Relation Definition

- **Database is collection of relations**
- **Relation R is subset of \( S_1 \times S_2 \times \ldots \times S_n \)**
  - Where \( S_i \) is the domain of attribute \( i \)
  - \( n \) is number of attributes of the relation
- **Relation is basically a table with rows & columns**
  - SQL uses word table to refer to relations

Properties of a Relation

- Each row represents an \( n \)-tuple of \( R \)
- Ordering of rows is immaterial
- All rows are distinct
- Ordering of columns is significant
  - Because two columns can have same domain
  - But columns are labeled so
  - Applications need not worry about order
  - They can simply use the names
- Domain of each column is a primitive type
- Relation consists of a **relation schema** and **instance**

More Definitions

- **Relation schema**: describes column heads
  - Relation name
  - Name of each field (or column, or attribute)
  - Domain of each field
- **Degree (or arity) of relation**: \( nb \) attributes
- **Database schema**: set of all relation schemas

Even More Definitions

- **Relation instance**: concrete table content
  - Set of tuples (also called records) matching the schema
- **Cardinality of relation instance**: \( nb \) tuples
- **Database instance**: set of all relation instances

Example

- **Relation schema**
  - `Supplier(sno: integer, sname: string, scity: string, sstate: string)`
- **Relation instance**

<table>
<thead>
<tr>
<th>sno</th>
<th>sname</th>
<th>scity</th>
<th>sstate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s1</td>
<td>city1</td>
<td>WA</td>
</tr>
<tr>
<td>2</td>
<td>s2</td>
<td>city1</td>
<td>WA</td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>city2</td>
<td>MA</td>
</tr>
<tr>
<td>4</td>
<td>s4</td>
<td>city2</td>
<td>MA</td>
</tr>
</tbody>
</table>
Integrity Constraints

- **Integrity constraint**
  - Condition specified on a database schema
  - Restricts data that can be stored in db instance

- DBMS enforces integrity constraints
  - Ensures only legal database instances exist

- Simplest form of constraint is domain constraint
  - Attribute values must come from attribute domain

Key Constraints

- **Key constraint**: “certain minimal subset of fields is a unique identifier for a tuple”

- **Candidate key**
  - Minimal set of fields
  - That uniquely identify each tuple in a relation

- **Primary key**
  - One candidate key can be selected as primary key

Foreign Key Constraints

- A relation can refer to a tuple in another relation

- **Foreign key**
  - Field that refers to tuples in another relation
  - Typically, this field refers to the primary key of other relation
  - Can pick another field as well

Key Constraint SQL Examples

```
CREATE TABLE Part {
  pno integer, 
pname varchar(20), 
psize integer, 
pcolor varchar(20), 
PRIMARY KEY (pno)
} ;
```

```
CREATE TABLE Supply( 
sno integer, 
pno integer, 
qty integer, 
price integer, 
PRIMARY KEY (pno)
);
```

```
CREATE TABLE Supply( 
sno integer, 
pno integer, 
qty integer, 
price integer,  
PRIMARY KEY (sno,pno)
);`
Key Constraint SQL Examples

CREATE TABLE Supply (
    sno integer,
    pno integer,
    qty integer,
    price integer,
    PRIMARY KEY (sno, pno),
    FOREIGN KEY (sno) REFERENCES Supplier,
    FOREIGN KEY (pno) REFERENCES Part
);

Key Constraint SQL Examples

CREATE TABLE Supply (
    sno integer,
    pno integer,
    qty integer,
    price integer,
    PRIMARY KEY (sno, pno),
    FOREIGN KEY (sno) REFERENCES Supplier
    ON DELETE NO ACTION,
    FOREIGN KEY (pno) REFERENCES Part
    ON DELETE CASCADE
);

General Constraints

• Table constraints serve to express complex constraints over a single table

CREATE TABLE Part (
    pno integer,
    pname varchar(20),
    psize integer,
    pcolor varchar(20),
    PRIMARY KEY (pno),
    CHECK (psize > 0)
);

Note: Also possible to create constraints over many tables

Relational Queries

• Query inputs and outputs are relations

• Query evaluation
  – Input: instances of input relations
  – Output: instance of output relation

Relational Algebra

• Query language associated with relational model

• Queries specified in an operational manner
  – A query gives a step-by-step procedure

• Relational operators
  – Take one or two relation instances as argument
  – Return one relation instance as result
  – Easy to compose into relational algebra expressions

Relational Operators

• Selection: \( \sigma_{\text{condition}}(S) \)
  – Condition is Boolean combination (\( \land, \lor \)) of terms
  – Term is: attr. \( \text{op} \) constant, attr. \( \text{op} \) attr.
  – Op is: \( <, \leq, =, \neq, \geq, > \), or >

• Projection: \( \pi_{\text{list-of-attributes}}(S) \)

• Union (\( \cup \)), Intersection (\( \cap \)), Set difference (\( \neg \)),

• Cross-product or cartesian product (\( \times \))

• Join: \( R \bowtie S = \sigma_{\text{condition}}(R \times S) \)

• Division: \( R / S \), Rename \( R(F) \rightarrow E \)
Selection & Projection Examples

\[ \sigma_{\text{disease} = \text{heart}}(\text{Patient}) \]

\[ \pi_{\text{zip}, \text{disease}}(\text{Patient}) \]

\[ \pi_{\text{zip}}(\sigma_{\text{disease} = \text{heart}}(\text{Patient})) \]

Relational Operators

- **Selection**: \( \sigma_{\text{condition}}(S) \)
  - Condition is Boolean combination (\( \land, \lor \)) of terms
  - Term is: attr. op constant, attr. op attr.
  - Op is: \(<, \leq, =, \neq, \geq, >\)

- **Projection**: \( \pi_{\text{list-of-attributes}}(S) \)

- **Union**: \( \cup \)
- **Intersection**: \( \cap \)
- **Set difference**: \(-\)
- **Cross-product or cartesian product**: \(\times\)
- **Join**: \( R \bowtie S = \sigma_{\theta}(R \times S) \)
- **Division**: \( R/S \), **Rename** \( \rho(R(F),E) \)

Cross-Product Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

Theta-Join Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

Equijoin Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>
Natural Join Example

AnonPatient P

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
</tbody>
</table>

Voters V

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

P \Join V

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
</tr>
</tbody>
</table>

More Joins

- **Outer join**
  - Include tuples with no matches in the output
  - Use NULL values for missing attributes

- **Variants**
  - Left outer join
  - Right outer join
  - Full outer join

Example of Algebra Queries

Q1: Names of patients who have heart disease

π name (Voter (σ disease='heart' (AnonPatient)))

Q2: Name of supplier of parts with size greater than 10

π sname (Supplier (σ psize > 10 (Part)))

Q3: Name of supplier of red parts or parts with size greater than 10

π sname (Supplier (σ psize > 10 (Part)) ∪ σ pcolor='red' (Part))

(Many more examples in the book)

Extended Operators of Relational Algebra

- **Duplicate elimination (δ)**
  - Since commercial DBMSs operate on multisets not sets

- **Aggregate operators (γ)**
  - Min, max, sum, average, count

- **Grouping operators (γ)**
  - Partitions tuples of a relation into “groups”
  - Aggregates can then be applied to groups

- **Sort operator (τ)**
Structured Query Language: SQL

- Influenced by relational calculus (see 344)
- Declarative query language
- Multiple aspects of the language
  - Data definition language
    - Statements to create, modify tables and views
  - Data manipulation language
    - Statements to issue queries, insert, delete data
  - More

SQL Query

Basic form: (plus many many more bells and whistles)

```
SELECT <attributes>
FROM <one or more relations>
WHERE <conditions>
```

Simple SQL Query

```
SELECT *
FROM Product
WHERE category='Gadgets'
```

Details

- Case insensitive:
  - Same: SELECT Select select
  - Same: Product product
  - Different: 'Seattle' 'seattle'
- Constants:
  - 'abc' - yes
  - "abc" - no

Eliminating Duplicates

```
SELECT DISTINCT category
FROM Product
```

```
SELECT category
FROM Product
```

Compare to:
### Ordering the Results

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

Ties are broken by the second attribute on the ORDER BY list, etc. Ordering is ascending, unless you specify the DESC keyword.

### Joins

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

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

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

### Tuple Variables

- **Person** (`pname`, `address`, `worksfor`)
- **Company** (`cname`, `address`)

Which address?

```sql
SELECT DISTINCT pname, address
FROM Person, Company
WHERE worksfor = cname
```

```sql
SELECT DISTINCT Person.pname, Company.address
FROM Person, Company
WHERE Person.worksfor = Company.cname
```

```sql
SELECT DISTINCT x.pname, y.address
FROM Person AS x, Company AS y
WHERE x.worksfor = y.cname
```

### Nested Queries

- **Nested query**
  - Query that has another query embedded within it
  - The embedded query is called a subquery

- Why do we need them?
  - Enables to refer to a table that must itself be computed

- Subqueries can appear in
  - WHERE clause (common)
  - FROM clause (less common)
  - HAVING clause (less common)

### Subqueries Returning Relations

- **Company** (`pname, city`)
- **Product** (`pname, maker`)
- **Purchase** (`id, product, buyer`)

Return cities where one can find companies that manufacture products bought by Joe Blow

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

You can also use:  
- `s > ALL R`
- `s > ANY R`
- `EXISTS R`

**Product** (`pname, price, category, maker`)

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

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

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

Movie (title, year, director, length)

Find movies whose title appears more than once.

Aggregation

SELECT avg(price)
FROM Product
WHERE maker = "Toyota"

SELECT count(*)
FROM Product
WHERE year > 1995

SQL supports several aggregation operations:
sum, count, min, max, avg

Except count, all aggregations apply to a single attribute

Grouping and Aggregation

SELECT S
FROM R_1, ..., R_n
WHERE C1
GROUP BY a_1, ..., a_k
HAVING C2

Conceptual evaluation steps:
1. Evaluate FROM-WHERE, apply condition C1
2. Group by the attributes a_1, ..., a_k
3. Apply condition C2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

Read more about it in the book...

Note: (1) scope of variables (2) this can still be expressed as single SFW