CSC343 Notes

Jenci Wei

Fall 2022

Contents

1 Intro

DBMS (Database Management System) creates and manages large amounts of data efficiently and allows it to persist over long periods of time

- *Database*: a collection of data managed by a DBMS
- Every DBMS is based on some *data model*, which includes
  - The structure of the data
  - Constraints onthe content of the data
  - Operations on the data
- DBMS provides the following:
  - Ability to specify the logical structure of the data (explicitly and have it enforced)
  - Ability to query or modify the data
  - Good performance under heavy loads (i.e. huge data and many queries)
  - Durability of the data
  - Concurrent access by multiple users/processes

The *relational data model* is based on the concept of relations in math

- Can think of as tables of rows and columns
- Can't record any data until we have formally defined a schema
- The schema must be strictly adhered to
- All rows should have data for all columns

Other Data Models

- Semi-structured data model
  - More flexible
  - E.g. JSON, XML
- Unstructured data
  - Data that doesn't conform to any fixed structure
  - E.g. document, video, etc.
  - Can be stored in a key-value store
- Graph data model
  - A node could represent a student or a course
  - An edge could represent taking the course
  - A query is defined by a path

2 Relational Model

Relations in Math

- A domain is a set of values
- Suppose $D_1, \ldots, D_n$ are domains
  - The **Cartesian product** $D_1 \times \cdots \times D_n$ is the set of all tuples $< d_1, \ldots, d_n >$ such that $d_i \in D_i$ for all $i \in [1, n]$
- A **(mathematical) relation** on $D_1, \ldots, D_n$ is a subset of the Cartesian product
- A database table is a relation

Schema and Instance

- **Schema** is the structure of a relation
  - E.g. teams have 3 attributes: name, home field, coach. No two teams can have the same name.
  - Formal notation: Teams(Name, HomeField, Coach)
- **Instance** is the content of a relation
  - Data in a table
- Changes in a database
  - Instances change constantly
  - Schemas (should) change rarely
  - Databases usually store the current version of the data
    - Databases that record the history are called *temporal databases*

Terminology

- **Relation**: table
- **Attribute**: column
- **Tuple**: row
- **Arity** of a relation: number of attributes
- **Cardinality** of a relation: number of tuples

Relations as Sets

- A relation is a *set* of tuples, which means
  1. There can be no duplicate tuples
  2. Order of the tuples doesn't matter
- We can have a model where relations are *bags*
  - A generalization of sets that allows duplicates
  - Commercial DBMSs use this model

Database Schema and Instance

- **Database schema**: a set of relation schemas

- **Database instance**: a set of relation instances

Key

- A **key** for a relation is a set of attributes $a_1, \ldots, a_n$ such that
  1. Their combined values are unique, i.e. $\nexists t_1, t_2$ such that $(t_1.a_1 = t_2.a_1) \wedge \cdots \wedge (t_1.a_n = t_2.a_n)$
  2. No subset of $a_1, \ldots, a_n$ has this property/is a key
- Notation for key: underline
  - E.g. Teams(Name, HomeField, Coach)
- A key defines a kind of *integrity constraint*

Superkey

- **Superkey** any superset of a key
- Useful for database theory, but in practice, we don't declare superkeys

Key vs. Superkey

- *Key*: a *minimal* set of atttributes such that no two tuples can have the same values on all of these attributes
- *Superkey*: a superset of some key (not necessarily minimal)

Foreign Keys

- Relations often refer to each other
- The referring attribute is called a **foreign key** because it refers to an attribute that is a key in another table
- A foreign key may need to have several attributes
- Notation $R[A]$ is the set of all tuples from relation R, but with only the attributes in list A
- We declare foreign key constraints with notation $R_1[X] \subseteq R_2[Y]$
  - X and Y may be lists of attributes, of same arity
  - Y must be a key in R_2

Referential Integrity Constraints

- Relationships such as $R_1[X] \subseteq R_2[Y]$ are a kind of **referential integrity constraint**
- Not all referential integrity constraints are foreign key constraints
  - They don't have to satisfy the property that Y is a *key* in R_2

3 Relational Algebra

Intro

- Operands: tables
- Operators:
  - Choose only the rows we want
  - Choose only the columns we want
  - Combine tables
  - Etc.

Simplifications

- For relational algebra, we assume that
  1. Relations are *sets*, so no two rows are the same
  2. Every cell has a value
- We would drop these assumptions for SQL

A **query** on a set of relations produces a relation as a result

- E.g. we have relations:
  - College(cName, state, enrolment)
  - Student(sID, sName, GPA, sizeHS)
  - Apply(sID, cName, major, decision)
- Foreign key constraints
  - Apply[sID] $\subseteq$ Student[sID]
  - Apply[cName] $\subseteq$ College[cName]
- Simplest query: the relation name
- Can use *operators* to filter, slice, and combine

The **select** operator picks certain rows

- Denoted by σ
- Notation: σ_{cond} Rel
  - Condition is a boolean expression
  - Result is a relation with the same schema as the operand, but with only the tuples that satisfy the condition
- E.g. students with GPA > 3.7
  - $\sigma_{\text{GPA}>3.7}$ Student
- E.g. students with GPA > 3.7 and HS < 1000
  - $\sigma_{\text{GPA}>3.7 \wedge \text{HS}<1000}$ Student
- E.g. applications to Stanford CS major

  - $\sigma_{\text{cName='Stanford'} \wedge \text{major='cs'}}$ Apply

The **project** operator picks certain columns

- Denoted by Π
- Notation: $\Pi_{A_1,\ldots,A_n}$ Rel
  - $\{A_1, \ldots, A_n\}$ is a subset of the attributes of the Rel
  - Result is a relation with all tuples from Rel, but with only the attributes in $\{A_1, \ldots, A_n\}$
- E.g. ID and decision of all applications
  - $\Pi_{\text{sID,dec}}$ Apply
- E.g. ID and name of students with GPA > 3.7
  - $\Pi_{\text{sID,sName}}\,(\sigma_{\text{GPA}>3.7}$ Student)
- Duplicate values are eliminated in relational algebra
  - Different from SQL, which is based on multisets/bags that does not eliminate duplicates

The **cross-product/Cartesian product** operator combines two relations

- Notation: $R_1 \times R_2$
  - The result is a relation with every combination of a tuple from R_1 concatenated to a tuple from R_2
  - Schema is every attribute from R_1 followed by every attribute from R_2
- E.g. Student $\times$ Apply
  - Has 8 columns: Student.sID, sName, GPA, HS, Apply.sID, cName, major, decision
  - If Student has S tuples and Apply has A tuples, Student $\times$ Apply has $S \times A$ tuples
- Can introduce nonsense tuples, which we could get rid of using selects

Natural Join

- Notation: $R \bowtie S$
- Result is defined by
  1. Taking the Cartesian product
  2. Selecting to ensure equality on attributes that are in both relations (as determined by *name*)
  3. Projecting to remove duplicate attributes
- Properties
  - Commutative: $R \bowtie S = S \bowtie R$
  - Associative: $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
    - Therefore can write $R \bowtie S \bowtie T$
- When no attributes are in common, then the result is the Cartesian product
- When no tuples match, then the result is the empty relation
- When two relations have exactly the same attribute, then the result is the intersection

- May over-match since two attributes could have the same name but we don't want them to match
- May under-match since two attributes could have different names but we want them to match

Theta Join

- Notation: $R \bowtie_\theta S$

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

  - θ is the condition

Operator Precedence

- Expressions can be composed recursively
- Parentheses and precedence rules define the order of evaluation

Higher ↑

σ, Π, ρ
$\times$, $\bowtie$, $\bowtie_\theta$
$\cap$
$\cup$, $-$

Lower ↓

Expression Trees

- A way of breaking down complex expressions
- Leaves are relations
- Interior nodes are operators
- Very similar to abstract syntax trees (CSC111)

Assignment Operator

- Notation: $R(A_1, \ldots, A_n) :=$ Expression
- Allows us to name all the attributes of the new relation
- Even if the names aren't changing, it is better to specify the names for readability
- R must be a temporary variable, not one of the relations in the schema
- E.g. DeansListScholars(sID, sName, GPA, sizeHS) := $\sigma_{\text{GPA} \geq 3.5}$Student

Rename Operation

- Notation: $\rho_{R_1}(R_2)$
- Alternate notation: $\rho_{R_1(A_1,\ldots,A_n)}(R_2)$
  - Can rename all the attributes as well as the relation
  - $R_1(A_1, \ldots, A_n) := R_2 \quad \equiv \quad R_1 := \rho_{R_1(A_1,\ldots,A_n)}(R_2)$
- Useful if we want to rename *within* an expression

Syntactic Sugar

- Some operations are not necessary, since we could get the same effect using a combination of other operations
- E.g. natural join, theta join

Set Operations

- Can use them since relations are sets
- Use only if the operands are relations over the same attributes (i.e. in number, name, and order)
- The set operators $\cap, \cup, -$ work the same way as usual in relational algebra

Integrity Constraints

- Notation to expression inclusion dependencies between relatoins R_1 and R_2: $R_1[X] \subseteq R_2[Y]$
- Suppose R and S are expressions in RA, we can write a constraint in either of the following ways:
  - $R = \varnothing$
  - $R \subseteq S$, which is equivalent to $R - S = \varnothing$

Strategies for Specific Types of Query

- *Max* (or analogously *min*)
  - Pair tuples and find those that are *not* the max
  - Subtract from all to find the maxes
- *k or more*
  - Make all combos of k different tuples that satisfy the condition (i.e. using self-join)
- *Exactly k*
  - "k or more" $-$ "$(k+1)$ or more"
- *Every*
  - Make all combos that should have occurred
  - Subtract those that *did* occur to find those that didn't always, which are failures
  - Subtract the failures from all

Relational Calculus

- Another abstract query language for the relational model
- Based on first-order logic
- Query example: $\{t | t \in \text{Movies} \wedge t[\text{director}] = \text{Scott}\}$
- Expressive power same as RA, which is "relationally complete"

4 Introduction to SQL

SQL: Structured Query Language

- DDL: Data Definition Language, used for defining schemas
- DML: Data Manipulation Language, used for writing queries and modifying database
- We will use PostgreSQL

SELECT-FROM-WHERE queries

- E.g.

```
SELECT name           -- choose the column called "name"
FROM Course           -- from the Course table
WHERE dept = 'CSC';   -- choose only rows that satisfy
```

- `WHERE` is equivalent to σ in relational algebra
  - Can use logical operators `=, <, >, <=, >=, !=, <>`
  - `!=` and `<>` mean "not equals"
  - Can use boolean operators `AND, OR, NOT`
- `SELECT` is equivalent to Π

Cartesian Product

- Put 2+ tables in a comma-separated list in the `FROM` clause
- E.g. `FROM Course, Offering, Took`

Temporarily renaming a table

- E.g.

```
SELECT e.name, d.name
FROM Employee e, Department d;
```

- Equivalent to ρ in relational algebra
- Renaming is *required* for self-joins

Wildcard in the `SELECT` clause

- Use a wildcard if we want every column to be included in the result
- E.g.

```
SELECT *
FROM ...
```

Naming Columns

- Can use `AS` expression to choose a column name in the result of a query
- E.g.

```
SELECT name AS title, dept
FROM ...
```

- If the column name cannot be determined (e.g. an expression), `?column?` will be used as the name

Sorting

- Can sort the results of a query by adding an `ORDER BY` clause to the end of a select-from-where query
- E.g.

```
SELECT sid, grade
FROM Took
WHERE grade > 90
ORDER BY grade;
```

- Default order is ascending
- Can override to descending by adding `DESC`
  - i.e. `ORDER BY grade DESC`
- Can order according to multiple columns
  - E.g. `ORDER BY grade, sid`
- Can use the value of an expression to determine ordering
  - E.g. `ORDER BY grade + sid`
- Done before `SELECT`, so all attributes are available

Expressions in a `SELECT` clause

- Can use an expression in a `SELECT` clause
  - Operands: attributes, constants
  - Operators: arithmetic operators, string operators
- E.g.

```
SELECT sid, grade + 10 AS adjusted
FROM ...
```

- E.g.

```
SELECT dept || cnum
FROM ...
```

  - `||` is string concatenation

Constant Expressions

- A `SELECT` clause can use a constant value in a column, e.g.

```
SELECT name, 'satisfies' as breadthRequirement
FROM Course
WHERE breadth;
```

  - This extracts only courses for which the value of `breadth` is true
  - The second column has value `satisfies` in every row

Pattern Matching

- The `LIKE` operator can compare a string to a pattern
- Notation: `<attribute> LIKE <pattern>` or `<attribute> NOT LIKE <pattern>`
- A pattern is a quoted string and can include the following special characters:
  1. _ matches any single character
  2. `%` matches any sequence of 0 or more characters
- E.g.

```
SELECT *
FROM Course
WHERE name LIKE '%to%';
```

  - Result would include 'Intro to Databases', 'Intro to Machine Learning', etc.
- The ~ operator supports regular expressions for string matching
  - E.g.

```
SELECT *
FROM Student
WHERE surname ~ '(M|F|L)a*'
```

  - Regular expressions can be slow

Case-Sensitivity and Whitespace

- Keywords (e.g. `SELECT`) are not case-sensitive
- Identifiers (i.e. names of tables or columns) are not case-sensitive
- Literal strings (e.g. `'StG'`) are case-sensitive, and require single quotes
- Line breaks and tabs are ignored by SQL
- Reasonable query format:
  1. Put each clause on a line
  2. Use all capitals for keywords
  3. Use a leading capital for table names
  4. Use lowercase for column names

SQL is a high-level language

- We only care about what we want from the database, not how to get it
- The DBMS can change how the data is stored with no impact on our queries
- We have *physical data independence*

5 Aggregation and Grouping

Computing on a Column

- SQL provides functions, such as `sum, avg, min, max, count`, that can be used in a `SELECT` clause to apply to a column
- Called *aggregation*
- E.g.

```
SELECT max(grade) - min(grade)
FROM Took;
```

  - Outputs one column `?column?` that has 1 row

- E.g.

```
SELECT count(*)
FROM Took;
```

  - Outputs one column `count` that has 1 row, representing the number of rows in the table

Duplicate Values

- Duplicate values contribute to aggregations
  - E.g. if the grade 85 occurs 10 times in the table, all of them contribute to `avg(grade)`, `count(grade)`, etc.
- To count each duplicate only once, use `DISTINCT`
  - E.g.

```
SELECT count(DISTINCT dept)
FROM Offering;
```

  - `DISTINCT` does not affect `min` or `max`

Multiple Aggregations

- The quantities need to be alike (i.e. one max, one min, one avg, etc.) in order to produce a structurally sound table
  - E.g.

```
SELECT max(grade), min(grade), count(distinct oid), count(*)
FROM Took;
```

Grouping

- If we follow a *SELECT-FROM-WHERE* expression with `GROUP BY`, the tuples that have the same value for that attribute will be treated as a *group*, and *one row* will be generated for each group
- Need to ensure that SQL makes a structurally sound, rectangular table to hold the results
- E.g.

```
SELECT sid, avg(grade)
FROM Took
GROUP BY sid
ORDER BY avg(grade);
```

  - All rows with the same `sid` collapses down to 1 row in the result, giving us each student's average

- We can order by something not in the `SELECT` clause
  - E.g.

```
SELECT sid, avg(grade)
FROM Took
GROUP BY sid
ORDER BY count(oid);
```

  - This is valid since each `count(oid)` correspond to an `sid` group
    - Would not be valid if we only put `oid`, since each `sid` may have multiple `oid`s

Grouping By Multiple Columns

- If we want to group by two columns, we need to ensure that there would be just one value of any other attribute per (col1, col2) combination
- E.g.

```
SELECT dept, cnum, count(cnum)
FROM Offering
GROUP BY dept, cnum
ORDER BY dept;
```

- `ORDER BY` comes after `GROUP BY`

Restrictions on Aggregation

- If any aggregation is used in a query, then each element of the `SELECT` list must be either:
  1. Aggregated, or
  2. An attribute in the `GROUP BY` list
- We can slightly break this rule in PostgreSQL, e.g.

```
SELECT sid, firstname, surname
FROM Student
GROUP BY sid;
```

  - Each `sid` has exactly 1 value for `firstname` and `surname`

6 Filtering Groups with `HAVING`

`HAVING` allows us to filter aggregated values

- Syntax:

```
...
GROUP BY <attributes>
HAVING <condition>
```

  - Only groups satisfying the condition are kept

- E.g.

```
SELECT oid, avg(grade), count(*)
FROM Took
GROUP BY oid
HAVING count(*) > 1
```

- Can also filter according to the value of an unaggregated attribute (as long as it is within the `GROUP BY` clause)
- Can filter on something that is not in the `SELECT` clause
  - `HAVING` is executed *before* the `SELECT` clause

Restrictions on `HAVING` Clauses

- May refer to an attribute only if it is either *aggregated* or is an attribute on the `GROUP BY` list

Order of Execution

1. `FROM` clause
   - Determines what table(s) will be examined
2. `WHERE` clause
   - Filters rows
   - Since `SELECT` has not happened yet, we cannot reference any column names that it defines (e.g. renames)
3. `GROUP BY` clause
   - Organizes the rows into groups, each of which will be represented by 1 row in the result table
4. `HAVING` clause
   - Filters groups
   - Since this happens before the `SELECT` clause, it *can* refer to attributes that are not included in the `SELECT`
5. `SELECT` clause
   - Chooses which columns to include in the result
   - May introduce new column names
6. `ORDER BY` clause
   - Sorts the rows of the result table
   - Since it occurs *after* the `SELECT` clause, it can reference column names that are introduced there

7 Views

We sometimes want to define new things in terms of old things

- We could do so by creating a new table, and putting the correct data into it
- However, if the old table is updated, the newly created table would be outdated
- Could take up much space

Virtual Views

- When we define a view, we are just asking SQL to remember a definition
- E.g.

```
1  CREATE VIEW DeansHonoursStudent AS
2  SELECT sid, term, avg(grade)
3  FRON Took JOIN Offering ON Took.oid = Offering.oid
4  GROUP BY sid, term
5  HAVING avg(grade) > 80;
```

  - When we say `DeansHonoursStudent`, we actually mean the code of its definition
- Dynamic to changes since they are recomputed every time they are used
  - Recomputation could sometimes be costly
- "Virtual" because other than definition, it is not stored
- Can be used to break down a large query
- Provides another way of looking at the same data
- We can grant users privileges to access the view but not the base tables it came from

Materialized View

- Computed and stored at the time it is defined
- Relationship between it and its base tables is maintained at all times
- Expensive to maintain the relationship between a materialized view and its base tables

8 Sets and Bags in SQL

SQL permits a table to have duplicate rows

- Provided that this does not violate any constraint that has been defined (e.g. primary keys)
- Tables in SQL are *bags* (also known as a *multiset*)
  - Generalization of a set that allows duplicates
  - Order does not matter
- Duplicates in the result tell us how many times something occurred
- Getting rid of duplicates is expensive

We can use `DISTINCT` to eliminate duplicates

- E.g.

```
SELECT DISTINCT oid
FROM Took;
```

- We can aonly ask for distinct rows overall, *not* by column
- Turns the query into a *set*

We can use `DISTINCT` inside an aggregation

- E.g.

```
SELECT count(DISTINCT sid), count(DISTINCT oid)
FROM Took;
```

Set Union, Intersection, and Difference in SQL

- Set union: `UNION`
- Set intersection: `INTERSECT`
- Set difference: `EXCEPT`
- Syntax

```
(<subquery>) UNION/INTERSECT/EXCEPT (<subquery>)
```

  - An operand to a set operator must be surrounded by *round brackets*
  - An operand to a set operator must be a *complete query* (e.g. select ... from ...)
    - *Cannot* simply use two relation names
  - The operands to a set operator must have *compatible schemas*
    - Details differ from DBMSs
- They work under *set semantics* (rather than bag semantics)
  - Duplicates are eliminated after a set operation
  - The operands are *first transformed into a set* before the set operation is applied

We can use `ALL` on set operations to keep duplicates

- E.g.

```
(SELECT...FROM...)
UNION ALL
(SELECT...FROM...);
```

- Bag union: one bag is extended after the other
  - E.g. $\{1,1,1,3,7,7,8\} \cup \{1,5,7,7,8,8\} = \{1,1,1,1,3,5,7,7,7,7,8,8,8\}$
- Bag intersection: take matching elements one by one
  - E.g. $\{1,1,1,3,7,7,8\} \cap \{1,5,7,7,8,8\} = \{1,7,7,8\}$
- Bag difference: remove matching elements one by one
  - E.g. $\{1,1,1,3,7,7,8\} - \{1,5,7,7,8,8\} = \{1,1,3\}$

9 Null Values in SQL

We can get a `NULL` value into a table using `INSERT INTO`

- E.g.

```
1          INSERT INTO SomeTable values (value1, NULL, value3, ...);
```

- If we don't want to allow `NULL` values in a given column, we can use a `NOT NULL` constraint in the table definition
- Can compare a value to `NULL` using `IS NULL`

The Unknown Truth Value

- With `NULL` values, we sometimes don't know if a condition is true or false
  - In which case it evaluates to "unknown"
- `WHERE` does *not* include a row for which the truth-value of the clause is unknown
  - Same goes with `NATURAL JOIN`, which is essentially a Cartesian product followed by a `WHERE` condition

Impact of `NULL` Values on Aggregation

- When we aggregate on a columns, `NULL` values in that column are ignored

	some nulls in A	All nulls in A
`min(A)`	ignore the nulls	null
`max(A)`		
`sum(A)`		
`avg(A)`		
`count(A)`		0
`count(*)`	all tuples count	

- Corner cases are handled differently by different DBMSs
  - E.g. are two `NULL` values considered distinct from each other when we do `NATURAL JOIN`?

Impact of `NULL` Values on Boolean Conditions

- Truth tables:

A	B	A and B	A or B
T	T	T	T
TF or FT		F	T
F	F	F	F
TU or UT		U	T
FU or UF		F	U
U	U	U	U

A	not A
T	F
F	T
U	U

- If we say `WHERE condition OR NOT condition`, `NULL` values are excluded

Summary

- Any comparison with `NULL` yields the truth-value unknown
- `WHERE` only accepts `TRUE` (same goes with `NATURAL JOIN`)
- Aggregation ignores `NULL` values
- In other situations regarding the behaviour of `NULL`, check the documentation of the DBMS since one DBMS may behave differently from another

10 Joins

All joins from relational algebra have a counterpart in SQL

- Correspondences:

Expression	Meaning
`R, S`	$R \times S$
`R cross join S`	
`R natural join S`	$R \bowtie S$
`R join S on Condition`	$R \bowtie_{condition} S$

- With `NATURAL JOIN` on tables R and S, R and S must agree on all attributes of the same name
  - E.g.

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL JOIN S

A	B	C
1	2	3

- With `JOIN ON` on tables R and S, R and S must satisfy the join condition
- If a row does not satisfy the join condition, it is excluded from the resulting table
  - Such row is called a **dangling tuple**
  - Use an outer join to include them

Outer Joins

- An **outer join** preserves dangling tuples by padding them with null values where needed
  - The joins that don't pad with nulls are called **inner joins**
- A **left outer join** preserves dangling tuples from the table on the left side only
  - E.g.

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL LEFT JOIN S

A	B	C
1	2	3
4	5	NULL

- A **right outer join** preserves dangling tuples from the table on the right side only

  - E.g.

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL RIGHT JOIN S

A	B	C
1	2	3
NULL	6	7

- A **full outer join** preserves all dangling tuples
  - E.g.

R

A	B
1	2
4	5

S

B	C
2	3
6	7

R NATURAL FULL JOIN S

A	B	C
1	2	3
4	5	NULL
NULL	6	7

SQL Syntax for Outer Joins

- A `NATURAL JOIN` or a `JOIN ON` (theta join) can optionally preserve dangling tuples
- SQL Syntax:

Cartesian product

```
A CROSS JOIN B          same as A,B
```

Theta-join

```
A JOIN B ON C
✓A {LEFT|RIGHT|FULL} JOIN B ON C
```

Natural join

```
A NATURAL JOIN B
✓A NATURAL {LEFT|RIGHT|FULL} JOIN B
```

✓ indicates that tuples are padded when needed.

Natural Join is Brittle

- If our schema changes, the meaning of natural joins could change
- The join condition is inferred from the schema rather than explicitly written by the programmer
- It is better to use `JOIN ON` than `NATURAL JOIN`

`JOIN USING`

- Similar to a natural join, but we can specify which columns to match
- E.g.

```
SELECT sID, instructor
FROM Student
    JOIN Took Using (sID)
    JOIN Offering USING (oID)
WHERE grade > 50;
```

- Need to use *round brackets* to enclose our list of attributes to be matched
- Not as brittle as `NATURAL JOIN`, but requires the column names to be the same

11 Subqueries

Subqueries with Set Operations

- Use `INTERSECT, UNION, EXCEPT` to join two operands, each operand is a subquery
- E.g.

```
(SELECT sid FROM Took WHERE grade > 95)
INTERSECT
(SELECT sid FROM took WHERE grade < 50);
```

Subquery in a `FROM` Clause

- The query considers the subquery as a table
- Two syntactic requirements:
  1. The subquery must be enclosed in round brackets
  2. We must name the result of the subquery, allowing us to refer to it elsewhere
- E.g.

```
SELECT ...
FROM A, (SELECT ... FROM ...) B
WHERE A.x = B.x;
```

Subquery as a Value in a `WHERE` Clause

- If the subquery produces exactly 1 row, we can compare to it in a `WHERE` clause
  - E.g.

```
SELECT sID
FROM Student
WHERE cgpa >
    (SELECT cgpa
    FROM Student
    WHERE sID = 99999);
```

    - Since `sID` is the primary key of the `Student` table, the subquery cannot produce more than 1 row
  - If the subquery returns an empty table, then the result is an empty table (since comparing to emptiness is always false)
  - If the subquery returns more than 1 row, an error would be thrown
  - We *cannot* do the analogous comparison in RA

Quantifying Over Multiple Results

- To compare an individual value to all those returned by the subquery, we have to tell SQL which quantifier to use
- `ALL`: universal quantification
  - Value is true iff the comparison holds for *every* tuple in the subquery result

- `SOME`: existential quantification
  - Can also use `ANY`
  - Value is true iff the comparison holds for *at least one* tuple in the subquery result
- `IN` and `NOT IN`
  - Value is true iff the value is in (not in) the set of rows generated by the subquery
  - `IN` is equivalent to `= ANY`
  - `NOT IN` is equivalent to `<> ALL`
- `EXISTS` does not involve any comparison, it just checks if the subquery returns any rows
  - Value is true iff the subquery has at least 1 tuple
  - "Exists a row in the subquery result"
  - Efficient since it short-circuits as soon as it finds a row

Scope

- If a name might refer to more than 1 thing, use the most closely nested one
- If a subquery refers only to names defined inside it, then it can be evaluated *only once* and be used repeatedly in the outer query
- If a subquery refers to any name defined outside of itself, then it must be evaluated *once for each tuple in the outer query*, called **correlated subquery**

Correlated Subqueries

- A subquery is **correlated** if it refers to an attribute of the outer query
- Every time we consider a new row, the subquery must be recomputed
- E.g.

```
SELECT instructor
FROM Offering
WHERE EXISTS (
    SELECT *
    FROM Took
    WHERE Took.oID = Offering.oID AND grade > 98
);
```

- An **uncorrelated subquery** makes no reference to the outer query
  - It can be computed once and reused over and over for each row of the outer query

Summary

- Where can subqueries go?
  - As a relation in a `FROM` clause
  - As a value in a `WHERE` clause
  - With `ANY`, `ALL`, `IN`, `NOT IN`, `EXISTS` in a `WHERE` clause
  - As operands to `UNION`, `INTERSECT`, `EXCEPT`

x *«comparison»* ALL (*«subquery»*)

$\forall$ y $\in$ *«subquery results»* | x *«comparison»* y

x *«comparison»* SOME (*«subquery»*)

$\exists$ y $\in$ *«subquery results»* | x *«comparison»* y

x IN (*«subquery»*)

Same as x = SOME (*«subquery»*)

x NOT IN (*«subquery»*)

Same as x <> ALL (*«subquery»*)

just for convenience

EXISTS (*«subquery»*)

$\exists$ y $\in$ *«subquery results»*

12 Database Modifications

Insertion of Literal Values

- E.g.

```
CREATE TABLE Ages(name TEXT, age INT);

INSERT INTO AGES VALUES
('Alice', 21), ('Bob', 25), ('Carol', NULL), ('Dave', 0);
```

Insertion of Computed Values

- E.g. we want to find everyone who has taken a first-year course and insert them into the table using 19 for their age

```
INSERT INTO Ages
(SELECT DISTINCT firstname, 19 AS age
FROM Student JOIN Took USING (sID)
JOIN Offering USING (oID)
WHERE cnum <= 199);
```

- `INSERT INTO` adds the new rows to what was already in the table

Insertion with Default Values

- We name the attributes that we *are* providing values for
- E.g.

```
CREATE TABLE Invite(
    name TEXT,
    campus TEXT DEFAULT 'StG',
    email TEXT,
    age INT);

INSERT INTO Invite(name, email)
(SELECT firstname, email
FROM Student
WHERE cgpa > 3.5);
```

  - We provide values for name and email, and let the DBMS fill in default values for the other columns

- If we don't explicitly define a default value, `NULL` is used

Deletion

- Specify the condition that must be satisfied in order for a row to be deleted
- E.g. delete failing grades

```
DELETE FROM Took
WHERE grade < 50;
```

- To delete *all* rows from a table, we omit the `WHERE` condition, e.g.

```
DELETE FROM Took;
```

Updates

- Use an `UPDATE` statement to change certain attributes of certain rows
- Syntax:

```
UPDATE <table> SET <list of attribute assignments> WHERE <condition on tuples>;
```

- E.g.

```
UPDATE Student
SET campus = 'UTM'
WHERE sID = 99999;
```

- Can update multiple rows at once

13 SQL Schemas

SQL Schemas

- In relational algebra a *schema* is the definitions of the structure of a relation or a set of relations
- In SQL, a schema is a namespace that prevents name conflicts
- By default, all definitions go to the schema `public`
- Can create schema

```
CREATE SCHEMA eatInToronto;
```

- To create a table inside a schema, name it as `schema.table`

```
CREATE TABLE eatInToronto.Restaurant (
    name TEXT,
    cuisine TEXT
)
```

- Can refer to such table using its full name

```
SELECT name
FROM eatInToronto.Restaurant;
```

The Search Path

- To see the current search path:

```
show search_path;
```

- We can set the search path to avoid referring to a table by its full name

```
SET SEARCH_PATH TO university, eatInToronto, public;
```

  - First look at `university`, then `eatInToronto`, then `public`
- If we define a table without specifying a schema, it will be created in the first schema in the search path
- Default search path is `$user, public`
  - `$user` is the name of the current user, which is not automatically created
  - If it s created, then it will be at the front of the default search path
- Can see the current search path with

```
SHOW SEARCH_PATH;
```

Removing a Schema

- We can remove a schema by

```
DROP SCHEMA university;
```

- If anything is already defined in the schema, dropping it will generate an error message
- If we intend to drop the schema and remove everything in it, add keyword `CASCADE`

```
1           DROP SCHEMA university CASCADE;
```

- To avoid getting an error message if the schema does not exist, add `if exists`
- At the top of the DDL file, it is useful to have a `DROP SCHEMA` statement followed by a `CREATE SCHEMA` statement, then followed by a `SET SEARCH_PATH` statement

```
1           DROP SCHEMA IF EXISTS university CASCADE;
2           CREATE SCHEMA university;
3           SET SEARCH_PATH TO university;
```

14 Data Definition Language (DDL)

Types

- When creating a table, we must define the type of each column

Built-in Types

- `CHAR(n)`: fixed-length string of n characters, padded with blanks if necessary
- `VARCHAR(n)`: variable-length string of up to n characters
- `TEXT`: variable-length, unlimited
  - Not in the SQL standard, supported by psql
  - E.g. `'Shakespeare''s Sonnets'`
  - Must surround with single quotes, not double quotes
  - Can put a literal single quote in a string by using an escape character, which is also a single quote
- `INT`, `INTEGER`
  - E.g. `37`
- `FLOAT`, `REAL`
  - E.g. `3.14159`, `3.14159e-10`
- `BOOLEAN`
  - E.g. `TRUE`, `FALSE`
  - Case doesn't matter
- `DATE`
  - E.g. `2018-03-01`, `'September 11, 2001'`
- `TIME`
  - E.g. `'12:34:56'`, `'12:34:56.789'`
- `TIMESTAMP` (date + time)
- Etc.

User-Defined Types

- Defined in terms of a built-in type
- Can define constraints and default values
- E.g.

```
create domain Grade as int
    default null
    check (value >= 0 and value <= 100);
create domain Campus as varchar(4)
    default 'StG'
    check (value in ('StG', 'UTM', 'UTSC'));
```

Semantics of Type Constraints

- **Syntax**: which strings are valid
- **Semantics**: what does a valid string mean/do
- Constraints on a type are checked every time a value is assigned to a column of that type

Semantics of Default Values

- The default value for a type is used when no value has been specified
- Different from **type default**, which is for *every* column of that type in *any* table

Primary Key Constraints

- Declaring that a set of one or more columns are the `PRIMARY KEY` for a table means
  - Their values will be unique across rows
  - Their values will never be null
  - Hint to DBMS: optimize for searches by this set of columns
- Every table must have 0 or 1 primary key
  - In practice, every table should have 1
- Can be declared at the end of the table definition

```
create table Person (
    ID integer,
    name varchar(25),
    primary key (ID, name)
);
```

  - Brackets are required
- For single-column key, can declare it with the column

```
create table Person (
    ID integer primary key,
    name varchar(25)
);
```

Uniqueness Constraints

- Declaring that a set of one or more attributes is `UNIQUE` means
  - Their values will be unique across rows
  - But their values *can* be null
- If we don't want them to be null, we need to separately declare that
- Can declare more than 1 set of attributes to be `UNIQUE`
- Can be declared at the end of the table definition

```
create table Person (
    ID integer,
    name varchar(25),
    unique (ID, name)
);
```

  - Brackets are required

- For single-column key, can declare it with the column

```
create table Person (
    ID integer primary key,
    name varchar(25) unique
);
```

- For uniqueness, no two nulls are considered equal, so two null inserts are allowed

Foreign Key Constraints

- E.g. `foreign key (sID) references Student`
  - Not specifying which attribute to reference to defaults to the primary key
- Column `sID` in this table is a foreign key that references the primary key of `Student`
- Every value for `sID` in this table must actually occur in the `Student` table
- Requirements: must be declared either primary key or unique in the "home" table
- Declare as a separate table element, or with the column (only possible if just a single column is involved)
- Can reference columns that are not the primary key as long as they are unique, in which case we need to specify such column

```
create table People (
    SIN integger primary key,
    name text,
    OHIP text unique
);
create table Volunteeers (
    email text primary key,
    OHIPnum text references People(OHIP)
)
```

Enforcing Foreign-Key Constraints

- The DBMS must ensure that
  - The referenced columns are `PRIMARY KEY` or `UNIQUE`
    - Checked once when schema declared
  - The values actually exist
    - Checked upon every insert, delete, and update
- We can define what the DBMS should do in case of a violation, called specifying a *reaction policy*

"Check" Constraints

- Can have a check clause on a user-defined domain
- Can define a check constraint
  - on a column
  - on the rows of a table

  - across tables

Column-Based "Check" Constraints

- Defined with a single column and constrain its value (in every row)
- E.g.

```
create table Application (
    sID integer,
    previous integer check (previous >= 0)
);
```

  - Column `previous` can only be positive

- Constrain that column's value in every row
- Condition can be anything that could go in a `WHERE` clause
- Some DBMS allows subqueries in column-based constraints, but not psql
- Checked only when a row is inserted into that table, or updated
- If a change somewhere else violates the constraint, the DBMS will not notice
  - E.g. in a subquery, when something in the other table changes

"Not Null" Constraints

- Specific kind of column-based constraint
- Declare that a column of a table is not null

```
create table Application (
    sID integer,
    previous integer not null
);
```

- Many columns should be not null in practice

Row-Based "Check" Constraints

- Defined as a separate element of the table schema
- Can refer to any column of the table
- Condition can be anything that could go in a `WHERE` clause, and can include a subquery
- E.g.

```
create table Student (
    sID integer,
    age integer,
    year integer,
    college varchar(4),
    check (year = age - 18),
    check college in (select name from Colleges)
);
```

- Checked only when a row is inserted to taht table, or updated
- If a change somewhere else violates the constraint, the DBMS will not notice
- A check constraint only fails if it evaluates to false
  - Unknown truth value (from null) would pass the check
  - Can prevent this by using not null constraints

Naming Constraints

- Can name a constraint and have more helpful error messages
- Can be done with any of the above constraints
- Add `constraint <name>` before the `check (<condition>)`, e.g.

```
create domain Campus as varchar(4)
    not null
    constraint validCampus check (value in ('StG', 'UTM', 'UTSC'));
```

Cross-Table Constraints: Assertions

- Assertions are schema elements at the top level, so can express cross-table constraints

```
create assertion (<name>) check (<predicate>);
```

- Assertions are costly because they have to be checked upon every database update, and each check can be expensive
- Testing and maintenance are also difficult
- Not supported by psql (as well as most other DBMSs)

Triggers

- We can specify a type of database event that we want to respond to, e.g.

```
after delete on Courses
```

or

```
before update of grade on Took
```

- We specify the result, e.g.

```
insert into Winners values (sID)
```

- The response is wrapped up in a function
- E.g. (function)

```
create function RecordWinner() returns trigger as
$$
BEGIN
    IF NEW.grade >= 85 THEN
        INSERT INTO Winners VALUES (NEW.sid);
```

```
        END IF;
        RETURN NEW;
    END;
    $$
    LANGUAGE plpgsql;
```

- `BEGIN` and `END` specify the body of the function
- Double dollar-signs are "quotation marks" that surround the body of the function

- E.g. (trigger)

```
create trigger TookUpdate
before insert on Took
for each row
execute procedure RecordWinner();
```

Reaction Policies

- `CASCADE`: propoagate the change to the referring table
- `SET NULL`: set the referring column(s) to null
- `RESTRICT`: prevent the deletion/update
- If we say nothing, the default is to forbid the change in the referred-to table (i.e. `RESTRICT`)
- Asymmetry of Reaction Policies
  - Suppose table R refers to table S
  - We can define "fixes" that propagate changes backwards from S to R
  - We *cannot* define "fixes" that propagate changes forwards from R to S
- Syntax:

```
foreign key (sID) references Student on delete cascade
```

  - Can use `on delete` (i.e. when a deletion creates a dangling reference), or `on update` (i.e. when an update creates a dangling reference), or both
    - E.g. `on delete restrict on update cascade`

Semantics of Deletion

- If we are deleting a few rows where deleting one of them violates a foreign key constraint, the DBMS would do the following:
  1. Halt at the error but keep any earlier deletes
  2. Roll back the earlier deletes and make none at all
  3. Make all the deletes except the violating ones
- If deleting one row affects the outcome for a row encountered later:
  - Deletion first marks all rows for which the `WHERE` condition is satisfied
  - Then it deletes the marked rows

Updating the Schema

- Can alter a domain or table

```
alter table Course
    add column numSections integer;
alter table Course
    drop column breadth;
```

- The `DROP` keyword removes a domain, table, or a whole schema
- If we drop a table that is referenced by another table, then we must specify `CASCADE`, which removes all referring rows

More About DDL

- Can define *indices*, which makes search faster
- Can define *privileges*, which specifies who can do what with which parts of the database

15 Embedded SQL

Problems with SQL

- Standard SQL is not Turing-complete
  - No loops or recursion
- Cannot control format of output
- Most users shouldn't be writing SQL queries
  - Want to run queries that are *based on user input*

SQL + a Conventional Language

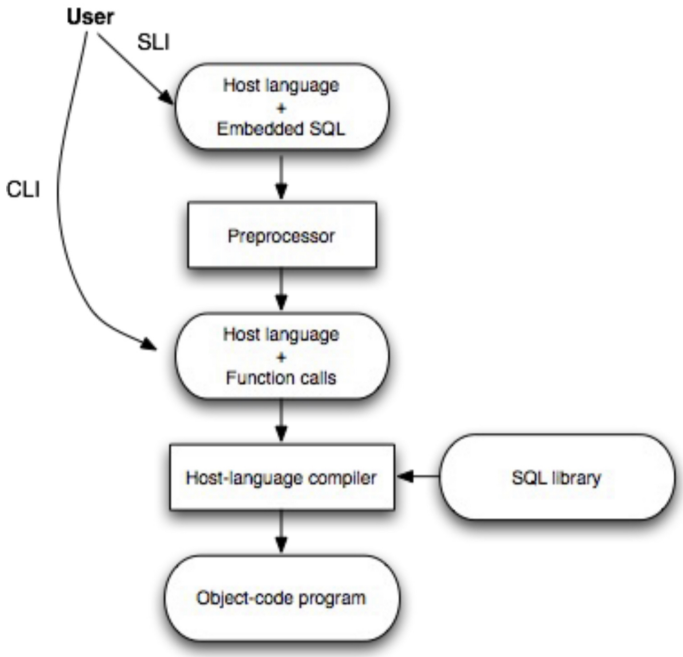
- Can solve the above problems by combining SQL with a conventional language
- However SQL is based on *relations*, while conventional languages have no such type
- Need to feed tuples from SQL to the other language one at a time, and feed each attribute value into a particular variable

Approach of Stored Procedures

- The SQL standard includes a language for defining *stored procedures*, which can
  1. have parameters and return value
  2. use local variables, ifs, loops, etc.
  3. execute SQL queries
- *Procedure*: an old technical term for a *function*
- Once defined, a stored procedure can be used in the following ways:
  - Called from the interpreter
  - Called from SQL queries
  - Called from another stored procedure
  - Be the action that a *trigger* performs
- Not a very standard approach since different DBMSs defined different proprietary languages for stored procedures
  - E.g. PL/pgSQL for PostgreSQL
- This approach is the most efficient, but code is not portable

Approach of Statement-Level Interface (SLI)

- Embed SQL statements into code in a conventional language like C or Java
- Use a preprocessor to replace the SQL with calls written in the host language to functions defined in an SQL library
- Special syntax indicates which bits of code the preprocessor needs to convert



Approach of Call-Level Interface (CLI)

- We can write the SQL calls by ourselves
- Eliminates the need to preprocess
- Each language has its own library for this (e.g. SQL/CLI for C, JDBC for Java, psycopg2 for Python)

SQL Injections

- Building up a query string and passing it to execute is vulnerable to injections
  - E.g. user input is

```
1               xyz'); DROP TABLE abc;--
```

  - The string is closed, drop table is executed, and the rest is a comment
- Use (psycopg2's) second argument of the execute method to dynamically complete a query at runtime
- Never use Python string concatenation (+) or string parameters interpolation (%) to pass variables to a SQL query string

16 Functional Dependency Theory

Want to get a schema that is in a *normal form*, which guarantees good properties

- *Normalization*: the process of converting a schema to a normal form

Example: A Poorly Designed Table

part	manufacturer	manAddress	seller	sellerAddress	price
1983	Hammers 'R Us	99 Pinecrest	ABC	1229 Bloor W	5.59
8624	Lee Valley	102 Vaughn	ABC	1229 Bloor W	23.99
9141	Hammers 'R Us	99 Pinecrest	ABC	1229 Bloor W	12.50
1983	Hammers 'R Us	99 Pinecrest	Walmart	5289 St Clair W	4.99

- Some data are redundant
  - Every part has 1 manufacturer
  - Every manufacturer has 1 address
  - Every seller has 1 address
- Redundant data can lead to anomalies

Anomalies

- *Update anomaly*: e.g. if Hammers 'R Us moves and we update only 1 tuple, the data is inconsistent
- *Deletion anomaly*: e.g. if ABC stops selling part 8624 and Lee Valley makes only that one part, we lose track of its address

Functional Dependency Definition

- Suppose R is a relation, and X and Y are subsets of the attributes of R
- $X \to Y$ asserts that if two tuples agree on all the attributes in set X, they must also agree on all the attributes in set Y
- We say that "$X \to Y$ holds in R", or "X *functionally determines* Y"
- $X \to Y$ is a *dependency* because the value of Y depends on the value of X
- $X \to Y$ is *functional* because there is a mathematical function that takes a value for X and gives a *unique* value for Y
- An FD is an assertion about *every* instance of the relation
  - Determining whether an FD holds requires domain knowledge

Formal Definition

$$A \to B : \forall \text{tuples } t_1, t_2, (t_1[A] = t_2[A]) \implies (t_1[B] = t_2[B])$$
$$\text{Equivalently } \neg\exists \text{tuples } t_1, t_2 \text{ such that } (t_1[A] = t_2[A]) \wedge (t_1[B] \neq t_2[B])$$

Generalization to Multiple Attributes

$$A_1 A_2 \cdots A_m \to B_1 B_2 \cdots B_n : \forall \text{tuples } t_1, t_2, \bigwedge_{i=1}^{m} t_1[A_i] = t_2[A_i] \implies \bigwedge_{j=1}^{n} t_1[B_j] = t_2[B_j]$$
$$\text{Equivalently } \neg\exists \text{tuples } t_1, t_2 \text{ such that } \bigwedge_{i=1}^{m} t_1[A_i] = t_2[A_i] \wedge \neg \bigwedge_{j=1}^{n} t_1[B_j] = t_2[B_j]$$

FDs in Example

- Every part has 1 manufacturer: part → manufacturer
- Every manufacturer has 1 address: manufacturer → manAddress
- Every seller has 1 address: seller → sellerAddress

Equivalent Sets of FDs

- When we write a set of FDs, we mean that *all* of them hold
- We can often rewrite sets of FDs in equivalent ways
- When we say that S_1 is equivalent to S_2, we mean that S_1 holds in a relation iff S_2 holds

Splitting Rules for FDs

- We *can* split the RHS of an FD into multiple FDs

$$AB \to CD \equiv AB \to C \wedge AB \to D$$

- We *cannot* split the LHS of an FD

FDs and Keys

- Suppose K is a set of attributes for relation R
- K is a *superkey* for R iff K functionally determines all of R
- Recall that a superkey is a set of attributes for which no two rows can have the same values
- FDs are a generalization of keys
  - Superkey: $X \to R$ where R represents every attribute
  - Functional dependency: $X \to Y$, where Y is not necessarily every attribute

Inferring FDs

- Given a set of FDs, we can often infer further FDs
- Big task: given a set of FDs, infer *every* other FD that must also hold
- Simpler task: given a set of FDs, check whether *one given* FD must also hold
- We are not generating new FDs, but testing a specific possible one
- Method 1: prove an FD follows using first principles
  - Prove by referring back to the FDs that we know hold, and the definition of functional dependency
- Method 2: prove an FD follows using the *closure test*
  - Assume we know that values of the LHS attributes, and figure out everything else that is determined
  - If it includes the RHS attributes (the ones that we want to test), then we know that LHS → RHS

```
attribute_closure(Y, S):
    # Y is a set of attributes, S is a set of FDs
    # Return the closure of Y under S
    initialize Y+ to Y
    while Y+ changes do
        if there is an FD LHS -> RHS in S such that LHS is in Y+ then
            add RHS to Y+
```

- If LHS is in Y^+ and LHS $\rightarrow$ RHS holds, we can add RHS to Y^+

```
fd_follows(S, LHS -> RHS):
    Y+ = attribute_closure(LHS, S)
    return RHS is in Y+
```

Projecting FDs

- When normalizing a schema, we would need to *decompose* relations
- We want to know what FDs hold in the decomposed smaller relations
- Can *project* our FDs onto the attributes of our new relations

```
project(S, L):
    # S is a set of FDs, L is a set of attributes
    # Return the projection of S onto L, which are all FDs that follow from S
    # and involve only attributes from L
    initialize T to {}
    for each subset X of L do
        compute X+ # close X and see what we get
        for every attribute A in X+ do
            if A in L then # X -> A only relevant if A in L
                add X -> A to T
    return T
```

  - No need to add $X \rightarrow A$ if $A = X$, it is a trivial FD
  - The following subsets of X won't yield anything, so no need to compute their closures:
    - The empty set
    - The set of all attributes
  - If we find that X^+ = all attributes, we can ignore any superset of X, as it can only give us "weaker" FDs (i.e. same FD with a larger LHS)
- Projection is expensive (exponential number of subsets)

Minimal Basis

- Given a set of FDs S, we want to find a *minimal basis*, which is a set of FDs that is equivalent, but has:
  - No redundant FDs, and
  - No FDs with unnecessary attributes on the LHS

```
minimal_basis(S):
    # S is a set of FDs
    # Return a minimal basis for S
    split the RHS of each FD
    for each FD X -> Y where |X| >= 2 do
        if we can remove an attribute from X and get an FD that follows from S then
            do so # it's a stronger FD
    for each FD f do
        if S - {f} implies f then
            remove f from S
```

- Often there are *multiple* minimal bases (depends on the order in which we consider the possible simplifications)
- After we identify a redundant FD, we must not use it when computing subsequent closures
  - One FD is likely redundant because of the presence of another FD
- When we are computing closures to decide whether the LHS of an FD $X \to Y$ can be simplified, we can continue to use that FD
- The two for-loops must be in this order
  - Otherwise the two loops need to be repeated until no changes occur

17 Database Design

Functional Dependency Use Cases

- The FDs can tell us that there are redundancy, therefore the design is bad

Decomposition

- To improve a badly-designed schema $R(A_1, \ldots, A_n)$, we can decompose it into two smaller relations

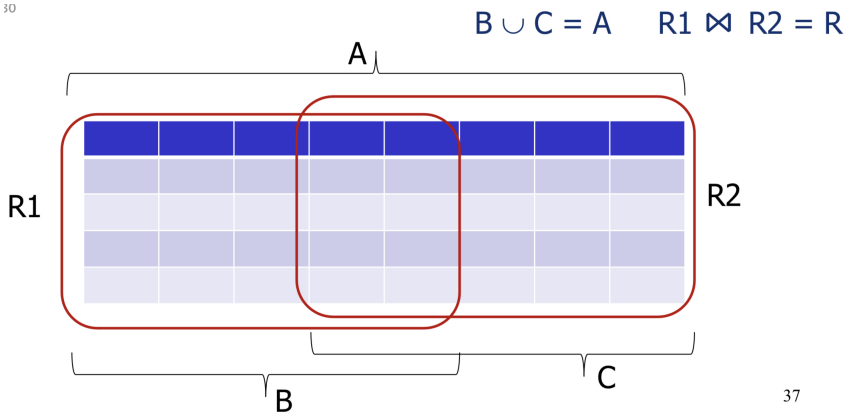
$$R_1(B_1, \ldots, B_j) = \Pi_{B_1,\ldots,B_j} R$$
$$R_2(C_1, \ldots, C_k) = \Pi_{C_1,\ldots,C_k} R$$

  such that

$$\{B_1, \ldots, B_j\} \cup \{C_1, \ldots, C_k\} = \{A_1, \ldots, A_n\}$$
$$R_1 \bowtie R_2 = R$$

  - No attributes leftout; no attributes added
  - We can reconstruct R

30



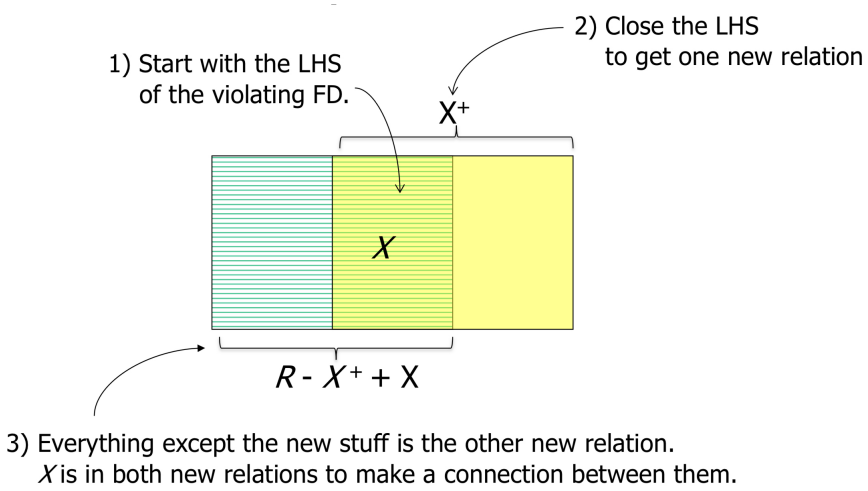
37

- Many possible decompositions for a relation
- **Boyce-Codd Normal Form** guarantees that the new schema that doesn't exhibit anomalies

Boyce-Codd Normal Form

- A relation R is **BCNF** if for every nontrivial FD $X \to Y$ that holds in R, X is a superkey
- *Nontrivial* means Y is not contained in X
- Recall that a *superkey* does not have to be minimal
- BCNF requires that only things that functionally determine *everything* can functionally determine *anything*

```
bcnf_decomp(R, F):
    # R is a relation, F is a set of FDs
    # Return the BCNF decomposition of R, given these FDs
    if an FD X -> Y in F violates BCNF then
        compute X+
        replace R by two relations with schemas:
            R1 = X+
            R2 = R - (X+ - X)
        project the FDs F onto R1 and R2
        Recursively decompose R1 and R2 into BCNF
```



- If more than 1 FD violates BCNF, then we could decompose based on any one of them
  - Multiple decompositions are possible
- The new relations we create may not be in BCNF, so we must recurse
  - We only keep the relations at the "leaves"
- We don't need to know any keys, since only superkeys matter
- We don't need to know *all* superkeys
  - Only need to check whether the LHS of each FD is a superkey
  - Can use the closure test
- When projecting FDs onto a new relation, check each new FD for whether the new relation violates BCNF because of it
  - If so, abort the projection, since we are about to discard this relation anyway (and decompose further)

Properties of Decompositions

- What we want from a decomposition:
  1. **No anomalies**
  2. **Lossless join**: it should be possible to project the original relations onto the decomposed schema, then reconstruct the original by joining and get back the original tuples
  3. **Dependency preservation**: all the original FDs should be satisfied
- BCNF decomposition satisfies the first two properties
  - The BCNF property alone does *not* guarantee lossless join
  - If we use the BCNF decomposition algorithm, then a lossless join is guaranteed
  - Since the BCNF algorithm breaks relations down too much, it is possible to create an instance that satisfies all the FDs in the final schema, but violates one of the original FDs

3rd Normal Form

- **3rd Normal Form (3NF)** modifies the BCNF condition to be less strict

- An attribute is **prime** if it is a member of any key
- $X \to A$ violates 3NF iff X is not a superkey and A is not prime

```
3NF_synthesis(F, L):
    # F is a set of FDs, L is a set of attributes
    # Synthesis and return a schema in 3rd Normal Form
    construct a minimal basis M for F
    for each FD X -> Y in M do
        define a new relation with schema X union Y
    if no relation is a superkey for L then
        add a relation whose schema is some key
```

- Comparison to BCNF
  - BCNF decomposition does not stop decomposing, until in all relations, if $X \to A$ then X is a superkey
  - 3NF generates relations where $X \to A$ and yet X is *not* a superkey, but A is at least prime
- 3NF guarantees lossless join and dependency preservation, but not no anomalies
  - 3NF allows FDs with a non-superkey on the LHS, which allows redundancy, and thus anomalies
- Decomposing too far $\implies$ cannot enforce all FDs
- Decomposing not far enough $\implies$ can have redundancy
- We consider a schema "good" if it is in either BCNF or 3NF
- Synthesis vs. decomposition
  - Synthesis: we build up the relations in the schema from nothing
  - Decomposition: we start with a bad relation schema and break it down

Testing for a Lossless Join

- We project R into $R_1, \ldots, R_k$, and attempt to recover R by joining
- We will get all of R since any tuple in R can be recovered from its projected fragments
- However we might geet a tuple that we didn't have in R, which is the part we want to check
- We would not need to test for lossless join if the schema was generated via BCNF decomposition or 3NF synthesis
- However merely satisfying BCNF/3NF does not guarantee lossless join

Chase Test

- Suppose tuple t appears in the join
- Then t is the join of projections of some tuples of R, one for each R_i of the decomposition
- Start by assuming $t = abc...$
- For each i, there is a tuple s_i of R that has $a, b, c, \ldots$ in the attributes of R_i, and s_i can have any values in other attributes
- Algorithm

1. If two rows agree in the left side of a FD, make their right sides agree too
2. Always replace a subscripted symbol by the corresopnding unsubscripted one, if possible
3. If we ever get a completedly unsubscripted row, then we know any tuple in the project-join is in the original (i.e. the join is lossless)
4. Otherwise, the final tableau is a counterexample (i.e. the join is lossy) since the subscripted symbols could represent different values

18 Entity/Relationship Model

Entity/Relationship (ER) Model

- **Modelling**: *map* the entities and relationships of the world into the concepts of a database
- Basic concepts:
  - *Entities*
  - *Relationships* among them
  - *Attributes* describing the entities and the relationships

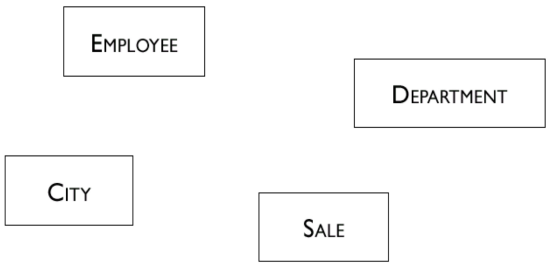
Defining a Schema

- E/R allows us to specify what the structures can/must look like
- We generalize fro specific entities and relatoinships to the sets they are drawn from

Instance	**Schema**
Entity (with attributes)	Entity Set (with attributes)
Relationship (with attributes)	Relationship Set (with attributes)

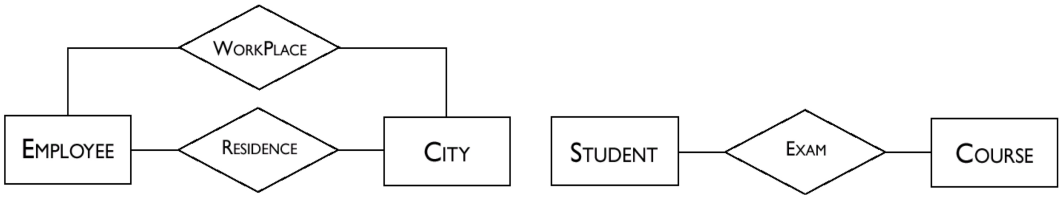
Entity Sets

- An **entity set** represents a *category* of objects that have properties in common and an autonomous existence (e.g. City, Department, Employee, Sale)
- An **entity** is an *instance* of an entity set (e.g. Toronto is a City)



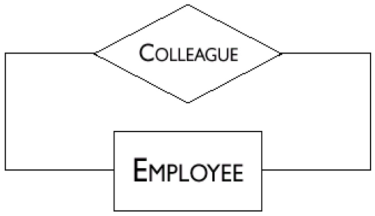
Relationship Sets

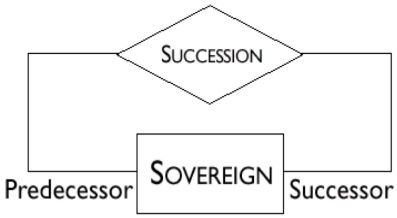
- A **relationship set** is an association between 2+ entity sets (e.g. Exam is a relationship set between entity sets Student and Course)
- A **relationship** is an instance of a n-ary relationship set (e.g. the pair ¡Diane, CSC343¿ is an instance of relationship Exam)



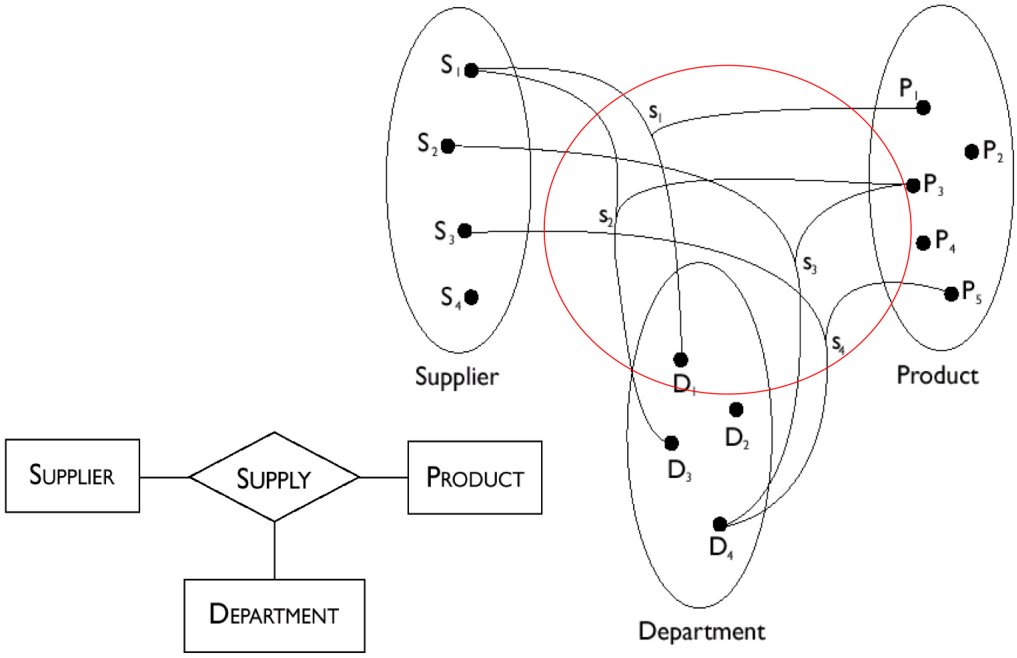
Recursive Relationships

- Recursive relatoinships relate an entity set to itself
- The relatoinship may be asymmetric
  - If so, we indicate the two *roles* that the entity plays in the relationship



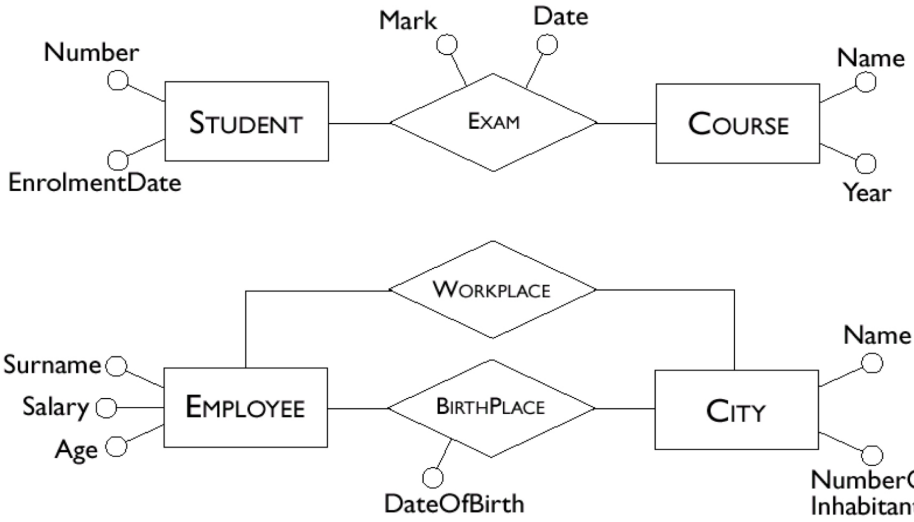


Ternary Relationships



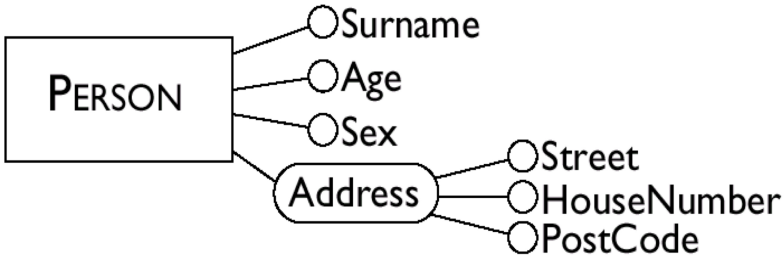
Attributes

- Describe elementary properties of entities or relationships (e.g. Surname, Salary are attributes of Employee)
- May be *single-valued*, or *multi-valued*



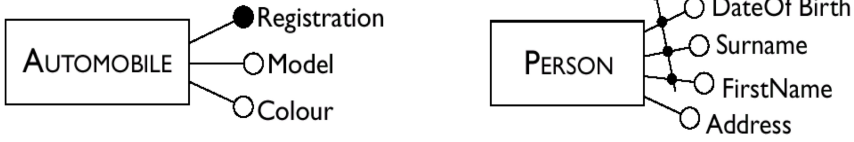
Composite Attributes

- **Composite attributes** are grouped attributes of the same entity or relationship that have closely connected meaning or uses



Keys

- Notation: solid circle
- If multi-attribute, connect with a line and a "knob"

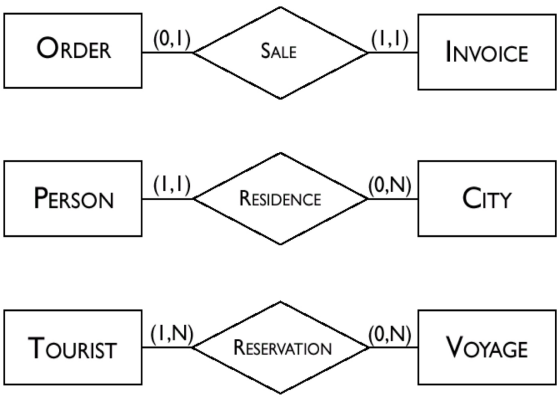


Cardinalities

- Each entity set participates in a relationship set with a *minimum* and *maximum* cardinality
- Cardinalities *constrain* how entity instances participate in relationship instances
- Notation: pairs of *(min, max)* values for each entity set

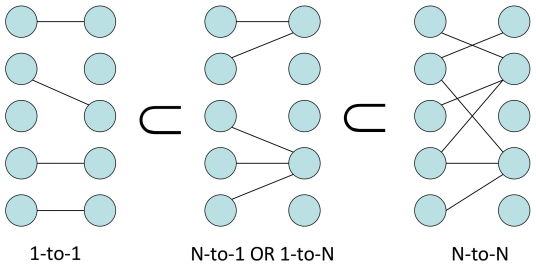
- $0 \leq \min \leq \max \in \mathbb{Z}$

- Minimum cardinality *min*
  - If 0, then entity participation in the relationship is *optional*
  - If 1, then entity participation in the relationship is *mandatory*
  - Other values are possible
- Maximum cardinality *max*
  - If 1, then each instance of the entity is associated with *at most one* instance of the relationship
  - If > 1, then each instance of the entity can be associated with *multiple* instances of the relationship
  - N indicates that there is no upper limit
  - Other values are possible



Multiplicity of Relationships

- Suppose entity sets E_1 and E_2 participate in relationship R with cardinalities (n_1, N_1) and (n_2, N_2), respectively
- Then the **multiplicity** of R is N_1-to-N_2 (or N_2-to-N_1)

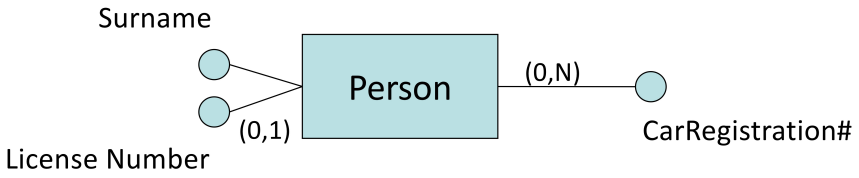


- 1-to-1 means no "branching" is allowed on either side
- 1-to-N means branching is allowed on one side
  - Must examine the ER diagram to find out which side
- N-to-N means branching is allowed on either side
- Conveys less information than the (min, max) notation since this only says what the *maxs* are

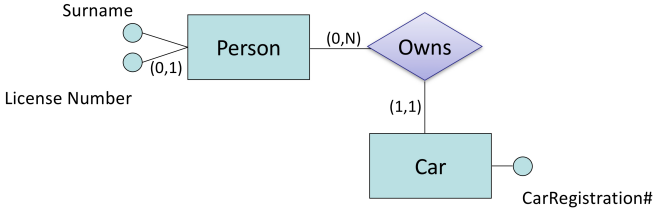
Cardinalities of Attributes

- Describe min/max number of values an attribute can have

- When the cardinality of an attribute is $(1, 1)$, this is a **single-valued attribute** and can be omitted from the diagram
- The value of an attribute may be null, or have several values (i.e. **multi-valued attribute**)

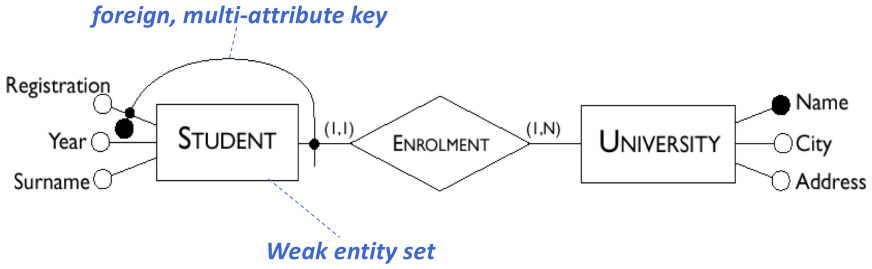


- Multi-valued attributes often represent situations that can be modelled with additional entities, e.g.



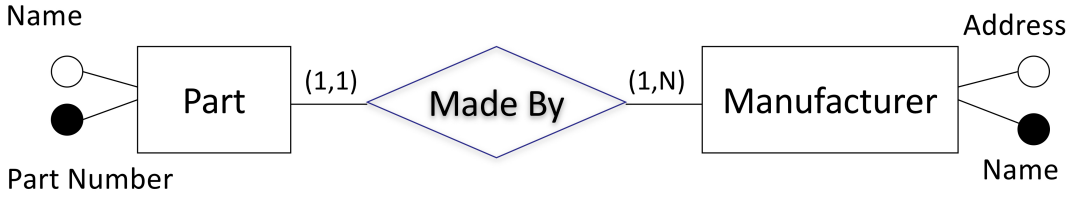
Weak Entity Sets

- An **internal key** is a key formed by one or more attributes of the entity itself
- A **weak entity set** is an entity set that doesn't have a key among its attributes
  - The keys of related entities are brought in to help with identification (i.e. becoming **foreign keys**)



Keys of Relationship Sets

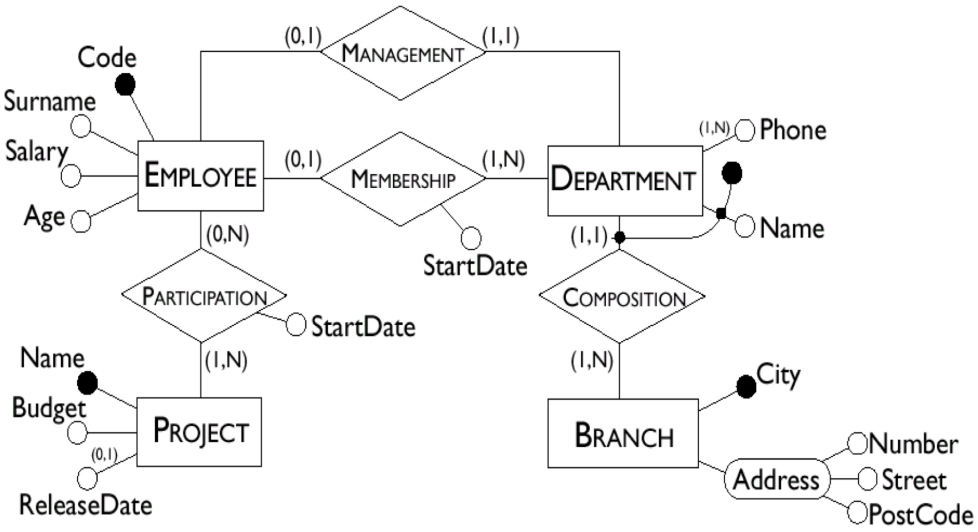
- The key for a relationship set consists of the keys of the entity sets that it relates
- E.g. the key of the *Made By* relationship set is *Part Number* and *Name*



Requirements for Keys

- Each *attribute* in a key must have $(1, 1)$ cardinality
- A foreign key for a weak entity set must come through a relationship which the entity set participates in with cardinality $(1, 1)$
- A foreign key may involve an entity that has itself a foreign key, as long as cycles are not generated

- Each entity set must have at least one (internal or foreign) key



Challenges of ER Modelling

- Design choices: shoudl a concept be modelled as an entity, an attribute, or a relationship?
- Limitations: some data semantics cannot be captured
- **Parsimony**: as complex as necessary, but no more; represent only relevant things

19 E/R Model To Database Schema

Steps

1. **Restructure** the ER schema to improve it, based on criteria
2. **Translate** the schema into the relational model
3. **Add missing constraints** to the schema

Restructuring

- Input: E/R schema
- Output: Restructured E/R schema
- This step involves:
  - Analysis of redundancies
  - Choosing entity set vs. attribute
  - Limiting the use of weak entity sets
  - Settling on keys
  - Creating entity sets to replace attributes with cardinality greater than 1

Analysis of Redundancies

- Within a relation, if one entity set has an attribute, and another entity set also has that attribute, then there is redundancy

Entity Sets vs. Attributes

- We prefer attributes
- An entity set should satisfy at least one of the following conditions:
  1. It is more than the name of something; it has at least one non-key attribute
  2. It is tha "many" in a many-one or many-many relationship
- A "thing" in its own right: entity set
- A "detail" about some other thing: attribute

Limiting the Use of Weak Entity Sets

- Use weak entity sets only when there is no global authority capable of creating unique IDs (e.g. unique student numbers all over the world)
- It is usually better to create unique IDs

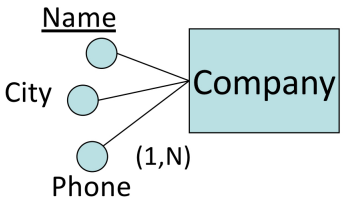
Settling on Keys

- Make sure that every entity set has a key
- Attributes with null values cannot be part of primary keys
- Internal keys are preferable to external ones
- A key that is used by many operations to access instances of an entity is preferable to others
- A key with *one/few attributes* is preferable

- An *integer* key is preferable
- Avoid multi-attribute and string keys
  - They take up more space to store than 4-byte integers
  - They break encapsulation (i.e. personal information may be leaked)
  - They are "brittle"
    - Names or phone numbers may change
    - People may not have no phone number
    - Two movies may have the same title and year

Creating Entity Sets to Replace Attributes with Cardinality Greater than 1

- The relational model doesn't allow multi-valued attributes, so we need to convert them to entity sets



- E.g.

Translationg an E/R Model into a Database Schema

- Input: E/R schema
- Output: Relational schema
- Starting from an E/R schema, an equivalent relational schema is constructed (i.e. a schema capable of representing the same information)
- A good translation should
  1. Not allow redundancy
  2. Not have unnecessary null values
- Idea
  - Each entity set becomes a relation
    - Its attributes are the attributes of the entity set
  - Each relationship becomes a relation
    - Its attributes are the keys of the entity sets that it connects (as key), and the attributes of the relationship itself

Many-to-Many Relationships

Employee(Number, Surname, Salary)

Project(Code, Name, Budget)

Participation(Number, Code, StartDate)

Participation[Number] ⊆ Employee[Number]

Participation[Code] ⊆ Project[Code]

One-to-Many Relationships

- With mandatory participation on the "one" side
  - Can "collapse" the relationship set into the entity set on the "one" side
  - Collapsing is okay iff the relation set has min-1 max-1 participation in the relationship set

One-to-One Relationships

- With mandatory participation for both
  - Can collapse the relationship set into either entity set
- With optional participation for one
  - Can collapse the relationship set into the entity set with mandatory participation

Final Considerations

- During or after the translation, we should
  - Express the foreign-key constraints
  - Consider better names for the referring attributes
  - Express the "max 1" constraints
  - Express the "min 1" constraints
- After this process, we would not have explicit functional dependencies

Redundancy can be Desirable

- Disadvantages
  - More storage
  - Additional operations to keep the data consistent
- Advantages
  - Speed: fewer accesses necessary to obtain information

- We could consider whether to allow some redundancy (i.e **denormalization**)
  - Speedup in operations made possible by the redundant information
  - Relative frequency of those operations
  - Storage needed for the redundant informations

20 Indices

Speeding Up Search

- A balanced binary search tree is good for searching
- The data in a database usually would not fit in memory, and following a file pointer is very slow
- To minimize pointer following, we could use a very large branching factor and achieve a short tree
- DBMSs usually use *B-trees* to achieve these goals

B-Tree

- A tree with branching factor $M > 2$ with the following balance rules:
  - All leaves are at the same depth
  - Every node has at least $\lceil M/2 \rceil$ children, except for possibly the root
  - The root has at least 2 children (unless it is the only node)
- For a B-tree of order M to have n values, height $\leq \log_{\lceil M/2 \rceil} n$

Storing Non-Integer Values

- The B-tree has only keys
- We navigate these to find the desired value at a leaf
- Along with the value is a pointer to the actual data
- We call the B-tree an **index** on the data

Index in SQL

```
1       create index off_dept on offering(dept);
```