR(255),  
    gender CHAR(1),  
    birthdate DATE,  
    PRIMARY KEY (name)  
)
Relation $R$:

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>gender</th>
<th>birthdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carrie Fisher</td>
<td>123 Maple St., Hollywood</td>
<td>F</td>
<td>9/9/99</td>
</tr>
<tr>
<td>Mark Hamill</td>
<td>456 Oak Rd., Brentwood</td>
<td>M</td>
<td>8/8/88</td>
</tr>
</tbody>
</table>

Relation $S$:

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>gender</th>
<th>birthdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carrie Fisher</td>
<td>123 Maple St., Hollywood</td>
<td>F</td>
<td>9/9/99</td>
</tr>
<tr>
<td>Harrison Ford</td>
<td>789 Palm Dr., Beverly Hills</td>
<td>M</td>
<td>7/7/77</td>
</tr>
</tbody>
</table>

Relation $R \cap S$:

<table>
<thead>
<tr>
<th>name</th>
<th>address</th>
<th>gender</th>
<th>birthdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carrie Fisher</td>
<td>123 Maple St., Hollywood</td>
<td>F</td>
<td>9/9/99</td>
</tr>
</tbody>
</table>
Projection: pick some columns

The **projection** operator is used to produce from a relation $R$ a new relation that has only some of $R$’s columns.

The value of expression

$$\pi_{A_1,A_2,\ldots,A_n}(R)$$

is a relation that has only the columns for attributes $A_1, A_2, \ldots, A_n$ of $R$.

The schema for the resulting value is the set of attributes $\{A_1, A_2, \ldots, A_n\}$, which we conventionally show in the ordered list.
Selection: pick some rows

The selection operator, applied to a relation $R$, produces a new relation with a subset of $R$’s tuples.

The tuples in the resulting relation are those that satisfy some condition $C$ that involves the attributes of $R$. We denote this operation by

$$\sigma_C(R).$$
The **Cartesian product** (or cross-product, or just product) of two relations \( R \) and \( S \) is the set of pairs that can be formed by choosing the first element of the pair to be any tuple of \( R \) and the second element to be any tuple of \( S \).

This product is denoted by

\[
R \times S.
\]

The relation schema for the resulting relation is the union of the schemas for \( R \) and \( S \). However, if \( R \) and \( S \) should happen to have some attributes in common, then we need to invent new names for at least one of each pair of identical attributes.

To disambiguate an attribute \( A \) that is in the schemas of both \( R \) and \( S \), we use \( R.A \) for the attribute from \( R \) and \( S.A \) for the attribute from \( S \).
Natural Join:

Combine two tuples of two relations only if the two tuples have the same values for common attributes
Theta Join:

Combine two tuples of two relations only if the two tuples satisfy some specified conditions.
Rename: change names of attributes or relation
Overview of relational algebra operations

The operations of the traditional relational algebra fall into four broad classes:

- The usual **set operations** – union, intersection, and difference – applied to relations.
- Operations that remove parts of a relation:
  - **selection** eliminates some rows (tuples).
  - **projection** eliminates some columns.
- Operations that combine the tuples of two relations.
  - **Cartesian product**, which pairs the tuples of two relations in all possible ways.
  - Various kinds of **join** operations, which selectively pair tuples from two relations.
- An operation called **renaming** that does not affect the tuples of a relation, but changes the relation schema, i.e., the names of the attributes and/or the name of the relation itself.
Combine operations

\[ M(H^0) = \frac{\pi}{137} \frac{8\sqrt{hc}}{G} \]

\[ 3987^{12} + 4365^{12} = 4472^{12} \]

\[ \Omega(t) > 1 \]

\[ \text{O} \rightarrow \text{O} + \text{C} \rightarrow \text{O} \]
Constraints for relations:

Another important part of data model
Referential Integrity Constraint:

The values in one column of a relation must also appear in a column of a different relation.

Example

In our movies database, if we see a $\text{StarsIn}$ tuple (which shows which star is in which movie) that has person $p$ in the “$\text{starName}$” component, we would expect that $p$ appears as the name of some star in the $\text{MovieStar}$ relation. (If not, then we would question whether the listed “star” really was a star.)
Chapter 3. Design Theory for Relational Databases
Better description:

George has black hair.  
Pipi has red hair.

George likes apples.  
George likes bananas.  
George likes pie.  
George likes oranges.  
George likes strawberries.

Pipi likes bananas.  
Pipi likes cherries.  
Pipi likes pie.  
Pipi likes pineapples.  
Pipi likes oranges.  
Pipi likes strawberries.
Better relations

George has black hair.
Pipi has red hair.

George likes apples.
George likes bananas.
George likes pie.
George likes oranges.
George likes strawberries.

Pipi likes bananas.
Pipi likes cherries.
Pipi likes pie.
Pipi likes pineapples.
Pipi likes oranges.
Pipi likes strawberries.
## Better relations

<table>
<thead>
<tr>
<th>Name</th>
<th>Hair color</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>black</td>
</tr>
<tr>
<td>Pipi</td>
<td>red</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Hair color</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>apple</td>
</tr>
<tr>
<td>George</td>
<td>banana</td>
</tr>
<tr>
<td>George</td>
<td>pie</td>
</tr>
<tr>
<td>George</td>
<td>orange</td>
</tr>
<tr>
<td>George</td>
<td>strawberry</td>
</tr>
<tr>
<td>Pipi</td>
<td>banana</td>
</tr>
<tr>
<td>Pipi</td>
<td>cherry</td>
</tr>
<tr>
<td>Pipi</td>
<td>pie</td>
</tr>
<tr>
<td>Pipi</td>
<td>pineapple</td>
</tr>
<tr>
<td>Pipi</td>
<td>orange</td>
</tr>
<tr>
<td>Pipi</td>
<td>strawberry</td>
</tr>
</tbody>
</table>
Functional Dependency
<table>
<thead>
<tr>
<th>Name</th>
<th>Hair color</th>
<th>Likes food</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>black</td>
<td>apple</td>
</tr>
<tr>
<td>George</td>
<td>black</td>
<td>banana</td>
</tr>
<tr>
<td>George</td>
<td>black</td>
<td>pie</td>
</tr>
<tr>
<td>George</td>
<td>black</td>
<td>orange</td>
</tr>
<tr>
<td>George</td>
<td>black</td>
<td>strawberry</td>
</tr>
<tr>
<td>Pipi</td>
<td>red</td>
<td>banana</td>
</tr>
<tr>
<td>Pipi</td>
<td>red</td>
<td>cherry</td>
</tr>
<tr>
<td>Pipi</td>
<td>red</td>
<td>pie</td>
</tr>
<tr>
<td>Pipi</td>
<td>red</td>
<td>pineapple</td>
</tr>
<tr>
<td>Pipi</td>
<td>red</td>
<td>orange</td>
</tr>
<tr>
<td>Pipi</td>
<td>red</td>
<td>strawberry</td>
</tr>
</tbody>
</table>

Name → Hair color
Redefine “Key”
Redefine “Key”

We say a set of one or more attributes \( \{A_1, A_2, \cdots, A_n\} \) is a key for a relation \( R \) if:

1. Those attributes functionally determine all other attributes of the relation. That is, it is impossible for two distinct tuples of \( R \) to agree on all of \( A_1, A_2, \cdots, A_n \).

2. No proper subset of \( \{A_1, A_2, \cdots, A_n\} \) functionally determines all other attributes of \( R \); i.e., a key must be minimal.
Boyce-Codd Normal Form (BCNF)

Usually we want every relation to be in BCNF.
BCNF (Boyce-Codd Normal Form)

**Definition (Boyce-Codd normal form (BCNF))**

A relation $R$ is in BCNF if and only if: whenever there is a nontrivial FD

$$A_1 A_2 \cdots A_n \rightarrow B_1 B_2 \cdots B_m$$

for $R$, it is the case that $\{A_1, A_2, \ldots, A_n\}$ is a superkey for $R$.

That is, the left side of every nontrivial FD must be a superkey.

Recall that a superkey need not be minimal.

Thus, an equivalent statement of the BCNF condition is that the left side of every nontrivial FD must contain a key.
How to get relations in BCNF?
Closure (of a set of attributes)

Given FDs for a relation R, the closure of a set of attributes \{A_1, A_2, \ldots, A_n\} are all the attributes of R that \{A_1, A_2,\ldots, A_n\} can functionally determine together.

Example: If A → B and B → C, then the closure of \{A\} is \{A,B,C\}, because A → A,B,C.

We denote the closure of a set of attributes $A_1A_2\cdots A_n$ by

$$\{A_1, A_2, \ldots, A_n\}^+.$$
How to find Closure?

In a greedy way.
Example: If $A \rightarrow B$ and $B \rightarrow C$, then the closure of $\{A\}$ is $\{A,B,C\}$

The greedy way:

$B \rightarrow C$

Closure of $A$
Computing The Closure of Attributes

Algorithm: Closure of a Set of Attributes.

INPUT: A set of attributes \( \{A_1, A_2, \ldots, A_n\} \) and a set of FDs \( S \).

OUTPUT: The closure \( \{A_1, A_2, \ldots, A_n\}^{+} \).

1. If necessary, split the FDs of \( S \), so each FD in \( S \) has a single attribute on the right.

2. Let \( X \) be a set of attributes that eventually will become the closure. Initialize \( X \) to be \( \{A_1, A_2, \ldots, A_n\} \).

3. Repeatedly search for some FD

\[
B_1 B_2 \cdots B_m \rightarrow C
\]

such that all of \( B_1, B_2, \ldots, B_m \) are in the set of attributes \( X \), but \( C \) is not. Add \( C \) to the set \( X \) and repeat the search. (Since \( X \) can only grow, this step will eventually end.)

4. The set \( X \), after no more attribute can be added to it, is the correct value of \( \{A_1, A_2, \ldots, A_n\}^{+} \).
How to turn the “bad relation” into good relations?

<table>
<thead>
<tr>
<th>UIN</th>
<th>Name</th>
<th>Course</th>
</tr>
</thead>
</table>

FD: UIN → Name

\{UIN\}^+ = \{UIN, Name\}

Good relations

<table>
<thead>
<tr>
<th>UIN</th>
<th>Name</th>
</tr>
</thead>
</table>

Closure of UIN

<table>
<thead>
<tr>
<th>UIN</th>
<th>Course</th>
</tr>
</thead>
</table>

UIN and attributes outside the closure of UIN
Boyce-Codd Normal Form (BCNF)

Usually we want every relation to be in BCNF.
BCNF (Boyce-Codd Normal Form)

Definition (Boyce-Codd normal form (BCNF))

A relation \( R \) is in BCNF if and only if: whenever there is a nontrivial FD

\[
A_1A_2\cdots A_n \rightarrow B_1B_2\cdots B_m
\]

for \( R \), it is the case that \( \{A_1, A_2, \ldots, A_n\} \) is a superkey for \( R \).

That is, the left side of every nontrivial FD must be a superkey.

Recall that a superkey need not be minimal.

Thus, an equivalent statement of the BCNF condition is that the left side of every nontrivial FD must contain a key.
What FDs hold for each of the two smaller relations?

We need to know this, because the small relations may not be in BCNF. If that’s true, we need to decompose them as well.
Method:

1. Find out all FDs of the previous big relation
2. For each small relation, see which FDs apply to it

How to find out all FDs of a relation?
By computing the closure of every subset of attributes.

Example: if we find \{A,B\}+ = \{A,B,C,D\}, the we have:

A,B \rightarrow C and A,B \rightarrow D
Consider the relation with schema

\{ title, year, studioName, president, presAddr \}.

Three FD’s that we would assume in this relation are:

\[ \text{title} \rightarrow \text{studioName} \]

\[ \text{studioName} \rightarrow \text{president} \]

\[ \text{president} \rightarrow \text{presAddr} \]

We discover that \{title, year\} is the only key. Thus the last two FD’s above violate BCNF.

Suppose we use studioName \rightarrow \text{president} for the decomposition. First, we get

\{studioName\}^+ = \{studioName, president, presAddr\}.
So we decompose the relation into these two smaller schemas:

1. \{title, year, studioName\}.
2. \{studioName, president, presAddr\}.

We project the FD’s to the two smaller relations, and find that:

1. The schema \{title, year, studioName\} has the FD

   \[\text{title} \ \text{year} \rightarrow \text{studioName}.\]

2. The schema \{studioName, president, presAddr\} has two FD’s

   \[\text{studioName} \rightarrow \text{president}\]
   \[\text{president} \rightarrow \text{presAddr}.\]

The first relation is in BCNF. But the second relation is not, because \text{president} \rightarrow \text{presAddr} is a BCNF violation.
Decomposition into BCNF

Example

So we decompose the relation \(\{\text{studioName}, \text{president}, \text{presAddr}\}\) based on the BCNF violation \(\text{president} \rightarrow \text{presAddr}\). And we get two (even smaller) relation schemas:

1. \(\{\text{studioName}, \text{president}\}\).
2. \(\{\text{president}, \text{presAddr}\}\).

So we have decomposed the original relation into three smaller relations in BCNF:

1. \(\{\text{title}, \text{year}, \text{studioName}\}\).
2. \(\{\text{studioName}, \text{president}\}\).
3. \(\{\text{president}, \text{presAddr}\}\).

Formal description of the decomposition algorithm: See Algorithm 3.20 on page 92 of the textbook.
See redundancy?

Pipi has a horse, and likes bananas.
Pipi has a horse, and likes cherries.
Pipi has a horse, and likes pineapples.
Pipi has a horse, and likes strawberries.
Pipi has a monkey, and likes bananas.
Pipi has a monkey, and likes cherries.
Pipi has a monkey, and likes pineapples.
Pipi has a monkey, and likes strawberries.
Pipi has a dog, and likes bananas.
Pipi has a dog, and likes cherries.
Pipi has a dog, and likes pineapples.
Pipi has a dog, and likes strawberries.
Pipi has a cat, and likes bananas.
Pipi has a cat, and likes cherries.
Pipi has a cat, and likes pineapples.
Pipi has a cat, and likes strawberries.
Better description:

Pipi has a horse.
Pipi has a monkey.
Pipi has a dog.
Pipi has a cat.

Pipi likes bananas.
Pipi likes cherries.
Pipi likes pineapples.
Pipi likes strawberries.
Better relations

<table>
<thead>
<tr>
<th>Name</th>
<th>Pet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipi</td>
<td>horse</td>
</tr>
<tr>
<td>Pipi</td>
<td>monkey</td>
</tr>
<tr>
<td>Pipi</td>
<td>dog</td>
</tr>
<tr>
<td>Pipi</td>
<td>cat</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Likes fruit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipi</td>
<td>banana</td>
</tr>
<tr>
<td>Pipi</td>
<td>cherry</td>
</tr>
<tr>
<td>Pipi</td>
<td>pineapple</td>
</tr>
<tr>
<td>Pipi</td>
<td>strawberry</td>
</tr>
</tbody>
</table>
Then how to remove the redundancy?
Multi-Valued Dependency (MVD)
A **multivalued dependency (MVD)** is a statement about some relation \( R \) that

- When you fix the values for one set of attributes, then the values in certain other attributes are independent of the values of all the other attributes in the relation.
More precisely, we say the MVD

\[ A_1 A_2 \cdots A_n \rightarrow\rightarrow B_1 B_2 \cdots B_m \]

holds for a relation \( R \) if when we restrict ourselves to the tuples of \( R \) that have particular values for each of the attributes among the \( A \)'s, then the set of values we find among the \( B \)'s is independent of the set of values we find among the attributes of \( R \) that are not among the \( A \)'s or \( B \)'s.

That is, when \( A \) is given, \( B \) and \( C \) become independent. Here \( A, B, C \) can all be sets of attributes.
Example: `name ➔ pet` or equivalently, `name ➔ likes fruit`

<table>
<thead>
<tr>
<th>Name</th>
<th>Pet</th>
<th>Likes fruit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipi</td>
<td>horse</td>
<td>banana</td>
</tr>
<tr>
<td>Pipi</td>
<td>horse</td>
<td>cherry</td>
</tr>
<tr>
<td>Pipi</td>
<td>horse</td>
<td>pineapple</td>
</tr>
<tr>
<td>Pipi</td>
<td>horse</td>
<td>strawberry</td>
</tr>
<tr>
<td>Pipi</td>
<td>monkey</td>
<td>banana</td>
</tr>
<tr>
<td>Pipi</td>
<td>monkey</td>
<td>cherry</td>
</tr>
<tr>
<td>Pipi</td>
<td>monkey</td>
<td>pineapple</td>
</tr>
<tr>
<td>Pipi</td>
<td>monkey</td>
<td>strawberry</td>
</tr>
<tr>
<td>Pipi</td>
<td>dog</td>
<td>banana</td>
</tr>
<tr>
<td>Pipi</td>
<td>dog</td>
<td>cherry</td>
</tr>
<tr>
<td>Pipi</td>
<td>dog</td>
<td>pineapple</td>
</tr>
<tr>
<td>Pipi</td>
<td>dog</td>
<td>strawberry</td>
</tr>
<tr>
<td>Pipi</td>
<td>cat</td>
<td>banana</td>
</tr>
<tr>
<td>Pipi</td>
<td>cat</td>
<td>cherry</td>
</tr>
<tr>
<td>Pipi</td>
<td>cat</td>
<td>pineapple</td>
</tr>
<tr>
<td>Pipi</td>
<td>cat</td>
<td>strawberry</td>
</tr>
</tbody>
</table>
Yes, FD is a special case of MVD.

Name $\rightarrow$ Pet
or
Name $\rightarrow\rightarrow$ Pet
?

Both are true.
MVD is NOT a special case of FD.

Can we say Name $\Rightarrow$ Likes fruit?
No.
Fourth Normal Form (4-NF)
The fourth normal form condition is essentially the BCNF condition, but applied to MVD’s instead of FD’s. Formally:

- A relation $R$ is in **fourth normal form (4NF)** if whenever
  \[ A_1 A_2 \cdots A_n \rightarrow\rightarrow B_1 B_2 \cdots B_m \]

  is a nontrivial MVD,

  \[ \{A_1, A_2, \cdots, A_n\} \]

  is a superkey.

Note that the notions of keys and superkeys depend on FD’s only; adding MVD’s does not change the definition of “keys.”
Then how to remove the redundancy?

Or, how to decompose a relation into relations in 4-NF?
Answer:
Whenever you see a relation \((A,B,C)\) with MVD \(A \rightarrow \rightarrow B\) where \(A\) is not a superkey, decompose it into \((A,B)\) and \((A,C)\).

Note: \(A, B\) and \(C\) can all be sets of attributes.
Chapter 4. High-Level Database Models
Steps in database design:

1. Show structure of data using a diagram

2. Turn the diagram into relations
Diagram for data:

People

- name
- hair color
- fly
In life, entities are related to each other. So there are relationships.
Schemas:

- **People**
  - name
  - hair color
  - fly

- **Foods**
  - food name
  - calories

- **Likes**
  - name
  - food name

Entity set: People

Relationship: Likes

Entity set: Foods
The Entity/Relationship Model

In the entity-relationship model (or E/R model), the structure of data is represented graphically, as an “entity-relationship diagram,” using three principle element types:

1. Entity sets,
2. Attributes, and
Attribute: oval
Relationship: diamond
Multiway Relationships

A multiway relationship is a relationship involving more than two entity sets. It is represented by lines from the relationship diamond to each of the involved entity sets.
Example

Here is a three-way relationship:

- Stars
- Contracts
- Movies
- Studios
Sometimes it is convenient, or even essential, to associate attributes with a relationship.
Subclasses in the E/R Model

Often, an entity set contains certain entities that have special properties not associated with all members of the set. If so, we find it useful to define certain special-case entity sets, or subclasses, each with its own special attributes and/or relationships. We connect an entity set to its subclasses using a relationship called isa.

Example: “chocolate” is a special kind of “candy”
Constraints
Representing Keys in the E/R Model

In the E/R diagram notation, we underline the attributes belonging to a key for an entity set.

Example

```
Stars
  name
  address

Stars-in

Movies
  title
  year

Owns

Studios
  name
  address
```
A president is the president of exactly one country. A country has at most one president.
In the E/R model, we can attach a bounding number to the edges that connect a relationship to an entity set, indicating limits on the number of entities that can be connected to any one entity of the related entity set.

Example

We choose to place a constraint on the degree of a relationship, such as that a movie entity cannot be connected by relation **Stars-in** to more than 10 star entities.
Weak Entity Set

Police: Who are you?

Dog: I am Bart Simpson’s dog.
UIN

attends

name

underline key attributes
How to turn E/R diagram into relations?
Schemas:

People (name, hair color, fly)

Foods (food name, calories)

Likes (name, food name)
Chapter 5. Algebraic and Logical Query Languages
Suppose that $R$ and $S$ are bags, and that tuple $t$ appears $n$ times in $R$ and $m$ times in $S$. Then:

- In the bag union $R \cup S$, tuple $t$ appears $n + m$ times.
- In the bag intersection $R \cap S$, tuple $t$ appears $\min(n, m)$ times.
- In the bag difference $R - S$, tuple $t$ appears $\max(0, n - m)$ times.
Extended Operations of Relational Algebra

So far we have learned

- The classical relational algebra for sets.
- The modifications necessary to treat relations as bags.

However, languages such as SQL have several other operations that have proved quite important in applications. These **extended operations** include:

1. **Duplicate elimination** operator $\delta$.
2. **Aggregation** operators.
3. **Grouping** of tuples according to their value in one or more attributes.
4. **Extended projection** gives additional power to the operator $\pi$.
5. **Sorting** operator $\tau$ turns a relation into a list of tuples, sorted according to one or more attributes.
6. **Outerjoin** operator is a variant of the join that avoids losing dangling tuples.
Chapter 6. The Database Language SQL
SELECT desired attributes
FROM one or more tables
WHERE condition about tuples of the tables
Three-Valued Logic

To understand how AND, OR and NOT work in 3-valued logic, think of

- TRUE = 1.
- FALSE = 0.
- UNKNOWN = \( \frac{1}{2} \).
- AND = min.
- OR = max.
- NOT(\(x\)) = 1 - \(x\).

Example

\[
\text{TRUE AND (FALSE OR NOT(UNKNOWN))} \\
= \min(1, \max(0, (1 - \frac{1}{2}))) \\
= \min(1, \max(0, \frac{1}{2})) \\
= \min(1, \frac{1}{2}) \\
= \frac{1}{2} \\
= \text{UNKNOWN}
\]
The **ORDER BY** clause follows the **WHERE** clause and any other clauses (which we will learn later). The ordering is performed on the result of the FROM, WHERE and other clauses, just before we apply the SELECT clause. The tuples of the result are then sorted by the attributes in the list of the ORDER BY clause, and then passed to the SELECT clause for processing in the normal manner.

**Example**

```sql
SELECT * 
FROM Movies 
WHERE studioName = 'Disney' AND year = 1990 
ORDER BY length, title;
```
Subqueries

- A parenthesized SELECT-FROM-WHERE statement (*subquery*) can be used as a value in a number of places, including FROM and WHERE clauses.

- Example: in place of a relation in the FROM clause, we can place another query, and then query its result.
  - Can use a tuple-variable to name tuples of the result.
The IN Operator

- `<tuple> IN <relation>` is true if and only if the tuple is a member of the relation.
  - `<tuple> NOT IN <relation>` means the opposite.
- IN-expressions can appear in WHERE clauses.
- The `<relation>` is often a subquery.
The Exists Operator

- EXISTS( <relation> ) is true if and only if the <relation> is not empty.
- Example: From Candies(name, manf), find those candies that are the unique candy by their manufacturer.
The Operator ANY

- $x = \text{ANY}( <\text{relation}> )$ is a boolean condition that is true if $x$ equals at least one tuple in the relation.
- Similarly, $=$ can be replaced by any of the comparison operators.
- Example: $x > \text{ANY}( <\text{relation}> )$ means $x$ is not the smallest tuple in the relation.
  - Note tuples must have one component only.
Similarly, \( x <> \text{ALL( <relation> )} \) is true if and only if for every tuple \( t \) in the relation, \( x \) is not equal to \( t \).

- That is, \( x \) is not a member of the relation.

The \( <> \) can be replaced by any comparison operator.

Example: \( x >= \text{ALL( <relation> )} \) means there is no tuple larger than \( x \) in the relation.
Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:

- \(( \text{subquery} ) \ \text{UNION} \ ( \text{subquery} )\)
- \(( \text{subquery} ) \ \text{INTERSECT} \ ( \text{subquery} )\)
- \(( \text{subquery} ) \ \text{EXCEPT} \ ( \text{subquery} )\)
Controlling Duplicate Elimination

- Force the result to be a set by `SELECT DISTINCT . . .`
- Force the result to be a bag (i.e., don’t eliminate duplicates) by `ALL`, as in `. . . UNION ALL . . .`
Natural join:
R NATURAL JOIN S;

Product:
R CROSS JOIN S;

Example:
Likes NATURAL JOIN Sells;

Relations can be parenthesized subqueries, as well.
Theta Join

- $R \ JOIN \ S \ ON <\text{condition}>$
- Example: using $\text{Consumers}(\text{name, addr})$ and $\text{Frequents}(\text{consumer, store})$:

  $\text{Consumers JOIN Frequents ON name = consumer;}$

  gives us all $(c, a, c, s)$ quadruples such that consumer $c$ lives at address $a$ and frequents store $s$. 
Outerjoins

- R OUTER JOIN S is the core of an outerjoin expression. It is modified by:
  1. Optional NATURAL in front of OUTER.
  2. Optional ON <condition> after JOIN.
  3. Optional LEFT, RIGHT, or FULL before OUTER.
     - LEFT = pad dangling tuples of R only.
     - RIGHT = pad dangling tuples of S only.
     - FULL = pad both; this choice is the default.
Aggregations

- SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause to produce that aggregation on the column.
- Also, COUNT(*) counts the number of tuples.
Grouping

- We may follow a SELECT-FROM-WHERE expression by GROUP BY and a list of attributes.

- The relation that results from the SELECT-FROM-WHERE is grouped according to the values of all those attributes, and any aggregation is applied only within each group.
Example: Grouping

- From `Sells(store, candy, price)`, find the average price for each candy:

```
SELECT candy, AVG(price)
FROM Sells
GROUP BY candy;
```
HAVING Clauses

- HAVING <condition> may follow a GROUP BY clause.
- If so, the condition applies to each group, and groups not satisfying the condition are eliminated.
More on SQL

Database Modification
Defining a Database Schema
Views
A *modification* command does not return a result (as a query does), but changes the database in some way.

Three kinds of modifications:

1. *Insert* a tuple or tuples.
2. *Delete* a tuple or tuples.
3. *Update* the value(s) of an existing tuple or tuples.
To insert a single tuple:

\[
\text{INSERT INTO } \langle \text{relation} \rangle \\
\text{VALUES ( } \langle \text{list of values} \rangle \text{ );}
\]

**Example:** add to \text{Likes}(\text{consumer, candy}) the fact that Sally likes Twizzlers.

\[
\text{INSERT INTO Likes} \\
\text{VALUES('Sally', 'Twizzler');}
\]
Deletion

- To delete tuples satisfying a condition from some relation:
  
  ```sql
  DELETE FROM <relation>
  WHERE <condition>;
  ```
Updates

- To change certain attributes in certain tuples of a relation:

```
UPDATE <relation>
SET <list of attribute assignments>
WHERE <condition on tuples>;
```
Defining a Database Schema

- A *database schema* comprises declarations for the relations (“tables”) of the database.
- Several other kinds of elements also may appear in the database schema, including views, indexes, and triggers, which we’ll introduce later.
Creating (Declaring) a Relation

- Simplest form is:
  ```sql
  CREATE TABLE <name> (  
  <list of elements>  
  );
  ```

- To delete a relation:
  ```sql
  DROP TABLE <name>;
  ```
Declaring Single-Attribute Keys

- Place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute.
- Example:

```sql
CREATE TABLE Candies (  
  name    CHAR(20) UNIQUE,  
  manf    CHAR(20)  
);  
```
Declaring Multiattribute Keys

- A key declaration can also be another element in the list of elements of a CREATE TABLE statement.
- This form is essential if the key consists of more than one attribute.
  - May be used even for one-attribute keys.
Some Other Declarations for Attributes

- NOT NULL means that the value for this attribute may never be NULL.
- DEFAULT <value> says that if there is no specific value known for this attribute’s component in some tuple, use the stated <value>.
Adding Attributes

- We may add a new attribute ("column") to a relation schema by:
  
  \[
  \text{ALTER TABLE <name> ADD}
  \]
  
  \[
  \text{<attribute declaration> ;}
  \]

- Example:

  \[
  \text{ALTER TABLE Stores ADD phone CHAR(16) DEFAULT 'unlisted';}
  \]
Deleting Attributes

- Remove an attribute from a relation schema by:

  ```sql
  ALTER TABLE <name>
  DROP <attribute>;
  ```

- Example: we don’t really need the license attribute for stores:

  ```sql
  ALTER TABLE Stores DROP license;
  ```
A view is a “virtual table” = a relation defined in terms of the contents of other tables and views.

Declare by:
CREATE VIEW <name> AS <query>;

Antonym: a relation whose value is really stored in the database is called a base table.
Chapter 7: Constraints and Triggers

Foreign Keys
Local and Global Constraints
Triggers
Kinds of Constraints

- **Keys.**
- **Foreign-key**, or referential-integrity.
- **Value-based** constraints.
  - Constrain values of a particular attribute.
- **Tuple-based** constraints.
  - Relationship among components.
- **Assertions**: any SQL boolean expression.
Expressing Foreign Keys

- Use the keyword REFERENCES, either:
  - Within the declaration of an attribute (only for one-attribute keys).
  - As an element of the schema:
    FOREIGN KEY ( <list of attributes> )
    REFERENCES <relation> ( <attributes> )

- Referenced attributes must be declared PRIMARY KEY or UNIQUE.
CREATE TABLE Sells ( 
  store CHAR(20),
  candy CHAR(20),
  price REAL,
  FOREIGN KEY(candy)
    REFERENCES Candies(name)
    ON DELETE SET NULL
    ON UPDATE CASCADE
);

Attribute-Based Checks

- Constraints on the value of a particular attribute.
- Add: CHECK( <condition> ) to the declaration for the attribute.
- The condition may use the name of the attribute, but any other relation or attribute name must be in a subquery.
Tuple-Based Checks

- CHECK ( <condition> ) may be added as a relation-schema element.
- The condition may refer to any attribute of the relation.
  - But any other attributes or relations require a subquery.
- Checked on insert or update only.
Assertions

- These are database-schema elements, like relations or views.
- Defined by:
  
  ```
  CREATE ASSERTION <name>
  CHECK ( <condition> );
  ```
- Condition may refer to any relation or attribute in the database schema.
CREATE TRIGGER PriceTrig
AFTER UPDATE OF price ON Sells
REFERENCING
  OLD ROW AS ooo
  NEW ROW AS nnn
FOR EACH ROW
WHEN(nnn.price > ooo.price + 1.00)
INSERT INTO RipoffStores
  VALUES(nnn.store);
Chapter 11

Semistructured Data Model
Relations are structured data

All tuples have the same structure (same set of attributes).
Despite federal safety investigations of Tesla’s self-driving cars, the company’s chief executive, Elon Musk, is hardly backing off on his grand plans for autonomous vehicles.

In a blog post late Wednesday, Mr. Musk updated Tesla’s “master plan” with a pledge to expand beyond electric cars into battery-powered pickups, semitrucks and buses, and to equip them with advanced self-driving systems.

He made no mention of the fatal May 7 accident involving a Tesla Model S with its Autopilot system engaged, or of the federal scrutiny of the technology. A criticism of that system has been that, despite its name, its collision-avoidance abilities depend on the human driver’s being ready to immediately retake control of the vehicle in a crisis.
If we describe data (in a structured way) along with its structure, then it is semi-structured data.
11.1.2 Semistructured Data Representation: Example

```
Root
  └── star
    ├── movie
    │   └── sw
    │       └── year
    │           └── 1977
    │   └── mh
    │       └── city
    │           └── Brentwood
    │   └── street
    │       └── Oak
    │   └── name
    │       └── Mark Hamill
    └── star
        └── name
            └── Carrie Fisher
                └── street
                    └── Maple
                        └── city
                            └── Hollywood
                                └── city
                                    └── Locust
                                        └── street
                                            └── Malibu
```
11.1.2 Semistructured Data Representation: Example

![Diagram showing a tree structure with nodes labeled as Root, cf, mh, and sw, and edges labeled as star, movie, starOf, starsIn, starOf, and starsIn. The nodes contain attributes such as name, address, street, city, and date.]
XML: Extensible Markup Language
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
  <Star>
    <Name>Carrie Fisher</Name>
    <Address>
      <Street>123 Maple St.</Street>
      <City>Hollywood</City>
    </Address>
  </Star>
  <Star>
    <Name>Mark Hamill</Name>
    <Address>
      <Street>456 Oak Rd.</Street>
      <City>Brentwood</City>
    </Address>
  </Star>
  <Movie>
    <Title>Star Wars</Title>
    <Year>1977</Year>
  </Movie>
</StarMovieData>
XML is designed to be used in two somewhat different modes:

1. Well-formed XML.
2. Valid XML.
11.2.5 Attributes That Connect Elements: Example

```xml
<? xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<StarMovieData>
  <Star StarID = "cf" starredIn = "sw">
    <Name>Carrie Fisher</Name>
    <Address>
      <Street>123 Maple St.</Street>
      <City>Hollywood</City>
    </Address>
    <Address>
      <Street>5 Locust Ln.</Street>
      <City>Malibu</City>
    </Address>
  </Star>
  <Star starID = "mh" starredIn = "sw">
    <Name>Mark Hamill</Name>
    <Address>
      <Street>456 Oak Rd.</Street>
      <City>Brentwood</City>
    </Address>
  </Star>
  <Movie movieID = "sw" startsOf = "cf", "mh">
    <Title>Star Wars</Title>
    <Year>1977</Year>
  </Movie>
</StarMovieData>
```
Namespace

Warren Buffett: A security guy?
Suppose that we want to say that in element **StarMovieData** shown previously, certain tags belong to the namespace defined in the document infolab.stanford.edu/movies.

We could choose a name such as **md** for the namespace by using the opening tag:

```
<md:StarMovieData xmlns:md="http://infolab.stanford.edu/movies">
```

Our intent is that StarMovieData itself is part of this namespace, so it gets the prefix **md:** , as does its closing tag

```
</md:StarMovieData>
```

Inside this element, we have the option of asserting that tags of subelements belong to this namespace by prefixing their opening and tags with **md:**.
DTD: Document Type Definition
11.3.1 The Form of a DTD: Example

<!DOCTYPE Stars [
  <!ELEMENT Stars (Star*)>
  <!ELEMENT Star (Name, Address+, Movies)>
  <!ELEMENT Name (#PCDATA)>
  <!ELEMENT Address (Street, City)>
  <!ELEMENT Street (#PCDATA)>
  <!ELEMENT City (#PCDATA)>
  <!ELEMENT Movies (Movie*)>
  <!ELEMENT Movie (Title, Year)>
  <!ELEMENT Title (#PCDATA)>
  <!ELEMENT Year (#PCDATA)>
]>
<!DOCTYPE StarMovieData [ 
  <!ELEMENT StarMovieData (Star*, Movie*)> 
  <!ELEMENT Star (Name, Address+)> 
    <!ATTLIST Star 
      starID ID #REQUIRED 
      starredIn IDREFS #IMPLIED 
    > 
  <!ELEMENT Name (#PCDATA)> 
  <!ELEMENT Address (Street, City)> 
  <!ELEMENT Street (#PCDATA)> 
  <!ELEMENT City (#PCDATA)> 
  <!ELEMENT Movie (Title, Year)> 
    <!ATTLIST Movie 
      movieID ID #REQUIRED 
      starOf IDREFS #IMPLIED 
    > 
  <!ELEMENT Title (#PCDATA)> 
  <!ELEMENT Year (#PCDATA)> ]>
XML Schema
1)  
   
2)  
   
3)  
   
4)  
   
5)  
   
6)  
   
7)  
   
8)  
   
9)  
   
10)  
   
11)  
   
12)  
   
13)  
   
14)  
   
15)  
   
11.4.4 Attributes: Example

1)    <? xml version = "1.0" encoding = "utf-8" ? > 
2)    <xs:schema xmlns:xs = "http : //www.w3.org/2001/XMLSchema" > 

3)    <xs:complexType name = "movieType" > 
4)        <xs:attribute name = "Title" type = "xs:string" use = "required" / > 
5)        <xs:attribute name = "Year" type = "xs:integer" use = "required" / > 
6)    </xs:complexType> 

7)    <xs:element name = "Movies" > 
8)        <xs:complexType> 
9)            <xs:sequence> 
10)       <xs:element name = "Movie" type = "movieType" minOccurs = 0 maxOccurs = "unbounded" / > 
11)   </xs:sequence> 
12)  </xs:complexType> 
13)  </xs:element> 
14)  </xs:schema>
Suppose we want to restrict the year of a movie to be no earlier than 1915. We can define a simple type as follows. The type `movieYearType` can then be used to replace `xs:integer` in our previous XML Schema example.

```xml
<xs:simpleType name = "movieYearType">
  <xs:restriction base = "xs:integer">
    <xs:minInclusive value = "1915"/>
  </xs:restriction>
</xs:simpleType>
```
**Example:** Let us design a simple type suitable for the *genre* of movies. In our running example, we have supposed that there are only four possible genres: comedy, drama, sciFi, and teen. The following example shows how to define a type `genreType` that could serve as the type for an element or attribute representing our genres of movies.

```xml
<xs:simpleType name = "genreType">
  <xs:restriction base = "xs:string">
    <xs:enumeration value = "comedy"/>
    <xs:enumeration value = "drama"/>
    <xs:enumeration value = "sciFi"/>
    <xs:enumeration value = "teen"/>
  </xs:restriction>
</xs:simpleType>
```
11.4.6 Keys in XML Schema: Example

1)    <xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
2)     <xs:simpleType name = "genreType">
3)         <xs:restriction base = "xs:string">
4)             <xs:enumeration value = "comedy" />
5)             <xs:enumeration value = "drama" />
6)             <xs:enumeration value = "sciFi" />
7)             <xs:enumeration value = "teen" />
8)         </xs:restriction>
9)     </xs:simpleType>
10)    </xs:schema>
11)    <xs:complexType name = "movieType">
12)     <xs:sequence>
13)         <xs:element name = "Title" type = "xs:string"/>
14)         <xs:element name = "Year" type = "xs:integer"/>
15)         <xs:element name = "Genre" type = "genreType" minOcccues = "0" maxOccures = "1"/>
16)     </xs:sequence>
17)    </xs:complexType>
18)  <xs:element name = “Movies”>
19)   <xs:complexType>
20)     <xs:sequence>
21)       <xs:element name = “Movie” type = “movieType”
          minOccurs = “0” maxOccurs = “unbounded” />
22)     </xs:sequence>
23)   </xs:complexType>
24)   <xs:key name = “movieKey”>
25)     <xs:selector xpath = “Movie” />
26)     <xs:field xpath = “Title” />
27)     <xs:field xpath = “Year” />
28)   </xs:key>
29)   </xs:element>
30) </xs:schema>
1.  <? xml version = "1.0" encoding = "uft-8"?>
2.  <xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
3.  <xs:element name = "Stars">
4.      <xs:complexType>
5.          <xs:sequence>
6.              <xs:element name = "Star" minOccurs = "1" maxOccures = "unbounded">
7.                  <xs:complexType>
8.                      <xs:sequence>
9.                          <xs:element name = "Name" type = "xs:string" />
10.                         <xs:element name = "Address" type = "xs:string" />
11.                            <xs:element name = "StarredIn" minOccurs = "0" maxOccurs = "unbounded"
11.4.7 Foreign Keys in XML Schema: Example

<xs:complexType>
    <xs:attribute name="title"
        type="xs:string"/>
    <xs:attribute name="year"
        type="xs:integer"/>
</xs:complexType>

<xs:element>
    <xs:sequence>
        <xs:complexType>
            <xs:element>
                <xs:sequence>
                    <xs:complexType>
                        <xs:keyref name="movieRef" refers="movieKey">
                            <xs:selector xpath="Star/StarredIn"/>
                            <xs:field xpath="@title"/>
                            <xs:field xpath="@year"/>
                        </xs:keyref>
                    </xs:complexType>
                </xs:sequence>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
Indexes on Sequential Files
Example of a Dense Index

Dense Index

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td></td>
</tr>
</tbody>
</table>

Sequential File

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

The diagram shows a dense index with values from 10 to 120, connected to a sequential file with the corresponding values.
Sparse Index Example

Sparse Index

<table>
<thead>
<tr>
<th>10</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td>70</td>
<td>40</td>
</tr>
<tr>
<td>90</td>
<td>50</td>
</tr>
<tr>
<td>110</td>
<td>60</td>
</tr>
<tr>
<td>130</td>
<td>70</td>
</tr>
<tr>
<td>150</td>
<td>80</td>
</tr>
<tr>
<td>170</td>
<td>90</td>
</tr>
<tr>
<td>190</td>
<td>100</td>
</tr>
<tr>
<td>210</td>
<td></td>
</tr>
<tr>
<td>230</td>
<td></td>
</tr>
</tbody>
</table>

Sequential File

<table>
<thead>
<tr>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>70</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>
Two-Level Index Example

Sparse 2nd level

| 10 | 10 |
| 90 | 30 |
| 170| 50 |
| 250| 70 |

| 330| 90 |
| 410| 110|
| 490| 130|
| 570| 150|

| 170| 170 |
| 190| 90  |
| 210| 100 |
| 230|     |

Sequential File

| 10 | 10 |
| 20 | 30 |

| 50 | 50 |
| 40 | 60 |

| 70 | 70 |
| 80 |  
| 90 |  
| 100|   |
More on Indexes

Secondary Indexes
B-Trees
Duplicate values & secondary indexes

```
10
20
30
40
50
60...
```

```
20
10
20
40
10
40
10
40
30
40
```

buckets
B-Tree Structure

- an example of a balanced search tree: every root-to-leaf path has same length
- each node (vertex) in the tree is a block, which contains search keys and pointers
- parameter $n$, which is largest value so that $n+1$ pointers and $n$ keys fit in one block
  - Ex: If block size is 4096 bytes, keys are 4 bytes, and pointers are 8 bytes, then $n = 340$. 
Constraints on B-Tree Nodes

- Keys in leaf nodes are copies of keys from data file, in sorted order
- Root contains between 2 and \( n+1 \) index node pointers
- Each internal node contains between \([ (n+1)/2 ]\) and \( n+1 \) index node pointers
- Each non-leaf node consists of \( \text{ptr}_1, \text{key}_1, \text{ptr}_2, \text{key}_2, \ldots, \text{key}_{m-1}, \text{ptr}_m \) where \( \text{ptr}_i \) points to index node with keys between \( \text{key}_{i-1} \) and \( \text{key}_i \)
Constraints (cont'd)

- Each leaf contains between \( \lceil (n+1)/2 \rceil \) and \( n \) data record pointers, plus a "next leaf" pointer.

- Associated with each data record pointer is a key, and the pointer points to the data record with that key.
B-Tree Example

n=3

Root

3 5 11
30 35
100 101 110
120 130
150 156 157 180
180 200
Insert into B+tree

(a) simple case
  • space available in leaf
(b) leaf overflow
(c) non-leaf overflow
(d) new root
(d) New root, insert 45

new root

n=3
Deletion from B-tree

(a) Simple case - no example
(b) Coalesce with neighbor (sibling)
(c) Re-distribute keys
(d) Cases (b) or (c) at non-leaf
(d) Non-leaf coalese

- Delete 37
Range Queries with B-Trees

- **Range query**: a query in which a range of values is sought. Examples:
  - `SELECT * FROM R WHERE R.k > 40;`
  - `SELECT * FROM R WHERE R.k >= 10 AND R.k <= 25;`

- To find all keys in the range `[a,b]`:
  - Do a lookup on `a`: leads to leaf where `a` could be
  - Search the leaf for all keys \( \geq a \)
  - If we find a key > `b`, we are done
  - Else follow next-leaf pointer and continue searching in the next leaf
  - Continue until finding a key > `b` or no more leaves
Chapter 17: Coping with System Failures
Operations:

- **Input (x):** block containing x → memory
- **Output (x):** block containing x → disk
- **Read (x,t):** do input(x) if necessary
  \[ t \leftarrow \text{value of } x \text{ in block} \]
- **Write (x,t):** do input(x) if necessary
  \[ \text{value of } x \text{ in block} \leftarrow t \]
Undo logging (Immediate modification)

T: Read (A,t); \( t \leftarrow t \times 2 \) A=B
Write (A,t);
Read (B,t); \( t \leftarrow t \times 2 \)
Write (B,t);
Output (A);
Output (B);

memory

disk

log

\( A:8 \)
\( B:8 \)
\( A:8 \)
\( B:8 \)

\(<T, \text{start}>\)
\(<T, A, 8>\)
\(<T, B, 8>\)
\(<T, \text{commit}>\)
Undo logging rules

(1) For every action generate undo log record (containing old value)

(2) Before $x$ is modified on disk, log records pertaining to $x$ must be on disk (write ahead logging: WAL)

(3) Before commit is flushed to log, all writes of transaction must be reflected on disk
Recovery rules: **Undo logging**

1. Let $S$ = set of transactions with $<T, \text{start}>$ in log, but no $<T, \text{commit}>$ (or $<T, \text{abort}>$) record in log.

2. For each $<T, X, v>$ in log, in reverse order (latest $\rightarrow$ earliest) do:
   - if $T \in S$ then
     - write $(X, v)$
     - output $(X)$

3. For each $T \in S$ do
   - write $<T, \text{abort}>$ to log
Redo logging (deferred modification)

T:  Read(A,t); t ← t×2; write (A,t);
    Read(B,t); t ← t×2; write (B,t);
    Output(A); Output(B)

memory:  A: 8 16
          B: 8 16

DB:        A: 8 16
          B: 8 16

LOG: <T, start>
     <T, A, 16>
     <T, B, 16>
     <T, commit>
Redo logging rules

(1) For every action, generate redo log record (containing new value)

(2) Before X is modified on disk (DB), all log records for transaction that modified X (including commit) must be on disk

(3) Flush log at commit to Disk
Recovery rules: Redo logging

(1) Let $S$ = set of transactions with $<T, \text{commit}>$ in log

(2) For each $<T, X, v>$ in log, in forward order (earliest $\rightarrow$ latest) do:
    - if $T \in S$ then
      \[
      \begin{cases} 
      \text{Write}(X, v) \\
      \text{Output}(X)
      \end{cases}
      \]
Undo/Redo Logging
Solution: undo/redo logging!

Update record in the log has the format

<T, X, new X val, old X val>
Rules

- Buffer containing X can be flushed to disk either before or after T commits
- Log record must be flushed to disk before corresponding updated buffer is (WAL, or “write after logging”)
Recovery with Undo/Redo Logging

1. Redo all committed transactions in order from earliest to latest
   - handles committed transactions with some changes not yet on disk
2. Undo all incomplete transactions in order from latest to earliest
   - handles uncommitted transactions with some changes already on disk
Checkpoint
Non-quiescent checkpoint for undo/redo

LOG

start ckpt
active T's: T1,T2,...

... end ckpt ...

... dirty buffer pool pages flushed ...

... for undo ...

... for ...

...
Recovery process:

- **Backwards pass** (end of log ➔ latest checkpoint start)
  - construct set S of committed transactions
  - undo actions of transactions not in S
- **Undo pending transactions**
  - follow undo chains for transactions in (checkpoint active list) - S
- **Forward pass** (latest checkpoint start ➔ end of log)
  - redo actions of S transactions