SQL

CMPSCI 645

Feb 13, 2006
Today: SQL Overview

- Query capabilities
  - SELECT-FROM-WHERE blocks
  - Set ops (union, intersect, except)
  - Nested queries (correlation)
  - Aggregation & Grouping
  - Null values, Outer Joins
- Modifying the database
- Views
- Access control: GRANT/REVOKE
- Integrity constraints

Refer to textbook, Ch 5
Basic SQL Query

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

- **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.

- Probably the least efficient way to compute a query -- optimizer will find more efficient plan.
Example of Conceptual Evaluation

SELECT S.sname
FROM Sailors 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>
</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>

<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>
Ordering the Results

```
SELECT   pname, price, manufacturer
FROM     Product
WHERE    category='gizmo' AND price > 50
ORDER BY price, pname
```

Ordering is ascending, unless you specify the DESC keyword.

Ties are broken by the second attribute on the ORDER BY list, etc.
Ordering the Results

```
SELECT Category
FROM Product
ORDER BY PName
```

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>
Joins

Product (pname, price, category, manufacturer)
Company (cname, stockPrice, country)

Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT PName, Price
FROM Product, Company
WHERE Manufacturer=CName AND Country='Japan'
    AND Price <= 200
```

Join between Product and Company
Joins in SQL

**Product**

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photog</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

**Company**

<table>
<thead>
<tr>
<th>Cname</th>
<th>StockPrice</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

**SQL Query**

```sql
SELECT PName, Price
FROM Product, Company
WHERE Manufacturer=CName AND Country='Japan'
     AND Price <= 200
```

**Result**

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
</tr>
</tbody>
</table>
Joins

Product (pname, price, category, manufacturer)
Purchase (buyer, seller, store, product)
Person(persname, phoneNumber, city)

Find names of people living in Seattle that bought some product in the ‘Gadgets’ category, and the names of the stores they bought such product from

```
SELECT DISTINCT persname, store
FROM Person, Purchase, Product
WHERE persname=buyer AND product = pname AND city='Seattle' AND category='Gadgets'
```
Disambiguating Attributes

- Sometimes two relations have the same attr:
  Person(pname, address, worksfor)
  Company(cname, address)

\[
\text{SELECT DISTINCT } \text{pname, address} \\
\text{FROM } \text{Person, Company} \\
\text{WHERE } \text{worksfor} = \text{cname}
\]

Which address?
Tuple Variables in SQL

Purchase (buyer, seller, store, product)

Find all stores that sold at least one product that was sold at ‘BestBuy’:

```sql
SELECT DISTINCT x.store
FROM Purchase AS x, Purchase AS y
WHERE x.product = y.product AND y.store = 'BestBuy'
```
Tuple Variables

General rule:
tuple variables introduced automatically by the system:

Product (name, price, category, manufacturer)

Becomes:

```
SELECT name
FROM Product
WHERE price > 100
```

Becomes:

```
SELECT Product.name
FROM Product AS Product
WHERE Product.price > 100
```

 Doesn’t work when Product occurs more than once:
In that case the user needs to define variables explicitly.
Renaming Columns

```
SELECT Pname AS prodName, Price AS askPrice
FROM Product
WHERE Price > 100
```
Set Operations: Union, Intersection, Difference

(\text{SELECT name} \\
\text{FROM Person} \\
\text{WHERE City=“Seattle”}) \\
\text{UNION} \\
(\text{SELECT name} \\
\text{FROM Person, Purchase} \\
\text{WHERE buyer=name AND store=“Whole Foods”})

Similarly, you can use \text{INTERSECT} and \text{EXCEPT}. You must have the same attribute names (otherwise: rename).
Conserving Duplicates

(\textbf{SELECT} name
FROM Person
WHERE City=“Seattle”)

\textbf{UNION} \textbf{ALL}

(\textbf{SELECT} name
FROM Person, Purchase
WHERE buyer=name AND store=“Whole Foods”)

\textbf{(SELECT name
FROM Person
WHERE City=“Seattle”)
UNION ALL
(SELECT name
FROM Person, Purchase
WHERE buyer=name AND store=“Whole Foods”)}
Subqueries

A subquery producing a single value:

```
SELECT Purchase.product
FROM     Purchase
WHERE  buyer =
                 (SELECT  name
                   FROM     Person
                   WHERE   ssn = ‘123456789‘);
```

In this case, the subquery returns one value.

If it returns more, it’s a run-time error.
Subqueries

Can say the same thing without a subquery:

```
SELECT Purchase.product
FROM     Purchase, Person
WHERE  buyer = name AND ssn = '123456789'
```
Subqueries Returning Relations

Find companies that manufacture products bought by Joe.

```
SELECT Company.name
FROM      Company, Product
WHERE     Company.name=Product.maker
          AND Product.name IN
            (SELECT Purchase.product
             FROM     Purchase
             WHERE    Purchase .buyer = 'Joe');
```

Here the subquery returns a set of values: no more runtime errors.
Subqueries Returning Relations

Other set-comparison operators:

- \( s > \text{ALL} \ R \)
- \( s > \text{ANY} \ R \)
- \( \text{EXISTS} \ R \)
- \( \text{UNIQUE} \ R \)

**Product** (\( \text{pname, price, category, maker} \))

Find products that are more expensive than all those produced by “Gizmo-Works”

```
SELECT name
FROM Product
WHERE price > ALL (SELECT price
                     FROM Purchase
                     WHERE maker='Gizmo-Works')
```
Correlated Queries

Movie (title, year, director, length)
Find movies whose title appears more than once.

```
SELECT DISTINCT title
FROM Movie AS x
WHERE year <> ANY
     (SELECT year
      FROM Movie
      WHERE title = x.title);
```

Synonyms: ANY, SOME
Complex Correlated Query

Product (pname, price, category, maker, year)

```
SELECT DISTINCT pname, maker
FROM Product AS x
WHERE price > ALL (SELECT price
                      FROM Product AS y
                      WHERE x.maker = y.maker AND y.year < 1972);
```

- Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

Powerful, but much harder to optimize!
Existential/Universal Conditions

Product (pname, price, company)
Company (cname, city)

Find all companies s.t. some of their products have price < 100

```
SELECT DISTINCT Company.cname
FROM Company, Product
WHERE Company.cname = Product.company and Product.price < 100
```

Existential: easy ！😊
Existential/Universal Conditions

Product (pname, price, company)
Company (cname, city)

Find companies s.t. all of their products have price < 100

Universal: hard 😞
Existential/Universal Conditions

1. Find the other companies: i.e. s.t. some product $\geq 100$

   $$
   \text{SELECT DISTINCT Company.cname}
   \text{FROM Company}
   \text{WHERE Company.cname IN (SELECT Product.company}
   \text{FROM Product}
   \text{WHERE Product.price }\geq 100)
   $$

2. Find all companies s.t. all their products have price $< 100$

   $$
   \text{SELECT DISTINCT Company.cname}
   \text{FROM Company}
   \text{WHERE Company.cname NOT IN (SELECT Product.company}
   \text{FROM Product}
   \text{WHERE Product.price }\geq 100)
   $$
Aggregation

```
SELECT Avg(price)
FROM Product
WHERE maker="Toyota"
```

SQL supports several aggregation operations:

- `COUNT (*)`
- `COUNT ( [DISTINCT] A )`
- `SUM ( [DISTINCT] A )`
- `AVG ( [DISTINCT] A )`
- `MAX (A)`
- `MIN (A)`
Aggregation: Count

```
SELECT  Count(*)
FROM     Product
WHERE   year > 1995
```

Except COUNT, all aggregations apply to a single attribute
Aggregation: Count

COUNT   applies to duplicates, unless otherwise stated:

```
SELECT  Count(category)
FROM     Product
WHERE   year > 1995
```

same as Count(*)

Better:

```
SELECT  Count(DISTINCT category)
FROM     Product
WHERE   year > 1995
```
Simple Aggregation

Purchase(product, date, price, quantity)

Example 1: find total sales for the entire database

SELECT Sum(price * quantity) 
FROM Purchase

Example 1’: find total sales of bagels

SELECT Sum(price * quantity) 
FROM Purchase
WHERE product = ‘bagel’
### Grouping

#### Sailors

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33</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>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35</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>74</td>
<td>horatio</td>
<td>9</td>
<td>35</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16</td>
</tr>
</tbody>
</table>

#### New Table

<table>
<thead>
<tr>
<th>rating</th>
<th>age?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
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.
Conceptual Evaluation

- The cross-product of relation-list is computed, tuples that fail qualification are discarded, `unnecessary` fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in grouping-list.
- The group-qualification is then applied to eliminate some groups. Expressions in group-qualification must have a single value per group!
- One answer tuple is generated per qualifying group.
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

**Answer relation:**

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>

**Sailors instance:**

<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>
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
NULLS in SQL

• Whenever we don’t have a value, we can put a NULL
• Can mean many things:
  – Value does not exists
  – Value exists but is unknown
  – Value not applicable
  – Etc.
• The schema specifies for each attribute whether it can be null (*nullable* attribute)
• How does SQL cope with tables that have NULLs?
Null Values

• If x = NULL then 4*(3-x)/7 is still NULL

• If x = NULL then x = “Joe” is UNKNOWN

• In SQL there are three boolean values:
  
  FALSE = 0
  UNKNOWN = 0.5
  TRUE = 1
Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

Example SQL query:

```
SELECT *
FROM Person
WHERE (age < 25) AND
      (height > 6 OR weight > 190)
```

E.g.
age=20
heigth=NULL
weight=200

Rule in SQL: include only tuples that yield TRUE
Null Values

Unexpected behavior:

```
SELECT *
FROM Person
WHERE age < 25  OR  age >= 25
```

Some Persons are not included!
Null Values

Can test for NULL explicitly:
- \( x \) IS NULL
- \( x \) IS NOT NULL

```sql
SELECT *
FROM Person
WHERE age < 25 OR age >= 25 OR age IS NULL
```

Now it includes all Persons
Outerjoins

Product(name, category)

Purchase(prodName, store)

Display list of all products, along with the stores where they were sold:

```
SELECT Product.name, Purchase.store
FROM     Product, Purchase
WHERE   Product.name = Purchase.prodName
```

But Products that never sold will be lost!
Outerjoins

Left outer joins in SQL:

Product(name, category)
Purchase(prodName, store)

```
SELECT Product.name, Purchase.store
FROM Product
LEFT OUTER JOIN Purchase
ON Product.name = Purchase.prodName
```
<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ProdName</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

**[LEFT OUTER JOIN]**

<table>
<thead>
<tr>
<th>Name</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
<tr>
<td>OneClick</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Outer Joins

• Left outer join:
  – Include the left tuple even if there’s no match

• Right outer join:
  – Include the right tuple even if there’s no match

• Full outer join:
  – Include the both left and right tuples even if there’s no match
Review: SQL Query capabilities

- SELECT-FROM-WHERE blocks
- Set ops (union, intersect, except)
- Nested queries (correlation)
  - set comparison operators
- Aggregation
- GROUP BY, HAVING
- Nulls, 3 valued logic
- Outer Joins
Modifying the Database

Three kinds of modifications

- Insertions
- Deletions
- Updates

Sometimes they are all called “updates”
Insertions

General form:

\[
\text{INSERT INTO } R(A_1, \ldots, A_n) \text{ VALUES } (v_1, \ldots, v_n)
\]

Example: Insert a new purchase to the database:

\[
\text{INSERT INTO } \text{Purchase(buyer, seller, product, store)} \text{ VALUES } ('Joe', 'Fred', 'wakeup-clock-espresso-machine', 'The Sharper Image')
\]

Missing attribute $\rightarrow$ NULL.
May drop attribute names if give them in order.
The query replaces the VALUES keyword.
Here we insert many tuples into PRODUCT
Deletions

Example:

```
DELETE FROM PURCHASE
WHERE seller = 'Joe' AND product = 'Brooklyn Bridge'
```

Factoid about SQL: there is no way to delete only a single occurrence of a tuple that appears twice in a relation.
Updates

Example:

```
UPDATE PRODUCT
SET price = price/2
WHERE Product.name IN
      (SELECT product
       FROM Purchase
       WHERE Date = 'Oct, 25, 1999');
```
Creating Relations in SQL

- Creates the Students 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 Enrolled table holds information about courses that students take.

```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

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

ALTER TABLE Students
ADD COLUMN firstYear: integer

- The schema of Students is altered by adding a new field; every tuple in the current instance is extended with a null value in the new field.
Views

• A **view** is just a relation, but we store a *definition*, rather than a set of tuples.

```
CREATE VIEW YoungActiveStudents (name, grade)
    AS SELECT S.name, E.grade
    FROM Students S, Enrolled E
    WHERE S.sid = E.sid and S.age<21
```

• Views can be dropped using the **DROP VIEW** command.
  • How to handle **DROP TABLE** if there’s a view on the table?
    • **DROP TABLE** command has options to let the user specify this.
Views are important

- Data independence
- Simplifying queries by isolating common sub-expressions
- Security
- Data integration
- If materialized: speed up query processing

Issue we will return to later: updating views
Discretionary Access Control in SQL

GRANT privileges ON object TO users [WITH GRANT OPTIONS]

privileges = SELECT | INSERT(column-name) | UPDATE(column-name) | DELETE | REFERENCES(column-name)

object = table | attribute
GRANT INSERT, DELETE ON Customers TO **Yuppy** WITH GRANT OPTIONS

Queries allowed to Yuppy:

INSERT INTO Customers(cid, name, address)
VALUES(32940, 'Joe Blow', 'Seattle')

DELETE Customers
WHERE LastPurchaseDate < 1995

Queries denied to Yuppy:

SELECT Customer.address
FROM Customer
WHERE name = 'Joe Blow'
Examples

GRANT SELECT ON Customers TO Michael

Now Michael can SELECT, but not INSERT or DELETE
Examples

GRANT SELECT ON Customers TO Michael WITH GRANT OPTIONS

Michael can say this:
   GRANT SELECT ON Customers TO Yuppi

Now Yuppi can SELECT on Customers
Examples

GRANT UPDATE (price) ON Product TO Leah

Leah can update, but only Product.price, but not Product.name
CREATE VIEW PublicCustomers
         SELECT Name, Address
         FROM Customers
GRANT SELECT ON PublicCustomers TO Fred

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Huston</td>
<td>450.99</td>
</tr>
<tr>
<td>Sue</td>
<td>Seattle</td>
<td>-240</td>
</tr>
<tr>
<td>Joan</td>
<td>Seattle</td>
<td>333.25</td>
</tr>
<tr>
<td>Ann</td>
<td>Portland</td>
<td>-520</td>
</tr>
</tbody>
</table>

David owns

Fred is not allowed to see this

David says
Views and Security

Customers:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Huston</td>
<td>450.99</td>
</tr>
<tr>
<td>Sue</td>
<td>Seattle</td>
<td>-240</td>
</tr>
<tr>
<td>Joan</td>
<td>Seattle</td>
<td>333.25</td>
</tr>
<tr>
<td>Ann</td>
<td>Portland</td>
<td>-520</td>
</tr>
</tbody>
</table>

CREATE VIEW BadCreditCustomers
         SELECT *
         FROM Customers
         WHERE Balance < 0
GRANT SELECT ON BadCreditCustomers TO John

David says

John is allowed to see only <0 balances

David owns
Views and Security

- Each customer should see only her/his record

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary</td>
<td>Huston</td>
<td>450.99</td>
</tr>
<tr>
<td>Sue</td>
<td>Seattle</td>
<td>-240</td>
</tr>
<tr>
<td>Joan</td>
<td>Seattle</td>
<td>333.25</td>
</tr>
<tr>
<td>Ann</td>
<td>Portland</td>
<td>-520</td>
</tr>
</tbody>
</table>

CREATE VIEW CustomerMary
SELECT * FROM Customers
WHERE name = ‘Mary’
GRANT SELECT ON CustomerMary TO Mary

CREATE VIEW CustomerSue
SELECT * FROM Customers
WHERE name = ‘Sue’
GRANT SELECT ON CustomerSue TO Sue

Doesn’t scale.
Need *row-level* access control!
Revocation

```
REVOKE [GRANT OPTION FOR] privileges
      ON object FROM users  {  RESTRICT | CASCADE  }
```

Administrator says:

```
REVOKE SELECT ON Customers  FROM David CASCADE
```

John loses SELECT privileges on BadCreditCustomers
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!
Primary Key Constraints

• A set of fields is a **key** for a relation if:
  1. No two distinct tuples can have same values in all key fields, and
  2. This is not true for any subset of the key.
     - Part 2 false? A **superkey**.
     - If there’s >1 key for a relation, one of the keys is chosen (by DBA) to be the **primary key**.

• E.g., *sid* is a key for Students. (What about *name*?) The set \{*sid*, *gpa*\} is a superkey.
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.) Like a `logical pointer`.

• E.g. *sid* is a foreign key referring to **Students**:
  - `Enrolled(sid: string, cid: string, grade: string)`
  - If all foreign key constraints are enforced, **referential integrity** is achieved, i.e., no dangling references.
  - Can you name a data model w/o referential integrity?
    • Links in HTML!
Foreign Keys in SQL

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

```
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>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Carnatic101</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>53666</td>
<td>Reggae203</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
<td></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@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>
Enforcing Referential Integrity

- Consider Students and Enrolled; 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? (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 default sid.
  - (In SQL, also: Set sid in Enrolled tuples that refer to it to a special value null, denoting ‘unknown’ or ‘inapplicable’.)
- Similar if primary key of Students tuple is updated.
Referential Integrity in SQL

- SQL/92 and SQL:1999 support 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 Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid),
FOREIGN KEY (sid)
  REFERENCES Students
    ON DELETE CASCADE
    ON UPDATE SET DEFAULT)
```
Where do ICs Come From?

- ICs are based upon the semantics of the real-world enterprise 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.