Relational Model, Relational Algebra, & SQL
Today

how we model & represent our data
  relational model

how we evaluate queries on our data
  relational algebra

what queries we can ask on our data
  SQL
RELATIONAL MODEL

RELATIONAL ALGEBRA

SQL
Why Study the Relational Model?

Simple yet expressive

Widely used
   Vendors: IBM, Microsoft, Oracle, etc.

More recently: key-value stores
Relational Database: Definitions

Relational database: a set of relations.

Relation: made up of 2 parts:

- **Schema**: specifies name of relation, plus name and type of each column.
  
  E.g. Students(\textit{sid}: string, \textit{name}: string, \textit{login}: string, \textit{age}: integer, \textit{gpa}: real)

- **Instance**: a table, with rows and columns.
  
  \#rows = \textit{cardinality}
  
  \#fields = \textit{degree} / \textit{arity}

Can think of a relation as a set of rows or tuples.

i.e., all rows are distinct, no order among rows.
Example: *Students* relation

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

Cardinality = 3, arity = 5, all rows distinct.

Do all values in each column of a relation instance have to be distinct?
Keys

Keys are a way to associate tuples in different relations

Keys are one form of integrity constraint (IC)

---

<table>
<thead>
<tr>
<th>Enrolled</th>
<th>Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>cid</td>
</tr>
<tr>
<td>53666</td>
<td>15-101</td>
</tr>
<tr>
<td>53666</td>
<td>18-203</td>
</tr>
<tr>
<td>53650</td>
<td>15-112</td>
</tr>
<tr>
<td>53666</td>
<td>15-105</td>
</tr>
</tbody>
</table>

FOREIGN Key

PRIMARY Key
Primary Keys

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

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

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

Examples:
- *sid* is a key for Students.
- What about *name*?
- The set \{*sid*, *gpa*\} is a superkey.
Enforcing Referential Integrity

Consider Students and Enrolled; \textit{sid} in Enrolled is a foreign key that references Students.

What should be done if an Enrolled tuple with a non-existent student id is inserted? (\textit{Reject it!})

What should be done if a Students tuple is deleted?

- Also delete all Enrolled tuples that refer to it?
- Disallow deletion of a Students tuple that is referred to?
- Set sid in Enrolled tuples that refer to it to a \textit{default sid}?

(In SQL, also: Set sid in Enrolled tuples that refer to it to a special value \textit{null}, denoting \textit{`unknown’} or \textit{`inapplicable’}.)

Similar issues arise if primary key of Students tuple is updated.
Integrity Constraints (ICs)

IC: condition that must be true for *any* instance of the database; e.g., *domain constraints.*

ICs are specified when schema is defined.

ICs are checked when relations are modified.

A *legal* instance of a relation is one that satisfies all specified ICs.

DBMS should not allow illegal instances.

If the DBMS checks ICs, stored data is more faithful to real-world meaning.

Avoids data entry errors, too!
Where do ICs Come From?

ICs are based upon the semantics of the real-world that is being described in the database relations.

We can check a database instance to see if an IC is violated, but we can NEVER infer that an IC is true by looking at an instance.

    An IC is a statement about all possible instances!
    From example, we know name is not a key, but the assertion that sid is a key is given to us.

Key and foreign key ICs are the most common; more general ICs supported too.
RELATIONAL MODEL

RELATIONAL ALGEBRA

SQL
Relational Algebra

Relational Query Languages

Selection & Projection

Union, Set Difference & Intersection

Cross product & Joins

Examples

Division
Relational Query Languages

*Query languages*: Allow manipulation and retrieval of data from a database.

Relational model supports simple, powerful QLs:
- Strong formal foundation based on logic.
- Allows for much optimization.

Query Languages != programming languages!
- QLs not expected to be “Turing complete”.
- QLs not intended to be used for complex calculations.
- QLs support easy, efficient access to large data sets.
Formal Relational Query Languages

Two mathematical Query Languages form the basis for “real” languages (e.g. SQL), and for implementation:

*Relational Algebra:* More operational, very useful for representing execution plans.

*Relational Calculus:* Lets users describe what they want, rather than how to compute it. (Non-procedural, *declarative*.)

Understanding Algebra is key to understanding SQL, query processing!
Preliminaries

A query is applied to *relation instances*, and the result of a query is also a relation instance.

*Schemas of input* relations for a query are *fixed* (but query will run over any legal instance)

The *schema for the result* of a given query is also *fixed*. It is determined by the definitions of the query language constructs.

**Positional vs. named-field notation:**

Positional notation easier for formal definitions, named-field notation more readable.

Both used in SQL
Relational Algebra: 5 Basic Operations

**Selection** (\(\sigma\)) Selects a subset of *rows* from relation (horizontal).

**Projection** (\(\pi\)) Retains only wanted *columns* from relation (vertical).

**Cross-product** (\(\times\)) Allows us to combine two relations.

**Set-difference** (\(\neg\)) Tuples in R1, but not in R2.

**Union** (\(\bigcup\)) Tuples in R1 and/or in R2.

Since each operation returns a relation, operations can be composed! (Algebra is “closed”.)
**Example Instances**

### Boats

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

### R1

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

### S1

<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>

### S2

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>
Relational Algebra

Relational Query Languages

Selection & Projection

Union, Set Difference & Intersection

Cross product & Joins

Examples

Division
Projection

Examples: \( \pi_{\text{age}}(S2) \) \quad \pi_{\text{sname}, \text{rating}}(S2)

Retains only attributes that are in the “projection list”.

Schema of result:

- exactly the fields in the projection list, with the same names that they had in the input relation.

Projection operator has to eliminate duplicates (How do they arise? Why remove them?)

Note: real systems typically don’t do duplicate elimination unless the user explicitly asks for it. (Why not?)
Projection

\[
\pi_{\text{fname}, \text{rating}}(S2)
\]

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sname</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>uppy</td>
<td>9</td>
</tr>
<tr>
<td>lubber</td>
<td>8</td>
</tr>
<tr>
<td>guppy</td>
<td>5</td>
</tr>
<tr>
<td>rusty</td>
<td>10</td>
</tr>
</tbody>
</table>
Projection

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

\[ \pi_{sname, rating}(S2) \]

\[ \pi_{age}(S2) \]
Selection ($\sigma$)

Selects rows that satisfy selection condition. Result is a relation.

*Schema* of result is same as that of the input relation.

Do we need to do duplicate elimination?

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>puppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

$\sigma_{\text{rating}>8}(S2)$

$\pi_{\text{sname, rating}}(\sigma_{\text{rating}>8}(S2))$
Relational Algebra

Relational Query Languages

Selection & Projection

Union, Set Difference & Intersection

Cross product & Joins

Examples

Division
Union and Set-Difference

All of these operations take two input relations, which must be *union-compatible*:

- Same number of fields.
- “Corresponding” fields have the same type.

For which, if any, is duplicate elimination required?
## Union

<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>

### S1

<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>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
</tbody>
</table>

### S2

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

### S1∪S2
### Set Difference

#### S1

<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>

#### S2

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

#### S1 \( \setminus \) S2

<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>
</tbody>
</table>

#### S2 \( \setminus \) S1

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
</tbody>
</table>
Compound Operator: Intersection

In addition to the 5 basic operators, there are several additional “Compound Operators”

These add no computational power to the language, but are useful shorthands.

Can be expressed solely with the basic ops.

Intersection takes two input relations, which must be union-compatible.

Q: How to express it using basic operators?

\[ R \cap S = R - (R - S) \]
### Intersection

<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>

S1

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

S1 \( \cap \) S2

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>uppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>
Relational Algebra

Relational Query Languages
Selection & Projection
Union, Set Difference & Intersection
Cross product & Joins
Examples
Division
Cross-Product

S1 x R1: Each row of S1 paired with each row of R1.
Q: How many rows in the result?

**Result schema** has one field per field of S1 and R1, with field names “inherited” if possible.

*May have a naming conflict:* Both S1 and R1 have a field with the same name.

In this case, can use the *renaming operator*:

\[ \rho (C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1) \]
**Cross Product Example**

<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>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

\[ S_1 \times R_1 = \]

<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>
Compound Operator: Join

Joins are compound operators involving cross product, selection, and (sometimes) projection.

Most common type of join is a “natural join” (often just called “join”). $R \bowtie S$ conceptually is:

1. Compute $R \times S$
2. Select rows where attributes that appear in both relations have equal values
3. Project all unique attributes and one copy of each of the common ones.

Note: Usually done much more efficiently than this.

Useful for putting “normalized” relations back together.
### Natural Join Example

<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>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

**S1**

**R1**

S1 $\bowtie$ R1 =

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</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>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Natural Join Example

\[ S_1 \times R_1 = \]

<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>
### Natural Join Example

**S1 X R1 =**

<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>
Natural Join Example

\[ S_1 \times R_1 = \]

\[ \sigma \]

\[ \pi \]

<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>

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</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>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Other Types of Joins

**Condition Join (or “theta-join”):**

\[ R \bowtie_c S = \sigma_c (R \times S) \]

<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>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>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

**Result schema** same as that of cross-product.

May have fewer tuples than cross-product.

**Equi-Join:** condition \( c \) contains only conjunction of *equalities*. 
Relational Algebra

Relational Query Languages

Selection & Projection

Union, Set Difference & Intersection

Cross product & Joins

Examples

Division
### 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>

### Boats

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>Blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>Red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>Green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>Red</td>
</tr>
</tbody>
</table>

### Reserves

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Examples

1. Find names of sailors who have reserved boat #103

2. Find names of sailors who have reserved a red boat

3. Find sailors who have reserved a red or a green boat

4. Find sailors who have reserved a red and a green boat
Answers

1. Find names of sailors who have reserved boat #103

Solution 1: \( \pi_{\text{iname}}((\sigma_{\text{bid}=103} \text{Reserves}) \bowtie \text{Sailors}) \)

Solution 2: \( \pi_{\text{iname}}(\sigma_{\text{bid}=103}(\text{Reserves} \bowtie \text{Sailors})) \)
Answers

2. Find names of sailors who have reserved a red boat

Information about boat color only available in Boats; so need an extra join:

$$\pi_{sname}(\sigma_{\text{color} = 'red'} \text{Boats} \bowtie \text{Reserves} \bowtie \text{Sailors})$$

A more efficient solution:

$$\pi_{sname}(\pi_{sid}(\pi_{bid}\sigma_{\text{color} = 'red'} \text{Boats} \bowtie \text{Res} \bowtie \text{Sailors}))$$

A query optimizer can find this given the first solution!
3. Find sailors who have reserved a red or a green boat

Can identify all red or green boats, then find sailors who have reserved one of these boats:

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

\[ \pi_{\text{sname}} (\text{Tempboats} \bowtie\bowtie \text{Reserves} \bowtie\bowtie \text{Sailors}) \]
4. Find sailors who have reserved a red and a green boat

Previous approach won’t work! Must identify sailors who have reserved red boats, sailors who have reserved green boats, then find the intersection (note that sid is a key for Sailors):

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

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

\[ \pi_{\text{name}}((\text{Tempred} \cap \text{Tempgreen}) \bowtie \text{Sailors}) \]
More examples ...

1. Find (the name of) all sailors whose rating is above 9
2. Find all sailors who reserved a boat prior to November 1, 1996
3. Find (the names of) all boats that have been reserved at least once
4. Find all pairs of sailors with the same rating
5. Find all pairs of sailors in which the older sailor has a lower rating
1. Find (the name of) all sailors whose rating is above 9

\[ \pi \text{name}(\sigma \text{rating}>9(Sailors)) \]
Answers ...

2. Find all sailors who reserved a boat prior to November 1, 1996

\[ \pi_{\text{sname}}(\text{Sailors} \bowtie \sigma_{\text{day<'11/1/96'}}(\text{Reserves})) \]
3. Find (the names of) all boats that have been reserved at least once

\[ \pi_{bname}(Boats \bowtie Reserves) \]
4. Find all pairs of sailors with the same rating

\[ \rho (S_1(1\rightarrow sid_1,2\rightarrow sname_1,3\rightarrow rating_1,4\rightarrow age_1), Sailors) \]

\[ \rho (S_2(1\rightarrow sid_2,2\rightarrow sname_2,3\rightarrow rating_2,4\rightarrow age_2), Sailors) \]

\[ \pi_{sname_1, sname_2} (S_1 \bowtie rating_1=rating_2 \land sid_1 \neq sid_2 \ S_2) \]
Answers ...

5. Find all pairs of sailors in which the older sailor has a lower rating

\[ \pi_{sname_1, sname_2} (S1 \bowtie agel > age2 \land rating1 < rating2^{S2}) \]
Relational Algebra

Relational Query Languages

Selection & Projection

Union, Set Difference & Intersection

Cross product & Joins

Examples

Division
**Last Compound Operator: Division**

Useful for expressing “for all” queries like:

*Find sids of sailors who have reserved all boats.*

For A/B attributes of B are subset of attrs of A.

May need to “project” to make this happen.

E.g., let A have 2 fields, x and y ; B have only field y :

\[ A/B = \{ \langle x \rangle | \forall \langle y \rangle \in B (\exists \langle x, y \rangle \in A) \} \]

A/B contains all x tuples such that for *every* y tuple in B, there is an xy tuple in A.
Examples of Division A/B

<table>
<thead>
<tr>
<th>sno</th>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
</tr>
<tr>
<td>s1</td>
<td>p3</td>
</tr>
<tr>
<td>s1</td>
<td>p4</td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
</tr>
<tr>
<td>s2</td>
<td>p2</td>
</tr>
<tr>
<td>s3</td>
<td>p2</td>
</tr>
<tr>
<td>s4</td>
<td>p2</td>
</tr>
<tr>
<td>s4</td>
<td>p4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
</tr>
<tr>
<td>p4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
</tr>
<tr>
<td>p2</td>
</tr>
<tr>
<td>p4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
<tr>
<td>s2</td>
</tr>
<tr>
<td>s3</td>
</tr>
<tr>
<td>s4</td>
</tr>
</tbody>
</table>

A

A/B1

B1

B2

B3
Examples of Division A/B

<table>
<thead>
<tr>
<th>sno</th>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
</tr>
<tr>
<td>s1</td>
<td>p3</td>
</tr>
<tr>
<td>s1</td>
<td>p4</td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
</tr>
<tr>
<td>s2</td>
<td>p2</td>
</tr>
<tr>
<td>s3</td>
<td>p2</td>
</tr>
<tr>
<td>s4</td>
<td>p2</td>
</tr>
<tr>
<td>s4</td>
<td>p4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
<tr>
<td>s2</td>
</tr>
<tr>
<td>s3</td>
</tr>
<tr>
<td>s4</td>
</tr>
</tbody>
</table>

A

A/B1

<table>
<thead>
<tr>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
</tr>
</tbody>
</table>

B1

<table>
<thead>
<tr>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
</tr>
<tr>
<td>p4</td>
</tr>
</tbody>
</table>

B2

<table>
<thead>
<tr>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
</tr>
<tr>
<td>p2</td>
</tr>
<tr>
<td>p4</td>
</tr>
</tbody>
</table>

B3

<table>
<thead>
<tr>
<th>sno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
<tr>
<td>s4</td>
</tr>
</tbody>
</table>

A/B2
Examples of Division A/B

<table>
<thead>
<tr>
<th>sno</th>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
</tr>
<tr>
<td>s1</td>
<td>p3</td>
</tr>
<tr>
<td>s1</td>
<td>p4</td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
</tr>
<tr>
<td>s2</td>
<td>p2</td>
</tr>
<tr>
<td>s3</td>
<td>p2</td>
</tr>
<tr>
<td>s4</td>
<td>p2</td>
</tr>
<tr>
<td>s4</td>
<td>p4</td>
</tr>
</tbody>
</table>

A

A/B1

B1

<table>
<thead>
<tr>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
</tr>
</tbody>
</table>

B2

<table>
<thead>
<tr>
<th>pno</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
</tr>
<tr>
<td>p4</td>
</tr>
</tbody>
</table>

B3

<table>
<thead>
<tr>
<th>sno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
<tr>
<td>s2</td>
</tr>
</tbody>
</table>

A/B2

<table>
<thead>
<tr>
<th>sno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
<tr>
<td>s3</td>
</tr>
<tr>
<td>s4</td>
</tr>
</tbody>
</table>

B3

<table>
<thead>
<tr>
<th>sno</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
</tbody>
</table>

A/B3
Expressing A/B Using Basic Operators

Division is not essential op; just a useful shorthand.

(Also true of joins, but joins are so common that systems implement joins specially.)

Idea: For A/B, compute all x values that are not “disqualified” by some y value in B.

x value is disqualified if by attaching y value from B, we obtain a xy tuple that is not in A.

\[
\text{Disqualified x values: } \pi x \left( (\pi x(A) \times B) - A \right)
\]

\[
A/B: \quad \pi x(A) - \text{Disqualified x values}
\]
Expressing A/B: \( \pi_{sno}(A) - \pi_{sno}((\pi_{sno}(A) \times B) - A) \)

\[
\begin{array}{|c|c|}
\hline
sno & pno \\
\hline
s1 & p1 \\
s1 & p2 \\
s1 & p3 \\
s1 & p4 \\
s2 & p1 \\
s2 & p2 \\
s3 & p2 \\
s4 & p2 \\
s4 & p4 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
sno & pno \\
\hline
s1 & p1 \\
s1 & p2 \\
s1 & p4 \\
s2 & p1 \\
s2 & p2 \\
s2 & p4 \\
s3 & p1 \\
s3 & p2 \\
s4 & p2 \\
s4 & p4 \\
\hline
\end{array}
\]

\[
A \implies B
\]

\[
T1 = \pi_{sno}(A) \times B
\]
Expressing $A/B$: $\pi_{sno}(A) - \pi_{sno}((\pi_{sno}(A) \times B) - A)$

$T_1 = \pi_{sno}(A) \times B$

$T_2 = \pi_{sno}(T_1 - A)$
Expressing $A/B$: \[ \pi_{\text{sno}}(A) - \pi_{\text{sno}}((\pi_{\text{sno}}(A) \times B) - A) \]

\[ A = \begin{array}{|c|c|}
\hline
\text{sno} & \text{pno} \\
\hline
s1 & p1 \\
s1 & p2 \\
s1 & p3 \\
s1 & p4 \\
s2 & p1 \\
s2 & p2 \\
s3 & p2 \\
s4 & p2 \\
s4 & p4 \\
\hline
\end{array} \]

\[ B = \begin{array}{|c|}
\hline
\text{pno} \\
\hline
p1 \\
p2 \\
p4 \\
\hline
\end{array} \]

\[ T1 = \pi_{\text{sno}}(A) \times B \]

\[ T2 = \pi_{\text{sno}}(T1 - A) \]

\[ A/B = \pi_{\text{sno}}(A) - T2 \]
Example of Division

Find the names of sailors who have reserved all boats

Uses division; schemas of the input relations to / must be carefully chosen:

\[
\rho (Tempsids, (\pi \text{sid,bid Reserves}) / (\pi \text{bid Boats}))
\]

\[
\pi \text{sname} (Tempsids \bowtie \text{Sailors})
\]

To find sailors who have reserved all “Interlake” boats:

\[
\ldots \pi \text{bid} (\sigma \text{bname} \neq \text{Interlake} \text{Boats})
\]
RELATIONAL MODEL

RELATIONAL ALGEBRA

SQL
Moving on to SQL

Database Management Systems (DBMS) store and manage large quantities of data

We want an **intuitive** way to ask **questions (queries)**

You have been taught *procedural languages* (C, java) which specify **how** to solve a problem (or answer a question)

Now, we talk about **SQL**

SQL is a **declarative query** language

We ask **what we want** and the DBMS is going to deliver!
SQL - A language for Relational DBs

SQL* (a.k.a. “Sequel”), standard language

Data Definition Language (DDL)
- create, modify, delete relations
- specify constraints
- administer users, security, etc.

Data Manipulation Language (DML)
- Specify *queries* to find tuples that satisfy criteria
- add, modify, remove tuples
Reiterate some terminology

**Relation (or table)**

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

**Row (or tuple)**

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

**Column (or attribute)**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
</tr>
</tbody>
</table>
Reiterate some terminology

Primary Key (PK)

The PK of a relation is the column (or the group of columns) that can uniquely define a row.

In other words:

Two rows cannot have the same PK.

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
</tr>
</tbody>
</table>
SQL Overview

CREATE TABLE <name> ( <field> <domain>, ... )

INSERT INTO <name> (<field names>)
  VALUES (<field values>)

DELETE FROM <name>
  WHERE <condition>

UPDATE <name>
  SET <field name> = <value>
  WHERE <condition>

SELECT   <fields>
FROM     <name>
WHERE    <condition>
GROUP BY <fields>
HAVING   <condition>
ORDER BY <fields>
Creating Relations in SQL

Creates the *Students* relation.

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

```sql
CREATE TABLE Students
    (sid CHAR(20),
     name CHAR(20),
     login CHAR(10),
     age INTEGER,
     gpa FLOAT)
```
Another example: the Enrolled table holds information about courses students take.

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2))
Primary and Candidate Keys in SQL

Possibly many *candidate keys* (specified using `UNIQUE`), one of which is chosen as the *primary key*.

Keys must be used carefully!

“For a given student and course, there is a single grade.”

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

VS.

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

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

*Foreign key*: Set of fields in one relation that is used to “refer” to a tuple in another relation.

- Must correspond to the primary key of the other relation.
- Like a “logical pointer”.

If all foreign key constraints are enforced, *referential integrity* is achieved (i.e., no dangling references.)
Foreign Keys in SQL

Example: Only students listed in the Students relation should be allowed to enroll for courses.

*sid* is a foreign key referring to Students:

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

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-101</td>
<td>C</td>
</tr>
<tr>
<td>53666</td>
<td>18-203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>15-112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>15-105</td>
<td>B</td>
</tr>
</tbody>
</table>

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

Can insert a single tuple using:

\[
\text{INSERT INTO Students (sid, name, login, age, gpa) VALUES (‘53688’, ‘Smith’, ‘smith@cs’, 18, 3.2)}
\]

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

\[
\text{DELETE FROM Students S WHERE S.name = ‘Smith’}
\]

Powerful variants of these commands are available; more later!
The simplest SQL query

“Find all contents of a table”
In this example: “Find all info for all students”

```
SELECT *
FROM Students S
```

To find just names and logins, replace the first line:

```
SELECT S.name, S.login
```
Show specific columns

“Find name and login for all students”

```
SELECT S.name, S.login
FROM Students S
```

<table>
<thead>
<tr>
<th>name</th>
<th>login</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>jones@cs</td>
</tr>
<tr>
<td>Smith</td>
<td>smith@ee</td>
</tr>
<tr>
<td>White</td>
<td>white@cs</td>
</tr>
</tbody>
</table>

This is called: “Project name and login from table Students”
Show specific rows

“Find all 18 year old students”

```
SELECT * 
FROM Students S 
WHERE S.age=18
```

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

This is called: “Select students with age 18.”
Querying Multiple Relations

Can specify a join over two tables as follows:

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='B'
```

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

```
sid      | name   | login     | age | gpa  \\
---|--------|-----------|-----|------
53666    | Jones  | jones@cs  | 18  | 3.4  \\
53688    | Smith  | smith@ee  | 18  | 3.2  
```

```
result =

<table>
<thead>
<tr>
<th>S.name</th>
<th>E.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>History105</td>
</tr>
</tbody>
</table>
```
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 tables in relation-list
qualification : Comparisons combined using AND, OR and NOT.

Comparisons are Attr op const or Attr1 op Attr2, where op is one of <, >, =, ≤, ≥, ≠

DISTINCT: optional keyword indicating that the answer should not contain duplicates.

In SQL SELECT, the default is that duplicates are not eliminated! (Result is called a “multiset”)

<table>
<thead>
<tr>
<th>SELECT</th>
<th>[DISTINCT] target-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>relation-list</td>
</tr>
<tr>
<td>WHERE</td>
<td>qualification</td>
</tr>
</tbody>
</table>
Query Semantics

Conceptually, a SQL query can be computed:

1. **FROM**: compute *cross-product* of tables (e.g., Students and Enrolled).
2. **WHERE**: Check conditions, discard tuples that fail. (called “*selection*”).
3. **SELECT**: Delete unwanted fields. (called “*projection*”).
4. If **DISTINCT** specified, eliminate duplicate rows.

Probably the least efficient way to compute a query!

**Query Optimization** helps us find more efficient strategies to get the *same answer*. 
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='B'

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Select all students who have grade ‘B’

**Step 1 – Cross Product**

Combine with cross-product all tables of the **FROM** clause.

<table>
<thead>
<tr>
<th>S.sid</th>
<th>S.name</th>
<th>S.login</th>
<th>S.age</th>
<th>S.gpa</th>
<th>E.sid</th>
<th>E.cid</th>
<th>E.grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53832</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

**SQL Query**

```sql
SELECT S.name, E.cid 
FROM Students S, Enrolled E 
WHERE S.sid=E.sid AND E.grade='B'
```
**Step 2 - Discard tuples that fail predicate**

Make sure the **WHERE** clause is true!

<table>
<thead>
<tr>
<th>S.sid</th>
<th>S.name</th>
<th>S.login</th>
<th>S.age</th>
<th>S.gpa</th>
<th>E.sid</th>
<th>E.cid</th>
<th>E.grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53832</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

**SQL Query**

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='B'
```
Step 3 - Discard Unwanted Columns

Show only what is on the `SELECT` clause.

<table>
<thead>
<tr>
<th>S.sid</th>
<th>S.name</th>
<th>S.login</th>
<th>S.age</th>
<th>S.gpa</th>
<th>E.sid</th>
<th>E.cid</th>
<th>E.grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53832</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

```sql
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='B'
```
Was this a fast way to evaluate:

```sql
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade='B'
```

If not what is an efficient way?
Now the Details...

We will use these instances of relations in our examples.
Example Schemas

CREATE TABLE Sailors (sid INTEGER, sname CHAR(20), rating INTEGER, age REAL, PRIMARY KEY sid)

CREATE TABLE Boats (bid INTEGER, bname CHAR(20), color CHAR(10), PRIMARY KEY bid)

CREATE TABLE Reserves (sid INTEGER, bid INTEGER, day DATE, PRIMARY KEY (sid, bid, date), FOREIGN KEY sid REFERENCES Sailors, FOREIGN KEY bid REFERENCES Boats)
### Another Join Query

```
SELECT sname
FROM   Sailors, Reserves
WHERE  Sailors.sid = Reserves.sid
       AND 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>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
<td>95</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Range Variables (1/2)

Can associate “range variables” with the tables in the FROM clause.

saves writing, makes queries easier to understand

Needed when ambiguity could arise.

for example, if same table used multiple times in same FROM (called a “self-join”)

```
SELECT sname
FROM Sailors, Reserves
WHERE Sailors.sid = Reserves.sid AND bid=103
```

Can be rewritten using range variables as:

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND bid=103
```
Range Variables (2/2)

Here is an example where range variables are required (self-join example):

```
SELECT  x.sname, x.age, y.sname, y.age
FROM Sailors x, Sailors y
WHERE  x.age > y.age
```

Note that target list can be replaced by "*" if you don’t want to do a projection:

```
SELECT  *
FROM Sailors x
WHERE  x.age > 20
```
Find sailors who have reserved at least one boat

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

Would adding DISTINCT to this query make a difference?

What is the effect of replacing \textit{S.sid} by \textit{S.sname} in the SELECT clause?

Would adding DISTINCT to this variant of the query make a difference?
Expressions

Can use arithmetic expressions in SELECT clause (plus other operations we’ll discuss later)

Use **AS** to provide column names

```
SELECT S.age, S.age-5 AS age1, 2*S.age AS age2
FROM   Sailors S
WHERE  S.sname = 'dustin'
```

Can also have expressions in WHERE clause:

```
SELECT  S1.sname AS name1, S2.sname AS name2
FROM   Sailors S1, Sailors S2
WHERE  2*S1.rating = S2.rating - 1
```
String operations

SQL also supports some string operations
“LIKE” is used for string matching.

```
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_%B'
```

__’ stands for any one character
‘%’ stands for 0 or more arbitrary characters.
Logical Operations

SQL queries produce new tables

If the results of two queries are set-compatible (same # and types columns) then we can apply logical operations

- UNION
- INTERSECTION
- SET DIFFERENCE (called EXCEPT or MINUS)
Find sids of sailors who have reserved a red or a green boat.

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

```
SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND
(B.color = 'red' OR B.color = 'green')
```

Vs.

```
SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = 'red'
UNION
SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = 'green'
```
Find sids of sailors who have reserved a red and a green boat

If we simply replace OR by AND in the previous query, we get the wrong answer. (Why?) Instead, could use a self-join:

```sql
SELECT R1.sid
FROM Boats B1, Reserves R1,
     Boats B2, Reserves R2
WHERE R1.sid=R2.sid
  AND R1.bid=B1.bid
  AND R2.bid=B2.bid
  AND (B1.color='red' AND B2.color='green')
```
INTERSECT: discussed in the book. Can be used to compute the intersection of any two *union-compatible* sets of tuples.

Also in text: EXCEPT (sometimes called MINUS)

Included in the SQL/92 standard, but some systems don’t support them.
1. Find (the names of) all sailors who are over 50 years old
2. Find (the names of) all boats that have been reserved at least once
3. Find all sailors who have not reserved a red boat (hint: use “EXCEPT”)
4. Find all pairs of same-color boats
5. Find all pairs of sailors in which the older sailor has a lower rating
1. Find (the names of) all sailors who are over 50 years old

$$\text{SELECT S.sname FROM Sailors S WHERE S.age > 50}$$
2. Find (the names of) all boats that have been reserved at least once

```
SELECT DISTINCT B.bname
FROM    Boats B, Reserves R
WHERE   R.bid=B.bid
```
3. Find all sailors who have not reserved a red boat

```
SELECT S.sid
FROM   Sailors S
EXCEPT
SELECT R.sid
FROM   Boats B,Reserves R
WHERE  R.bid=B.bid
       AND B.color=‘red’
```
Answers ...

4. Find all pairs of same-color boats

```sql
SELECT B1.bname, B2.bname
FROM   Boats B1, Boats B2
WHERE  B1.color = B2.color
```
5. Find all pairs of sailors in which the **older** sailor has a **lower** rating

```sql
SELECT S1.sname, S2.sname
FROM   Sailors S1, Sailors S2
WHERE  S1.age > S2.age
       AND S1.rating < S2.rating
```
Queries With GROUP BY and HAVING

Group rows by columns in grouping-list
Use the HAVING clause to restrict which group-rows are returned in the result set
Conceptual Evaluation

1. Cross-product of *relation-list*
2. Select only tuples that follow the where clause *(qualification)*
3. Partition rows by the value of attributes in *grouping-list*
4. Select only groups that follow the *group-qualification*
   
   Expressions in *group-qualification* must have a *single value per group*! That is, attributes in *group-qualification* must be arguments of an aggregate op or must also appear in the *grouping-list*.

5. One answer tuple is generated per qualifying group.
Find the age of the youngest sailor with age \( \geq 18 \), for each rating with at least 2 such sailors

\[
\text{SELECT} \quad S.\text{rating}, \quad \text{MIN} (S.\text{age}) \\
\text{FROM} \quad \text{Sailors} \ S \\
\text{WHERE} \quad S.\text{age} \geq 18 \\
\text{GROUP BY} \quad S.\text{rating} \\
\text{HAVING} \quad \text{COUNT} \ (\star) > 1
\]
Find sailors who’ve reserved all boats.

Example in book, not using EXCEPT:

SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (SELECT B.bid
               FROM Boats B
               WHERE NOT EXISTS (SELECT R.bid
                                   FROM Reserves R
                                   WHERE R.bid=B.bid
                                   AND R.sid=S.sid)))

Sailors S such that ...

there is no boat B without ...

a Reserves tuple showing S reserved B
Find sailors who’ve reserved all boats.

Can you do this using Group By and Having?

```
SELECT    S.name
FROM      Sailors S, reserves R
WHERE     S.sid = R.sid
GROUP BY  S.name, S.sid
HAVING    COUNT(DISTINCT R.bid) =
           (Select COUNT(*) FROM Boats)
```

Note: must have both sid and name in the GROUP BY clause. Why?
```
SELECT  S.name, S.sid  
FROM    Sailors S, reserves R  
WHERE   S.sid = r.sid  
GROUP BY S.name, S.sid  
HAVING  COUNT(DISTINCT R.bid) =  
        (Select COUNT(*) FROM Boats_)
```

```
<table>
<thead>
<tr>
<th>s.name</th>
<th>s.sid</th>
<th>r.sid</th>
<th>r.bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dustin</td>
<td>22</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>Lubber</td>
<td>31</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>Bob</td>
<td>95</td>
<td>22</td>
<td>101</td>
</tr>
<tr>
<td>Dustin</td>
<td>22</td>
<td>95</td>
<td>102</td>
</tr>
<tr>
<td>Lubber</td>
<td>31</td>
<td>95</td>
<td>102</td>
</tr>
<tr>
<td>Bob</td>
<td>95</td>
<td>95</td>
<td>102</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>
```

Count (*) from boats = 4

Apply having clause to groups
Sorting the Results of a Query

ORDER BY \textit{column} \ [ ASC \mid DESC ] [, ...]

\begin{verbatim}
SELECT S.rating, S.sname, S.age
FROM Sailors S, Boats B, Reserves R
    AND B.color='red'
ORDER BY S.rating, S.sname;
\end{verbatim}

Extra reporting power obtained by combining with aggregation.

\begin{verbatim}
SELECT S.sid, COUNT (*) AS redrescnt
FROM Sailors S, Boats B, Reserves R
    AND B.color='red'
GROUP BY S.sid
ORDER BY redrescnt DESC;
\end{verbatim}