Grading

- Final exam (150 points) – open book
- Midterm (100 points) – open book
  - Open book does NOT include electronic devices!

- Approximately 6 homework assignments
  - 10 points each, drop lowest score.
  - Assignments for CS630 will have additional questions
  - Assignments are individual – submit your own work only!
  - No plagiarism! See student code of conduct
Course Overview

- Relational Data Model
- Relational Algebra
- Structured Query Language
  - The most important part of the course
- Conceptual design – the ER model
- Views
- After midterm Exam:
  - Database application development
    - Java/JDBC, PL/SQL
- Design Theory
- Database Security
The Relational Model
Relational Schema

- **Schema**: specifies name of relation, plus name and domain of each column

  Students ( sid: integer,
             name: string,
             login: string,
             age: integer,
             gpa: real )

- Each relation must have a schema
  - Similar to a data type in programming languages

- Relational database schema = collection of relations’ schemas
More about Relations

- Relations are sets of tuples
  - Sets are NOT ordered
  - Do NOT retrieve by row number, but by content!

- Relation Instance
  - Contents of a relation may change over time
    - Tuples are added/deleted/modified
    - E.g., Students join or leave the university
  - Instance represents set of tuples at a certain point in time

- Schemas may change too
  - Although this is not frequent in practice
  - Changing schema is very expensive
**Keys**

- A key of a relation is a set of fields $K$ such that:
  1. No two distinct tuples in ANY relation instance have same values in all key fields, and
  2. No subset of $K$ is a key (otherwise $K$ is a superkey)

- Key may not be unique
  - Multiple candidate keys may exist
  - One of the keys is chosen (by DBA) to be the primary key

- Keys are shown underlined in schema

- In the relational model, duplicate tuples do not exist!
  - But most DBMS implementations do allow duplicates
  - Key constraints must be set by DBA to avoid duplicates
  - Rule of thumb: every table should have a primary key
Example of Keys

Students\((sid:\text{ string, name: string, login: string, age: integer, gpa: real})\)

- \(sid\) is a \textbf{key}; \\{\textit{sid, name}\} is a \textbf{superkey}

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{sid} & \text{name} & \text{login} & \text{age} & \text{gpa} \\
\hline
53666 & Jones & jones@cs & 18 & 3.4 \\
53688 & Smith & smith@eecs & 18 & 3.2 \\
53650 & Smith & smith@math & 19 & 3.8 \\
\hline
\end{array}
\]

- In practice, it is not easy to know when there exists a unique attribute combination in the data (e.g., names)
  - artificial keys are created: student ID, customer ID, etc.
  - SSN is also often used for keys (but what if multinational…)}
Relational Algebra

CS430/630
Lectures 2, 3

Slides based on “Database Management Systems” 3rd ed, Ramakrishnan and Gehrke
Relational Algebra (examples in Lecture 2)

- **Basic operations:**
  - **Selection** \( \sigma \) Selects a subset of rows from relation
  - **Projection** \( \pi \) Deletes unwanted columns from relation
  - **Cross-product** \( \times \) Allows us to combine several relations
  - **Join** \( \bowtie \) Combines several relations using conditions
  - **Division** \( \div \) A bit more complex, will cover later on
  - **Set-difference** — **Union** \( \cup \) **Intersection** \( \cap \)
  - **Renaming** \( \rho \) Helper operator, does not derive new result, just renames relations and fields
    \[ \rho(R(F), E) \]
    - F contains *oldname* \( \rightarrow \) *newname* pairs
Operator Precedence (from Lecture 3)

- In decreasing order of priority:
  1. **Selection** $\sigma$  **Projection** $\pi$
  2. **Cross-product** $\times$  **Join** $\bowtie$
  3. **Set-difference** $-\setminus$  **Intersection** $\cap$
  4. **Union** $\cup$

**Example:**

$$\sigma_{bid=103} \text{Reserves} \bowtie \text{Sailors}$$

means

$$(\sigma_{bid=103} \text{Reserves}) \bowtie \text{Sailors}$$

not

$$\sigma_{bid=103}(\text{Reserves} \bowtie \text{Sailors})$$
Sample Query 0

Find names of sailors who have ratings at least 8.

\[ \pi_{\text{sname}}(\sigma_{\text{rating} \geq 8} \text{Sailors}) \]

\[ \pi_{\text{sname}}\sigma_{\text{rating} \geq 8} \text{Sailors} \]
Sample Query 1

Find names of sailors who’ve reserved boat #103

\[ \pi_{\text{surname}}(\sigma_{\text{bid}=103}(\text{Reserves} \bowtie \text{Sailors})) \]

\[ \pi_{\text{surname}}(\sigma_{\text{bid}=103}((\text{Reserves} \bowtie \text{Sailors}))) \]
### Sample Query 2

**Sailors**

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
</table>

**Boats**

<table>
<thead>
<tr>
<th>bid</th>
<th>name</th>
<th>color</th>
</tr>
</thead>
</table>

**Reserves**

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
</table>

- **Find names of sailors who’ve reserved a red boat**
  - **Detail of sailor sid**
  - **sid, bid …**
  - **Detail of boat bid**

\[
\pi_{sname}(\pi_{sid}(\pi_{bid}(\sigma_{color='red'} B)) \bowtie R) \bowtie S)
\]

\[
\pi_{sname}(\sigma_{color='red'} Boats) \bowtie Reserves \bowtie Sailors
\]
### Sample Query 3

Find names of sailors who’ve reserved a red or a green boat

\[ \rho (\text{Tempboats}, (\sigma \color='red' \lor \color='green', \text{Boats})) \]

\[ \pi_{\text{name}}(\text{Tempboats} \bowtie \text{Reserves} \bowtie \text{Sailors}) \]

\[ \pi_{\text{name}}(\sigma \color='red' \lor \color='green', \text{B} \bowtie \text{R} \bowtie \text{S}) \]
Find names of sailors who’ve reserved a red and a green boat

\[ \rho (\text{Tempred}, \pi_{sid}((\sigma_{color='red'} \text{Boats}) \bowtie \text{Reserves}))) \]

\[ \rho (\text{Tempgreen}, \pi_{sid}((\sigma_{color='green'} \text{Boats}) \bowtie \text{Reserves}))) \]

\[ \pi_{sname}((\text{Tempred} \cap \text{Tempgreen}) \bowtie \text{Sailors}) \]
Sample Query 5

Find names of sailors who’ve reserved only red boats

\[
\rho \left( \text{Tempred, } \pi_{sname} \left( \left( \sigma_{\text{color} = 'red'} \text{Boats} \right) \bowtie \text{Reserves} \right) \right)
\]

\[
\rho \left( \text{Tempothers, } \pi_{sname} \left( \left( \sigma_{\text{color} <> 'red'} \text{Boats} \right) \bowtie \text{Reserves} \right) \right)
\]

\[
\pi_{sname} \left( \left( \text{Tempred - Tempothers} \right) \bowtie \text{Sailors} \right)
\]
Structured Query Language
Relational Query Language: SQL

- Supports simple, yet powerful querying of data.
  - Precise semantics for relational queries.
  - DML (Data Manipulation Language)
  - DDL (Data Definition Language)
- SQL developed by IBM (system R) in the 1970s
- Standards:
  - SQL-86
  - SQL-89 (minor revision)
  - SQL-92 (major revision) : third and most important standard
    - Early enough to affect Oracle, DB2, other important commercial databases, so the real common ground.
  - SQL-99 (major extensions, triggers, recursive queries)
SQL’s Most Important Data Types

- **Character strings**
  - CHAR(n), VARCHAR(n): fixed and variable-length strings
    - We will stick to ASCII characters here to avoid problems
    - But note that Unicode is supported now by Oracle, mysql, … and SQL 2003

- **Numerical:**
  - INTEGER (INT) Usually 32-bit, but not guaranteed by SQL-92.
  - Floating point: FLOAT or DOUBLE PRECISION: Usually 64-bit IEEE floating point, but not guaranteed by SQL-92.
    - Oracle: use BINARY_DOUBLE for IEEE format.
  - Fixed precision: DECIMAL(n,d)
    - 1234.56 is of type DECIMAL(6,2), precision 6, scale 2

- **DATE and TIME and TIMESTAMP**
  - Not used in R&G, see coverage pp. 262-269 in Murach
  - Timezones a challenge: SQL-92 says DB uses GMT in its data, but mysql does this only for TIMESTAMP
Creating Relations in SQL

CREATE TABLE Students
    (sid CHAR(20),
     name CHAR(20),
     login CHAR(10),
     age INTEGER,
     gpa REAL);

CREATE TABLE Enrolled
    (sid CHAR(20),
     cid CHAR(20),
     grade CHAR(2));
Destroying and Altering Relations

DROP TABLE Students;

- Deletes relation *Students*, including schema information *and* all the tuples

ALTER TABLE Students
  ADD firstYear INTEGER;

- Add new column to schema
- Every tuple is extended with **NULL** value in added field
- Default value may be specified instead of **NULL**
Structure of SQL SELECT Query in Lecture 4

- **relation-list** = list of relation names
  - possibly with a *range-variable* (AKA correlation name or table alias) after each name

- **target-list** = list of attributes of relations in **relation-list**

- **qualification** = conditions Attr \( op \) const or Attr1 \( op \) Attr2
  - \( op \) is one of \(<, >, =, \geq, \leq, <>, \), or string operators
  - Expressions connected using **AND, OR and NOT**

- **DISTINCT** = optional, eliminates duplicates
  - By default duplicates are **NOT** eliminated!

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
```
Conceptual Evaluation Strategy

- Semantics of SQL query
  1. Compute the cross-product of \textit{relation-list}
  2. Discard resulting tuples if they fail \textit{qualifications}
  3. Delete attributes that are not in \textit{target-list}
  4. If \textit{DISTINCT} is specified, eliminate duplicate rows

- This strategy is least efficient way to compute a query!
  - Optimizer finds efficient strategies to compute \textit{the same result}
Expressions and Strings

“Find rating and number of years to retirement for sailors whose names begin with ‘d’, end with ‘n’ and contain at least three characters”

```sql
SELECT S.rating, 60 - S.age AS Yr_to_retire 
FROM Sailors S 
WHERE S.sname LIKE 'd_%n'
```

- **AS** allows to (re)name fields in result.
- **LIKE** is used for string matching
  - _ stands for any one character
  - % stands for 0 or more arbitrary characters
- **Mysql** also has RLIKE, with regular expression matching
Expressions and Strings - Example

```
SELECT  S.rating, 60 - S.age AS Yr_to_retire
FROM    Sailors S
WHERE   S.sname LIKE 'd_\%n'
```

**Sailors**

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>Yr_to_retire</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>15</td>
</tr>
</tbody>
</table>
Set Operations

- **UNION**
  - compute the union of any two *union-compatible* sets of tuples

- **INTERSECT** (not supported in mysql)
  - compute the intersection of any two *union-compatible* sets of tuples

- **EXCEPT or MINUS** (not supported in mysql)
  - Set difference of any two *union-compatible* sets of tuples

- Duplicates eliminated by default!
  - UNION ALL, INTERSECT ALL, EXCEPT ALL retain duplicates
  - Contrast with non-set SQL operations
Integrity Constraints (ICs)
Basics in Lectures 4-5

- **IC**: condition that must hold for *any* instance of the database; e.g., *domain constraints*
  - Specified when schema is defined.
  - Checked when relations are modified.
- A *legal* instance satisfies all specified ICs
  - It is the DBMS’s role to enforce IC
- ICs we study
  - Primary key constraints
  - Foreign key constraints: referential integrity
  - Unique constraints
  - NOT NULL constraints
Connecting queries and subqueries
(Lecture 6)

- Depends on what the subquery returns:
  - A scalar value (1x1 table) – such a subquery can appear in a query in the same place where a constant appears, in SQL-99-compliant DBs.
  - A relation

- Where can subqueries appear?
  - Most often in **WHERE** clause of parent query
  - Also used in **FROM** clause followed by range variable
    - … FROM Sailors, (SELECT bid FROM Boats) Bids …
  - In **HAVING** clauses
    - Will discuss later on
  - Also, if scalar, in the select-list. Not covered in text, and not in Entry-level SQL-92.
Conditions involving relations

- Test that a relation satisfies some condition
  ... WHERE EXISTS (SELECT ...) - TRUE if subquery result is not empty
  ... WHERE UNIQUE (SELECT ...) - TRUE if subquery result has no duplicates*

```sql
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
  FROM Reserves R
  WHERE R.bid=103 AND S.sid=R.sid)
```

- Find names of sailors who’ve reserved boat #103

- Subquery is CORRELATED with parent query

  *Note UNIQUE (select…) is not in Entry SQL-92*
Conditions involving relations and tuples

- Typically have some sort of set operations semantics
  
  …WHERE field IN (SELECT … )
  … WHERE field op ANY (SELECT …)
  … WHERE field op ALL (SELECT …)

  ```sql
  SELECT S.sname
  FROM Sailors S
  WHERE S.age >= ALL (SELECT S1.age
                      FROM Sailors S1)
  ```

- Find names of sailors with maximum age
Subqueries in the FROM clause

```
SELECT  SQ.sname, SQ.bname
FROM ( SELECT S.sname, B.name AS bname
       FROM Sailors S, Boats B, Reserves R
     ) SQ
WHERE   SQ.bname='interlake';
```

- Find names of sailors who reserved 'interlake'
Rewriting INTERSECT Queries Using IN

Find sid’s of sailors who’ve reserved both a red and a green boat:

SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
   AND S.sid IN (SELECT S2.sid
                   FROM Sailors S2, Boats B2, Reserves R2
                   WHERE S2.sid=R2.sid AND R2.bid=B2.bid
                   AND B2.color='green')

- Similarly, EXCEPT queries re-written using NOT IN.
Nested Queries - Review

- **Nested queries returning a constant**
  - Typically constant is compared with other value in the WHERE clause
  - … WHERE field = (SELECT bid FROM …) …

- **Nested queries returning a relation**
  - in WHERE clause
    - … WHERE EXISTS|UNIQUE (SELECT bid FROM …) …
    - … WHERE field IN (SELECT bid FROM …) …
    - … WHERE field op ANY|ALL (SELECT bid FROM …) …
  - in FROM clause followed by range variable
    - … FROM Sailors, (SELECT bid FROM Boats) Bids …
Aggregate Operators (Lecture 7)

Significant extension of relational algebra

<table>
<thead>
<tr>
<th>COUNT (*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT ([DISTINCT] A)</td>
</tr>
<tr>
<td>SUM ([DISTINCT] A)</td>
</tr>
<tr>
<td>AVG ([DISTINCT] A)</td>
</tr>
<tr>
<td>MAX (A)</td>
</tr>
<tr>
<td>MIN (A)</td>
</tr>
</tbody>
</table>

A is a single column

Result is **single** value obtained by applying aggregate over all qualifying tuples

SELECT **COUNT** (*)
FROM Sailors S
Queries With GROUP BY and HAVING

\[
\begin{align*}
\text{SELECT} & \quad \text{[DISTINCT]} \quad \text{target-list} \\
\text{FROM} & \quad \text{relation-list} \\
\text{WHERE} & \quad \text{qualification} \\
\text{GROUP BY} & \quad \text{grouping-list} \\
\text{HAVING} & \quad \text{group-qualification}
\end{align*}
\]

- The **target-list** contains:
  - (i) attribute names list (column names, possibly with corr. names)
  - (ii) terms with aggregate operations (e.g., \texttt{MIN (S.age)})

- The **attribute list (i)** must be a subset of **grouping-list**
  - A **group** is a set of tuples that have the same value for all attributes in **grouping-list**
  - Each answer tuple corresponds to a **group**, so these attributes must have a single value per group.
GROUPBY Query Example

“Find age of the youngest sailor with age at least 18, for each rating with at least 2 such sailors”

```
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>horatio</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>bob</td>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>frodo</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Null Values (Lecture 8)

- Field values in a tuple may sometimes be
  - **unknown**: e.g., a rating has not been assigned, or a new column is added to the table
  - **inapplicable**: e.g., CEO has no manager, single person has no spouse
- SQL provides a special value **NULL** for such situations
  - Special operators **IS NULL**, **IS NOT NULL**
    ```sql
    SELECT * FROM Sailors WHERE rating IS NOT NULL
    ```
  - Note: **NULL** must not be used as constant in expressions!
  - A field can be declared as **NOT NULL**, means NULL values are not allowed (by default, PK fields are NOT NULL)
Dealing with Null Values

The presence of **NULL** complicates some issues

- **NULL op value** has as result **NULL** (op is +, -, *, /)
- What does `rating > 8` evaluate to if `rating` is equal to **NULL**?
- Answer: **unknown**

3-valued logic: true, false and **unknown**

- Recall that WHERE eliminates rows that don’t evaluate to true
- What about **AND**, **OR** and **NOT** connectives?
  - `unknown AND true` = **unknown**
  - `unknown OR false` = **unknown**
  - **NOT unknown** = **unknown**
- Also, `<NULL_value> = <NULL_value>` is **unknown**!
“Find sailors who’ve reserved all red boats”

\[ \rho(\text{Temp},(\pi_{\text{sid,bid}}\text{Reserves})/(\pi_{\text{bid}}(\sigma_{\text{color='red'}}\text{Boats}))) \]

\[ \pi_{\text{sname}}(\text{Temp} \bowtie \text{Sailors}) \]
“Find sailors who’ve reserved all boats.”

With **EXCEPT** (use **MINUS** in Oracle):

```sql
SELECT  S.sname
FROM    Sailors S
WHERE   NOT EXISTS
        (  -- look for bid(s) unconnected via R to this sid
            (SELECT  B.bid FROM  Boats B)
        EXCEPT  -- ones connected via R to S.sid
            (SELECT  R.bid FROM  Reserves R
             WHERE  R.sid=S.sid)
        )
```

Note: the parentheses around the two inner SELECTs are not needed:

“subquery EXCEPT subquery” qualifies as a subquery, and the two are surrounded by parentheses needed by exists (…)

Join Expressions

- SQL shorthands for expressions we already saw

  Cross Product:
  
  **Sailors** CROSS JOIN **Reserves**

  Condition Join:
  
  **Sailors** JOIN **Reserves** **ON** *<condition>*

  Natural Join: uses same-named columns for join columns
  
  **Sailors** NATURAL JOIN **Reserves**

Usage Example:

SELECT *
FROM **Sailors** JOIN **Reserves** **ON** Sailors.sid=Reserves.sid
The explicit syntax for an outer join

```
SELECT select_list  
FROM table_1       
{LEFT|RIGHT|FULL} [OUTER] JOIN table_2  
  ON join_condition_1  
[ {LEFT|RIGHT|FULL} [OUTER] JOIN table_3  
  ON join_condition_2]...
```

What outer joins do

<table>
<thead>
<tr>
<th>Join</th>
<th>Keeps unmatched rows from</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>The left table</td>
</tr>
<tr>
<td>Right</td>
<td>The right table</td>
</tr>
<tr>
<td>Full</td>
<td>Both tables</td>
</tr>
</tbody>
</table>
A left outer join

```sql
SELECT department_name AS dept_name,  
    d.department_number AS dept_no,  
    last_name  
FROM departments d  
    LEFT JOIN employees e  
    ON d.department_number = e.department_number  
ORDER BY department_name
```

- This join preserves departments even if they have no employees
- It would still be useful even if we had a non-null FK in place.
A SELECT statement with an outer and inner join

SELECT department_name AS dept_name,  
       last_name, project_number  
FROM departments dpt 
JOIN employees emp 
    ON dpt.department_number = emp.department_number 
LEFT JOIN projects prj 
    ON emp.employee_id = prj.employee_id 
ORDER BY department_name 

The result set

<table>
<thead>
<tr>
<th>DEPT_NAME</th>
<th>LAST_NAME</th>
<th>PROJECT_NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting</td>
<td>Hernandez</td>
<td>P1011</td>
</tr>
<tr>
<td>Maintenance</td>
<td>Hardy</td>
<td>(null)</td>
</tr>
<tr>
<td>Payroll</td>
<td>Simonian</td>
<td>P1012</td>
</tr>
<tr>
<td>Payroll</td>
<td>Smith</td>
<td>P1012</td>
</tr>
<tr>
<td>Payroll</td>
<td>Aaronsen</td>
<td>P1012</td>
</tr>
<tr>
<td>Personnel</td>
<td>Jones</td>
<td>(null)</td>
</tr>
<tr>
<td>Personnel</td>
<td>O'Leary</td>
<td>P1011</td>
</tr>
</tbody>
</table>

Here we’re saying we are only interested in employees in known departments, but want to include employees with no projects.

(7 rows selected)
SQL/92 and SQL:1999 support all options on deletes and updates.

- Default is **NO ACTION** *(delete/update is rejected)*
- **CASCADE** *(delete/update all tuples that refer to deleted/updated tuple)*
- **SET NULL / SET DEFAULT** *(sets foreign key value of referencing tuple)*
Conceptual Design.
The Entity-Relationship (ER) Model
Database Design Overview

- **Conceptual design**
  - The Entity-Relationship (ER) Model, UML
  - High-level, close to human thinking
  - Semantic model, intuitive, rich constructs
    - Not directly implementable

- **Logical Design**
  - The relational data model
  - Machine-implementable, fewer and more basic constructs
  - Logical design translates ER into relational model (SQL)

- **Physical Design** (not in this course)
  - Storage and indexing details
Relationship Set Representation

Representation Convention:
- Entity sets: rectangles
- Attributes: ovals, with key attributes underlined
- Relationship sets: diamonds
- Edges connect relationship sets to entity sets, and relationship sets to relationship set attributes
- Other notations are also in use: Murach uses “crow’s foot” notation
Example 2

- **Manages relationship:** each dept has *at most one* manager

  one-to-many

  from *Employees* to *Departments*, or

  *many-to-one*

  from *Departments* to *Employees*

Arrow for “to-one” direction of key constraint
Participation Constraints

Total vs Partial Participation

Total: every department must have a manager
- “Departments” entity set has total participation in relationship
- Represented as thickened line (there is a key constraint as well, represented by the arrow)

Partial: not every employee is a manager
- “Employees” entity set has partial participation, thin line

Diagram:
- Employees
  - ssn
  - name
  - lot
- Manages
- Departments
  - dname
  - budget
  - did
  - since
Relationship Sets to Tables:

CREATE TABLE Works_In(
    ssn  CHAR(11),
    did  INTEGER,
    since  DATE,
    PRIMARY KEY (ssn, did),
    FOREIGN KEY (ssn)
      REFERENCES Employees,
    FOREIGN KEY (did)
      REFERENCES Departments)

This is definitely the way to do it if the relationship is many-to-many.

If we know it’s many-to-one or one-to-many, there may be another way…
“Each department has at most one manager” key constraint on Manages

- This means Manages is many-to-one (Departments to Employees) and deserves an arrow from Departments to Manages:

- It isn’t a thick arrow because a department may have no manager at all (this is assumed unless explicitly stated otherwise)
Variant 1 for “Each department has at most one manager” key constraint on Manages

- Map relationship to its own table:
  - Note that `did` is the key now!
    - That means we can’t have two rows here for one department, pointing to two managers for that department.
  - Note this allows a department without a manager: it would simply be missing from this table.

```sql
CREATE TABLE Manages(
    ssn CHAR(11),
    did INTEGER,
    since DATE,
    PRIMARY KEY (did),
    FOREIGN KEY (ssn) REFERENCES Employees,
    FOREIGN KEY (did) REFERENCES Departments)
```
Variant 2 for “Each department has at most one manager” key constraint on Manages

- This is the “thin arrow” case in a binary relationship
- Since each department has at most one manager, we could instead combine Manages and Departments.
- And use a nullable FK on ssn.
- A department without a manager would have a null ssn

```sql
CREATE TABLE Dept_Mgr(
  did INTEGER,
  dname CHAR(20),
  budget INTEGER,
  ssn CHAR(11), -- note nullable
  since DATE,
  PRIMARY KEY (did),
  FOREIGN KEY (ssn) REFERENCES Employees)
```
Variant 2 for “Each department has exactly one manager” key constraint on Manages

- This is the “thick arrow” case in a binary relationship
- We can again combine Manages and Departments.
- And use a not-null FK on ssn.
- Variant 1 can’t implement the total participation constraint, so this is the preferred schema

```sql
CREATE TABLE Dept_Mgr(
    did INTEGER,
    dname CHAR(20),
    budget INTEGER,
    ssn CHAR(11) not null,  -- note “not null” here
    since DATE,
    PRIMARY KEY (did),
    FOREIGN KEY (ssn) REFERENCES Employees)
```
Views
Creating a view

**Defining Query** (also referred to as View Subquery)

```
CREATE VIEW RegionalSales (category, sales, state, sid)
AS SELECT P.category, S.sales, L.state, S.id
FROM Products P, Sales S, Locations L
WHERE P.pid=S.pid AND S.locid=L.locid
```
Querying views

Querying Views

```
SELECT R.category, R.state, SUM(R.sales)
FROM RegionalSales R GROUP BY R.category, R.state
```

- Views are queried just like regular tables
  - A view is just another relation (albeit a virtual one)
  - Queries can involve both views and base tables
  - Helps to think of views in terms of analogy with window on data
Views as subqueries

Equivalent Query (without views)

```
SELECT R.category, R.state, SUM(R.sales)
FROM (SELECT P.category, S.sales, L.state, S.id
     FROM Products P, Sales S, Locations L
     WHERE P.pid=S.pid AND S.locid=L.locid) R
GROUP BY R.category, R.state
```