What Is a DBMS?

- A very large, integrated collection of data.
- Models real-world enterprise.
  - Entities (e.g., students, courses)
  - Relationships (e.g., Madonna is taking CS564)
- A Database Management System (DBMS) is a software package designed to store and manage databases.
Files vs. DBMS

- (Large) files are cumbersome to manipulate and sometimes hard to extract useful information from. Special code for different queries
- Must protect data from inconsistency due to multiple concurrent users
- Crash recovery
- Security and access control
Why Use a DBMS?

- Data independence and efficient access.
- Reduced application development time.
- Data integrity and security.
- Uniform data administration.
- Concurrent access, recovery from crashes.
Why Study Databases??

“Groundbreaking. Not only is it fun to read, it may just change the way you think.”
STEVEN D. LEVITT, co-author of Freakonomics

Why Thinking-By-Numbers Is the New Way to Be Smart

SUPER CRUNCHERS

IAN AYRES

Harvard Business Review

Getting Control of Big Data

How vast new streams of information are changing the art of management.

PAGE 39
Why Study Databases??

- “I believe that their adoption in business, government, and education has enabled the progress we’ve made in productivity, and just made things generally better. I believe that if the relational databases were to stop right now, you would be stuck in San Diego until they got going again”.

  Bruce Lindsey, IBM Fellow

- “Between the dawn of civilization and 2003, we only created five exabytes of information; now we’re creating that amount every two days”.

  Eric Schmidt, Google
What is a Database

- A *database* is an integrated and organized collection of data
- In particular *Relational Database Management Systems (RDBMS)* allow to use one set of data in multiple ways, including ways that may be unforeseen at the time the database is built and the first applications are written.
Data Models

- A data model is a collection of concepts for describing data.
- A schema is a description of a particular collection of data, using a given data model.
- The relational model of data is the most widely used model today.
  - Main concept: relation, basically a table with rows and columns.
  - Every relation has a schema, which describes the columns, or fields.
Example: Patients Database

- Conceptual schema:
  - Patients(pid: string, name: string, email: string, age: integer, risk: real)
  - Unit(uid: string, uname: string, beds: integer)
  - Admitted(pid: string, uid: string, temperature: real)
Overview of Database Design

- **Requirements analysis**
  - What are the *entities* and *relationships* in the enterprise?
  - What information about these entities and relationships should we store in the database?
  - What are the *integrity constraints* or *business rules* that hold?
Overview of Database Design

- **Conceptual design**: develop a high-level description of data stored in database, and the constraint over the data
  - *ER Model* is used at this stage
  - Often represented pictorially: *ER diagrams (ER graph)*
Overview of Database Design

- **Logical design**: convert the conceptual design into a database schema of chosen DBMS
  - Consider only relational DBMSs
  - Map an ER diagram into a relational schema
  - Resulting logical schema
ER Model Basics

- **Entity**: Real-world object distinguishable from other objects. An entity is described (in DB) using a set of **attributes**.

- **Entity Set**: A collection of similar entities. E.g., all doctors
  - All entities in an entity set have the same set of attributes.
  - Each attribute has a **domain**
  - Each entity set has a **key**
ER Model Basics (Contd.)

- **Relationship**: Association among two or more entities. E.g., Attishoo works in Pharmacy department.

- **Relationship Set**: Collection of similar relationships
  - A set of n-tuples, each n-tuple denotes an n-ary relationship
  - Same entity set could participate in different relationship sets, or in different “roles” in same set
  - Descriptive attributes

Extra Note:
- **Diagram**:
  - **Doctors** entity with attributes: doc_id, name, specialty.
  - **Departments** entity with attributes: did, dname, budget.
  - **Works_In** relationship with attributes: since.
  - **Reports_To** relationship with attributes: doc_id, specialty, name, supervisor, subordinate.
Key Constraints

- Consider Works_In: A doctor can work in many departments; a dept can have many employees.
- In contrast, each dept has at most one manager, according to the *key constraint* on Manages.
- Departments are keys: given a Department entity, we can uniquely determine the Manager relationship, i.e., each department appear once in the Manages table.
- One-to-many: one doctor can manage many departments, but each department can have only one manager.
Types of Relationships

1-to-1

1-to Many

Many-to-1

Many-to-Many
Summary of Conceptual Design

- Conceptual design follows requirements analysis,
  - Yields a high-level description of data to be stored
- ER model popular for conceptual design
  - Constructs are expressive, close to the way people think about their applications.
- Basic constructs: entities, relationships, and attributes (of entities and relationships).

- Some additional constructs: weak entities, ISA hierarchies, and aggregation.
- Note: There are many variations on ER model.
Relational Database: Definitions

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

<table>
<thead>
<tr>
<th>pid</th>
<th>name</th>
<th>email</th>
<th>age</th>
<th>risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

- Cardinality = 3, degree = 5, all rows distinct
- Do all columns in a relation instance have to be distinct?
Relational Query Languages

- A major strength of the relational model: supports simple, powerful *querying* of data.
- Queries can be written intuitively, and the DBMS is responsible for efficient evaluation.
  - The key: precise semantics for relational queries.
  - Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change.

```sql
SELECT P.name, A.uid, A.temperature
FROM Patient P, Admitted A
WHERE P.pid=A.pid AND P.age >18 AND A.uid=‘U231’
```
Creating Relations in SQL

- Creates the Patients relation.
  Observe that the type (domain) of each field is specified, and enforced by the DBMS whenever tuples are added or modified.
- As another example, the Admitted table holds information about units in which patients are being admitted.

```sql
CREATE TABLE Patients (pid TEXT, name TEXT, email TEXT, age INT, risk REAL)
CREATE TABLE Admitted (pid TEXT, uid TEXT, temp REAL)
```
Destroying and Altering Relations

DROP TABLE Patients

- Destroys the relation Patients. The schema information and the tuples are deleted.

ALTER TABLE Patients
    ADD COLUMN dob: date

- The schema of Patients is altered by adding a new field; every tuple in the current instance is extended with a null value in the new field.
Adding and Deleting Tuples

- Can insert a single tuple using:
  
  ```
  INSERT INTO Patients (pid, name, email, age, risk)
  VALUES (53699, 'Green ', 'green@ee', 18, 3.5)
  ```

- More inserts:
  
  ```
  INSERT INTO Patients (pid, name, email, age, risk)
  VALUES (53666, 'Jones', 'jones@cs', 18, 3.4)
  INSERT INTO Patients (pid, name, email, age, risk)
  VALUES (53688, 'Smith ', 'smith@eecs', 18, 3.2)
  INSERT INTO Patients (pid, name, email, age, risk)
  VALUES (53650, 'Smith ', 'smith@math', 19, 3.8)
  ```
Adding and Deleting Tuples
(continued)

- Patients relation after inserts:

<table>
<thead>
<tr>
<th>pid</th>
<th>name</th>
<th>email</th>
<th>age</th>
<th>risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eeecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
<tr>
<td>53600</td>
<td>Green</td>
<td>green@ee</td>
<td>18</td>
<td>3.5</td>
</tr>
</tbody>
</table>
Adding and Deleting Tuples (continued)

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

```
DELETE
FROM Patients P
WHERE P.name = 'Smith'
```

- **Patients** instance after delete:

<table>
<thead>
<tr>
<th>pid</th>
<th>name</th>
<th>email</th>
<th>age</th>
<th>risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53600</td>
<td>Green</td>
<td>green@ee</td>
<td>18</td>
<td>3.5</td>
</tr>
</tbody>
</table>
The SQL Query Language

- To find all 18 year old patients, we can write:

```sql
SELECT * FROM Patients P WHERE P.age=18
```

<table>
<thead>
<tr>
<th>pid</th>
<th>name</th>
<th>email</th>
<th>age</th>
<th>risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
</tr>
</tbody>
</table>

- To find just names and emails, replace the first line:

```sql
SELECT P.name, P.email FROM Patients P
```
Adding and Deleting Tuples (continued)

- Insert tuples into the Admitted instance:

  ```sql
  INSERT INTO Admitted (pid, uid, temp)
  VALUES ('53831', '101', 36)
  INSERT INTO Admitted (pid, uid, temp)
  VALUES ('53831', '203', 37)
  INSERT INTO Admitted (pid, uid, temp)
  VALUES ('53650', '112', 39)
  INSERT INTO Admitted (pid, uid, temp)
  VALUES ('53666', '105', 40)
  ```
Querying Multiple Relations

What does the following query compute?

```
SELECT P.name, A.uid
FROM Patients P, Admitted A
WHERE P.pid=A.pid AND A.temp=40
```

Given the following instance of Admitted:

<table>
<thead>
<tr>
<th>pid</th>
<th>uid</th>
<th>temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>101</td>
<td>36</td>
</tr>
<tr>
<td>53831</td>
<td>203</td>
<td>37</td>
</tr>
<tr>
<td>53650</td>
<td>112</td>
<td>39</td>
</tr>
<tr>
<td>53666</td>
<td>105</td>
<td>40</td>
</tr>
</tbody>
</table>

we get:

<table>
<thead>
<tr>
<th>P.name</th>
<th>A.uid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>105</td>
</tr>
</tbody>
</table>
Primary and Candidate Keys in SQL (continued)

- For Patients relation with PID as the primary key

  CREATE TABLE Patients
  (pid TEXT, name TEXT,
   email TEXT, age INT,
   risk REAL, PRIMARY KEY (pid) )

- Are there any separate fields or combinations of fields which also are candidates for primary key?
  - How about email?
  - How about age?
  - How about age & risk?
Foreign Keys, Referential Integrity

- **Foreign key**: Set of fields in one relation that is used to `refer` to a tuple in another relation. (Must correspond to primary key of the second relation.)

- E.g. *pid* is a foreign key referring to Patients:
  - Admitted (*pid*: string, *uid*: string, *temp*: real)
  - If all foreign key constraints are enforced, referential integrity is achieved, i.e., no dangling references.
Foreign Keys in SQL

- Only patients listed in the Patients relation should be allowed to be admitted to units.

```sql
CREATE TABLE Admitted
    (pid CHAR(20), uid CHAR(20), temp REAL,
     PRIMARY KEY (pid,uid),
     FOREIGN KEY (pid) REFERENCES Patients )
```

<table>
<thead>
<tr>
<th>Admitted</th>
<th>Patients</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>name</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
</tr>
<tr>
<td>53666</td>
<td>Smith</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
</tr>
<tr>
<td>53668</td>
<td></td>
</tr>
</tbody>
</table>
Referential Integrity in SQL

- SQL supports all 4 options on deletes and updates.
  - Default is **NO ACTION** *(delete/update is rejected)*
  - **CASCADE** (also delete all tuples that refer to deleted tuple)
  - **SET NULL / SET DEFAULT** *(sets foreign key value of referencing tuple)*

```sql
CREATE TABLE Admitted
(pid TEXT,
uid TEXT,
temp REAL,
PRIMARY KEY (pid,uid),
FOREIGN KEY (pid)
REFERENCES Patients
ON DELETE CASCADE
ON UPDATE SET DEFAULT)
```
SQL: Queries
We will use these instances of the Doctors and Reserves (e.g. beds) relations in our examples.

If the key for the Reserves relation contained only the attributes \textit{sid} and \textit{bid}, how would the semantics differ?
Basic SQL Query

- **relation-list** A list of relation names (possibly with a range-variable after each name).
- **target-list** A list of attributes of relations in relation-list
- **qualification** Comparisons (Attr \( op \) const or Attr1 \( op \) Attr2, where \( op \) is one of \(<, >, =, \leq, \geq, \neq\) combined using AND, OR and NOT.
- DISTINCT is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are **not** eliminated!
Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of relation-list.
  - Discard resulting tuples if they fail qualifications.
  - Delete attributes that are not in target-list.
  - If DISTINCT is specified, eliminate duplicate rows.
Example of Conceptual Evaluation

```sql
SELECT S.sname
FROM Doctors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103
```

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Find the sid of doctors who’ve reserved a single bed

```
SELECT R.sid
FROM Beds B, Reserves R
WHERE R.bid = B.bid AND B.size = 'single'
```

- Query contains a join of two tables (cross product, selection, projection), followed by a selection on the size of beds
- If we wanted the name of the doctors, we must include the Doctors relation as well
Find the name of doctors who’ve reserved a single bed

SELECT  S.sname
FROM    Doctors S, Beds B, Reserves R
WHERE   S.sid=R.sid AND R.bid=B.bid AND B.size = ‘single’

- Query contains a join of three tables, followed by a selection on the size of beds
Find doctors who’ve reserved at least one bed

```
SELECT S.sid
FROM Doctors S, Reserves R
WHERE S.sid = R.sid
```

- Query contains a join of two tables
- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing $S.sid$ by $S.sname$ in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?
Expressions and Strings

```
SELECT  S.age, S.age-5 as age1, 2*S.age AS age2
FROM    Doctors S
WHERE   S.sname LIKE 'B_%';
```

- Illustrates use of arithmetic expressions and string pattern matching: *Find triples (of ages of doctors and two fields defined by expressions) for doctors whose names begin with B and contain at least two characters.*
- **AS** is a way to name fields in the result.
- **LIKE** is used for string matching. ‘’ stands for any one character and ‘%’ stands for 0 or more arbitrary characters.
Find sid’s of doctors who’ve reserved a single or a double bed

- **UNION**: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).

- If we replace **OR** by **AND** in the first version, what do we get? (intersection)

- Also available: **EXCEPT** (What do we get if we replace **UNION** by **EXCEPT**?)

```sql
SELECT S.sid
FROM Doctors S, Beds B, Reserves R
  AND (B.size = 'single' OR B.size= 'double')
UNION
SELECT S.sid
FROM Doctors S, Beds B, Reserves R
  AND B.size= 'single'
UNION
SELECT S.sid
FROM Doctors S, Beds B, Reserves R
  AND B.size= 'double'
```
Find sid’s of doctors who’ve reserved a single and a double bed

- **INTERSECT**: Can be used to compute the intersection of any two union-compatible sets of tuples.
- Included in the SQL/92 standard, but some systems don’t support it.
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

```sql
SELECT S.sid
FROM Doctors S, Beds B1, Reserves R1, Beds B2, Reserves R2
WHERE S.sid=R1.sid AND R1.bid=B1.bid
AND S.sid=R2.sid AND R2.bid=B2.bid
AND (B1.size=‘single’ AND B2.size=‘double’)
```

Key field!

```sql
SELECT S.sid
FROM Doctors S, Beds B, Reserves R
AND B.size=‘single’
INTERSECT
SELECT S.sid
FROM Doctors S, Beds B, Reserves R
AND B.size=‘double’
```
Nested Queries

Find names of doctors who ’ve reserved bed #103:

```
SELECT S.sname
FROM Doctors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid=103)
```

- A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
- To find doctors who’ ve not reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a nested loops evaluation: For each Doctors tuple, check the qualification by computing the subquery.
Nested Queries with Correlation

Find names of doctors who’ve reserved bed #2:

\[
\text{SELECT S.sname}
\text{FROM Doctors S}
\text{WHERE EXISTS (SELECT *}
\text{FROM Doctors R}
\text{WHERE R.bid=2 AND S.sid=R.sid)}
\]

- EXISTS is another set comparison operator, like IN.
- Allows test whether a set is nonempty
Rewriting INTERSECT Queries Using IN

Find sid’s of doctors who’ve reserved both a single and a double bed:

SELECT S.sid
FROM Doctors S, Beds B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.size = 'single'
    AND S.sid IN (SELECT S2.sid
                   FROM Doctors S2, Beds B2, Reserves R2
                   WHERE S2.sid=R2.sid AND R2.bid=B2.bid
                        AND B2.size = 'double')

✓ Similarly, EXCEPT queries re-written using NOT IN.
Aggregate Operators

SELECT COUNT (*)
FROM Doctors S

SELECT AVG (S.age)
FROM Doctors S
WHERE S.rating = 10

SELECT COUNT (DISTINCT S.rating)
FROM Doctors S
WHERE S.sname = 'Bob'

SELECT COUNT ( [DISTINCT] A)
SUM ( [DISTINCT] A)
AVG ( [DISTINCT] A)
MAX (A)
MIN (A)

SELECT S.sname
FROM Doctors S
WHERE S.rating = (SELECT MAX(S2.rating)
FROM Doctors S2)

SELECT AVG (DISTINCT S.age)
FROM Doctors S
WHERE S.rating = 10
Find name and age of the oldest doctor(s)

- The first query is illegal!
  
  ```sql
  SELECT S.sname, MAX (S.age)
  FROM Doctors S
  ```

- The second is the correct query

  ```sql
  SELECT S.sname, S.age
  FROM Doctors S
  WHERE S.age =
  (SELECT MAX (S2.age)
  FROM Doctors S2)
  ```
GROUP BY and HAVING

- So far, we’ve applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several groups of tuples.

- Consider: *Find the age of the youngest doctor for each rating level.*
  - In general, we don’t know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

    For $i = 1, 2, \ldots, 10$:
    
    ```sql
    SELECT MIN(S.age) 
    FROM Doctors S 
    WHERE S.rating = i
    ```
Queries With GROUP BY and HAVING

```
SELECT        [DISTINCT] target-list
FROM           relation-list
WHERE          qualification
GROUP BY       grouping-list
HAVING         group-qualification
```

- The *target-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (S.age)).
  - The attribute list (i) must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list.*)
Example

<table>
<thead>
<tr>
<th>store</th>
<th>product</th>
<th>date</th>
<th>sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>150</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>250</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>100</td>
</tr>
</tbody>
</table>
Example

- Select store, product, sum(sale) from R group by store, product
  - 1 1 25
  - 1 2 45
  - 1 3 15
  - 2 1 250
  - 2 2 450
  - 2 3 150

- Select store, sum(sale) from R group by store, product

- Select store, product, sum(sale) from R group by store
Find the age of the youngest doctor with age >= 18, for each rating with at least 2 such doctors.

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

- Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes `unnecessary`.
- 2nd column of result is unnamed. (Use AS to name it.)

<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>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</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>age</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Answer relation
Find the age of the youngest doctor with age > 18, for each rating with at least 2 doctors (of any age)

SELECT S.rating, MIN(S.age) 
FROM Doctors S 
WHERE S.age > 18 
GROUP BY S.rating 
HAVING 1 < (SELECT COUNT(*) 
            FROM Doctors S2 
            WHERE S.rating=S2.rating)

- Shows HAVING clause can also contain a subquery.
- Compare this with the query where we considered only ratings with 2 sailors over 18!
- What if HAVING clause is replaced by:
  - HAVING COUNT(*) >1
Null Values

- Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable* (e.g., no spouse’s name).
  - SQL provides a special value `null` for such situations.
- The presence of `null` complicates many issues. E.g.:
  - Special operators needed to check if value is/is not `null`.
  - Is `rating>8` true or false when `rating` is equal to `null`? What about AND, OR and NOT connectives?
  - We need a 3-valued logic [true, false and unknown].
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don’t evaluate to true.)
  - New operators (in particular, outer joins) possible/needed.
Readings

  - Chapters 1, 2, 3, 5

  - Chapters 1, 2, 3, 4, 5.
  - This book uses different ER notation and has an excellent description of the rules to convert ER diagrams to the relational model.