Lecture 4: Updates, Views, Constraints and Other Fun Features

Monday, April 19th, 2004

Agenda

• Nulls and outerjoins
• Creating and updating schemas
• Views: updating and reusing them
• Constraints
• Programming with SQL
• Relational algebra

Null Values and Outerjoins

Explicit joins in SQL:

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

Same as:

But Products that never sold will be lost!

SELECT Product.name, Purchase.store
FROM Product JOIN Purchase ON
Product.name = Purchase.prodName

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

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

But Products that never sold will be lost!

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
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:

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

Example: Insert a new purchase to the database:

\[
\text{INSERT INTO PRODUCT(name) VALUES (v1, \ldots, vn)}
\]

Example: Insert a new purchase to the database:

\[
\text{INSERT INTO PRODUCT(name) SELECT DISTINCT Purchase.product FROM Purchase WHERE Purchase.date > '10/06/01'}
\]

Example: Insert a new purchase to the database:

\[
\text{INSERT INTO Product(name) SELECT DISTINCT prodName FROM Purchase WHERE prodName NOT IN (SELECT name FROM Product)}
\]

Example: Insert a new purchase to the database:

\[
\text{INSERT INTO Product(name, listPrice) VALUES (v1, \ldots, vn)}
\]

Suppose database got corrupted and we need to fix it:

Task: Insert all prodNames from Purchase into Product

- Case 1:
  - Depends on the implementation.
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 Productname IN
    (SELECT product
    FROM Purchase
    WHERE Date = 'Oct, 25, 1999');
```

Data Definition in SQL

So far we have seen the Data Manipulation Language, DML
Next: Data Definition Language (DDL)

Data types:
- Defines the types.

Data definition: defining the schema.

- Create tables
- Delete tables
- Modify table schema

Indexes: to improve performance

Data Types in SQL

- Characters:
  - CHAR(20) — fixed length
  - VARCHAR(40) — variable length
- Numbers:
  - INT, REAL plus variations
- Times and dates:
  - DATE, DATETIME (SQL Server only)
- To reuse domains:
  CREATE DOMAIN address AS VARCHAR (55)

Creating Tables

Example:

```
CREATE TABLE Person(
    name VARCHAR(30),
    social-security-number INT,
    age SHORTINT,
    city VARCHAR(30),
    gender BIT(1),
    Birthdate DATE
);
```

Deleting or Modifying a Table

Deleting:

Example: DROP Person

Exercise with care!!

Altering (adding or removing an attribute):

```
ALTER TABLE Person
ADD phone CHAR(16);
```

ALTER TABLE Person
DROP age;

What happens when you make changes to the schema?
Default Values

Specifying default values:

```
CREATE TABLE Person(
    name VARCHAR(30),
    social-security-number INT,
    age SHORTINT DEFAULT 100,
    city VARCHAR(30) DEFAULT 'Seattle',
    gender CHAR(1) DEFAULT '?',
    Birthdate DATE
);
```

The default of defaults: NULL

Indexes

REALLY Important to speed up query processing time.

Suppose we have a relation

```
Person (name, age, city)
```

Sequential scan of the file Person may take long

```
SELECT * FROM Person
WHERE name = 'Smith'
```

Create an index on name:

```
CREATE INDEX nameIndex ON Person(name);
```

• B+ trees have fan-out of 100s: max 4 levels!

Indexes can be created on more than one attribute:

```
CREATE INDEX doubleindex ON Person(age, city);
```

Example:

```
SELECT *
FROM Person
WHERE age = 55 AND city = 'Seattle'
```

Helps in:

```
SELECT *
FROM Person
WHERE age > 25 AND age < 28
```

But not in:

```
SELECT *
FROM Person
WHERE city = 'Seattle'
```

Creating Indexes

Indexes can be useful in range queries too:

```
CREATE INDEX ageIndex ON Person(age);
```

B+ trees help in:

```
SELECT *
FROM Person
WHERE age > 25 AND age < 28
```

Why not create indexes on everything?
Agenda

- Nulls and outer joins
- Creating and updating schema
  - Views: updating and reusing them
  - Constraints
- Programming with SQL
- Relational algebra

Defining Views

Views are relations, except that they are not physically stored.

For presenting different information to different users

Employee (ssn, name, department, project, salary)

```
CREATE VIEW Developers AS
    SELECT name, project
    FROM Employee
    WHERE department = "Development"
```

Payroll has access to Employee, others only to Developers

A Different View

Person (name, city)
Purchase (buyer, seller, product, store)
Product (name, maker, category)

We have a new virtual table:
Seattle-view (buyer, seller, product, store)

```
CREATE VIEW Seattle-view AS
    SELECT buyer, seller, product, store
    FROM Person, Purchase
    WHERE Person.city = "Seattle" AND Person.name = Purchase.buyer
```

Using a View

```
SELECT name, store
FROM Seattle-view, Product
WHERE Seattle-view.product = Product.name AND Product.category = "shoes"
```

We can later use the view:

```
SELECT name, Purchase.store
FROM Person, Purchase, Product
WHERE Person.city = "Seattle" AND Person.name = Purchase.buyer AND Purchase.product = Product.name AND Product.category = "shoes"
```

What Happens When We Query a View?

```
SELECT name, Seattle-view.store
FROM Seattle-view, Product
WHERE Seattle-view.product = Product.name AND Product.category = "shoes"
```

```
SELECT name, Purchase.store
FROM Person, Purchase, Product
WHERE Person.city = "Seattle" AND Person.name = Purchase.buyer AND Purchase.product = Product.name AND Product.category = "shoes"
```

Types of Views

- Virtual views:
  - Used in databases
  - Computed only on-demand and - slower at runtime
  - Always up to date
- Materialized views
  - Used in data warehouses
  - Faster at runtime
  - May have stale data
Updating Views

How can I insert a tuple into a table that doesn't exist?

Employee(ssn, name, department, project, salary)

CREATE VIEW Developers AS
SELECT name, project
FROM Employee
WHERE department = "Development"

INSERT INTO Developers
VALUES("Joe", "Optimizer")

INSERT INTO Employee
VALUES(NULL, "Joe", "Development", "Optimizer", NULL)

If we make the following insertion:

It becomes:

Non-Updatable Views

CREATE VIEW Seattle-view AS
SELECT seller, product, store
FROM Person, Purchase
WHERE Person.city = "Seattle" AND Person.name = Purchase.buyer

How can we add the following tuple to the view?

We need to add "Joe" to Person first, but we don't have all its attributes

Answering Queries Using Views

• What if we want to use a set of views to answer a query.
• Why?
  - The obvious reason...
  - Answering queries over web data sources.
• Very cool stuff! (i.e., I did a lot of research on this).

Reusing a Materialized View

• Suppose I have only the result of Seattle-view:
  SELECT buyer, seller, product, store
  FROM Person, Purchase
  WHERE Person.city = "Seattle" AND Person.name = Purchase.buyer

  and I want to answer the query
  SELECT buyer, seller
  FROM Person, Purchase
  WHERE Person.city = "Seattle" AND Person.name = Purchase.buyer AND Purchase.product = "gizmo".

  Then, I can rewrite the query using the view.

Another Example

• I still have only the result of Seattle-view:
  SELECT buyer, seller, product, store
  FROM Person, Purchase
  WHERE Person.city = "Seattle" AND Person.name = Purchase.buyer

  but I want to answer the query
  SELECT buyer, seller
  FROM Person, Purchase
  WHERE Person.city = "Seattle" AND Person.name = Purchase.buyer AND Purchase.product = "gizmo".

Query Rewriting Using Views

Rewritten query:
SELECT buyer, seller
FROM Seattle-view
WHERE product = "gizmo"

Original query:
SELECT buyer, seller
FROM Person, Purchase
WHERE Person.city = "Seattle" AND Person.name = Purchase.buyer AND Purchase.product = "gizmo".
And Now?

- I still have only the result of SeattleView:
  
  $\text{SELECT buyer, seller, product, store}$
  
  $\text{FROM Person, Purchase, Product}$
  
  $\text{WHERE Person.city = 'Seattle' AND}$
  
  $\text{Person.person = Purchase.buyer AND}$
  
  $\text{Purchase.product = Product.name}$

- But I want to answer the query
  
  $\text{SELECT buyer, seller}$
  
  $\text{FROM Person, Purchase}$
  
  $\text{WHERE Person.city = 'Seattle' AND}$
  
  $\text{Person.person = Purchase.buyer}$. 

And what if it’s the other way around?

Finally...

- I still have only the result of:
  
  $\text{SELECT seller, buyer, \text{Sum(Price)}}$
  
  $\text{FROM Purchase}$
  
  $\text{WHERE Purchase.store = 'The Bon'}$
  
  $\text{Group By seller, buyer}$

- But I want to answer the query
  
  $\text{SELECT seller, \text{Sum(Price)}}$
  
  $\text{FROM Purchase}$
  
  $\text{WHERE Person.store = 'The Bon'}$
  
  $\text{Group By seller}$. 

The General Problem

- Given a set of views $V_1, \ldots, V_n$, and a query $Q$, can we answer $Q$ using only the answers to $V_1, \ldots, V_n$?

- Why do we care?
  
  - We can answer queries more efficiently.
  
  - We can query data sources on the WWW in a principled manner.

- Many, many papers on this problem.


Querying the WWW

- Assume a virtual schema of the WWW, e.g.,
  
  $\text{Course(number, university, title, prof, quarter)}$

- Every data source on the web contains the answer to a view over the virtual schema:
  
  $\text{UW database: SELECT number, title, prof}$
  
  $\text{FROM Course}$
  
  $\text{WHERE university = 'UW' AND quarter = '2/02'}$

  $\text{Stanford database: SELECT number, title, prof, quarter}$
  
  $\text{FROM Course}$
  
  $\text{WHERE university = 'Stanford'}$

- User query: find all professors who teach "database systems"

Agenda

- Nulls and outer-joins
- Creating and updating schem as
- Views: updating and reusing them
- Constraints
- Programming with SQL
- Relational algebra
Constraints in SQL

- A constraint = a property that we’d like our database to hold
- The system will enforce the constraint by taking some actions:
  - forbid an update
  - or perform compensating updates

Constraints in SQL:
- Keys, foreign keys
- Attribute-level constraints
- Tuple-level constraints
- Global constraints: assertions

The more complex the constraint, the harder it is to check and to enforce

Keys

```
CREATE TABLE Product (
    name CHAR(30) PRIMARY KEY,
    category VARCHAR(20))
```

OR:

```
CREATE TABLE Product (
    name CHAR(30),
    category VARCHAR(20),
    price INT,
    PRIMARY KEY (name, category))
```

Keys with Multiple Attributes

```
CREATE TABLE Product (
    name CHAR(30),
    category VARCHAR(20),
    PRIMARY KEY (name, category))
```

Other Keys

```
CREATE TABLE Product (
    productID CHAR(10),
    name CHAR(30),
    category VARCHAR(20),
    price INT,
    PRIMARY KEY (productID),
    UNIQUE (name, category))
```

Foreign Key Constraints

```
CREATE TABLE Purchase (
    prodName CHAR(30),
    date DATETIME)
```

prodName is a foreign key to Product(name)

There is at most one PRIMARY KEY; there can be many UNIQUE

prodName is a foreign key to Product(name)
name must be a key in Product.
### Foreign Key Constraints

- OR

```sql
CREATE TABLE Purchase (
    prodName CHAR(30),
    category VARCHAR(20),
    date DATETIME,
    FOREIGN KEY (prodName, category) REFERENCES Product(name, category)
)
```

• (name, category) must be a PRIMARY KEY

### What happens during updates?

**Types of updates:**
- In Purchase: insert/update
- In Product: delete/update

### What happens during updates?

- SQL has three policies for maintaining referential integrity:
  - **Reject** violating modifications (default)
  - **Cascade:** after a delete/update do a delete/update
  - **Set-null** set foreign-key field to NULL

**Reading Assignment:** 7.1.5, 7.1.6

### Constraints on Attributes and Tuples

- **Constraints on attributes:**
  - **NOT NULL** — obvious meaning...
  - **CHECK** condition — any condition!
- **Constraints on tuples**
  - **CHECK** condition
**General Assertions**

```sql
CREATE ASSERTION myAssertion
CHECK

NOT EXISTS(
    SELECT Product.name
    FROM Product, Purchase
    WHERE Product.name = Purchase.prodName
    GROUP BY Product.name
    HAVING count(*) > 200)
```

---

**Final Comments on Constraints**

- Can give them names, and alter later
  - Read in the book.
- We need to understand exactly when they are checked
- We need to understand exactly what actions are taken if they fail

---

**Triggers**

Enable the database program to specify:
- When to check a constraint,
- What exactly to do.

A trigger has 3 parts:
- An event (e.g., update to an attribute)
- A condition (e.g., a query to check)
- An action (deletion, update, insertion)

When the event happens, the system will check the constraint, and if satisfied, will perform the action.

**NOTE:** Triggers may cause cascading effects. Database vendors did not wait for standards with triggers!

---

**Elements of Triggers (in SQL3)**

- Timing of action execution: before, after or instead of triggering event.
- The action can refer to both the old and new state of the database.
- Update events may specify a particular column or set of columns.
- A condition is specified with a WHEN clause.
- The action can be performed either for:
  - once for every tuple, or
  - once for all the tuples that are changed by the database operation.

---

**Example: Row Level Trigger**

```sql
CREATE TRIGGER NoLowerPrices
AFTER UPDATE OF price ON Product
REFERENCING
OLD AS OldTuple
NEW AS NewTuple
WHEN (OldTuple.price > NewTuple.price)
UPDATE Product
SET price = OldTuple.price
WHERE name = NewTuple.name
FOR EACH ROW
```

---

**Statement Level Trigger**

```sql
CREATE TRIGGER average-price-preserve
INSTEAD OF UPDATE OF price ON Product
REFERENCING
OLD_TABLE AS OldStuff
NEW_TABLE AS NewStuff
WHEN (1000 < (SELECT AVG (price)
FROM ((Product EXCEPT OldStuff) UNION NewStuff))
DELETE FROM Product
WHERE (name, price, company) IN OldStuff;
INSERT INTO Product
(SELECT * FROM NewStuff)
```
Bad Things Can Happen

CREATE TRIGGER Bad-trigger
AFTER UPDATE OF price IN Product
REFERENCING OLD AS OldTuple
    NEW AS NewTuple
WHEN (NewTuple.price > 50)
UPDATE Product
SET price = NewTuple.price * 2
WHERE name = NewTuple.name
FOR EACH ROW

Agenda

- Nulls and outerjoins
- Creating and updating schema
- Views: updating and reusing them
- Constraints
  - Programming with SQL
  - Relational algebra

Embedded SQL

- Direct SQL (= ad-hoc SQL) is rarely used
- In practice: SQL is embedded in some application code
- SQL code is identified by special syntax

Impedance Mismatch

- Example: SQL in C:
  - C uses int, char[], pointers, etc.
  - SQL uses tables
- Impedance mismatch = incompatible types

The Impedance Mismatch Problem

Why not use only one language?

- Forgetting SQL: "We can quickly dispense with this idea" [textbook, pg. 351].
- SQL cannot do everything that the host language can do.

Solution: use cursors

Programs with Embedded SQL

Host language + Embedded SQL
Preprocessor
Host Language + function calls
Call level interface (CLI)
ODBC, JDBC, ADO
Host Language compiler
Host Language program
Interface: SQL / Host Language

Values get passed through shared variables.

Colons precede shared variables when they occur within the SQL statements.

EXEC SQL: precedes every SQL statement in the host language.

The variable SQLSTATE provides error messages and status reports (e.g., "00000" says that the operation completed with no problem).

EXEC SQL BEGIN DECLARE SECTION;
char productName[30];
EXEC SQL END DECLARE SECTION;

Example

Product (name, price, quantity, make)
Purchase (buyer, seller, store, name)
Company (name, city)
Person (name, phone, city)

Using Shared Variables

Example

void simpleInsert() {  
EXEC SQL BEGIN DECLARE SECTION;  
char n[20], c[30]; /* product-name, company-name */  
int p, q; /* price, quantity */  
char SQLSTATE[6];  
EXEC SQL END DECLARE SECTION;  
/* get values for name, price and company somehow */  
EXEC SQL INSERT INTO Product (name, price, quantity, make) VALUES (:n, :p, :q, :c);  
}

Single-Row Select Statements

int getPrice(char*name) {  
EXEC SQL BEGIN DECLARE SECTION;  
char n[20];  
int p;  
char SQLSTATE[6];  
EXEC SQL END DECLARE SECTION;  
strcpy(n, name); /* copy name to local variable */  
EXEC SQL SELECT price INTO:p FROM Product WHERE Product.name = :n;  
return p;  
}

Cursors

1. Declare the cursor
2. Open the cursor
3. Fetch tuples one by one
4. Close the cursor

void product2XML() {  
EXEC SQL BEGIN DECLARE SECTION;  
char n[20], c[30];  
int p, q;  
char SQLSTATE[6];  
EXEC SQL END DECLARE SECTION;  
EXEC SQL DECLARE crs CURSOR FOR SELECT name, price, quantity, make FROM Product WHERE Product.name = :n;  
EXEC SQL OPEN crs;  
...}
Cursors

```c
printf("<allProducts>
");  
while(1) {
  EXEC SQL FETCH FROM crs INTO :n, :p, :q, :c;
  if (NO_MORE_TUPLES) break;
  printf("<product>
");
  printf("<name>      %s     </name>
",      n);
  printf("<price>       %d    </price>
",      p);
  printf("<quantity>  %d    </quantity>
", q);
  printf("<maker>     %s    </maker>
",     c);
  printf("</product>
");
}  
EXEC SQL CLOSE crs;
printf("</allProducts>
");  
}
```

What is NO_MORE_TUPLES?

```c
#define NO_MORE_TUPLES !(strcmp(SQLSTATE, "02000"))
```

More on Cursors

- Cursors can modify a relation as well as read it.
- We can determine the order in which the cursor will get tuples by the ORDER BY key in the SQL query.
- Cursors can be protected against changes to the underlying relations.
- The cursor can be a scrolling one: can go forward, backward, +n, -n, Abs(n), Abs(-n).

Agenda

- NULLs and outer joins
- Creating and updating schemas
- Views: updating and reusing them
- Constraints
- Programming with SQL
- Relational algebra

Relational Algebra

- Formalism for creating new relations from existing ones
- Its place in the big picture:

```plaintext
Declarative query language
  A language
    Implementation

SQL, relational calculus
  Relational algebra
  Relational bag algebra
```

Relational Algebra

- Five operators:
  - Union:
  - Difference:
  - Selection:
  - Projection:
  - Cartesian Product:
- Derived or auxiliary operators:
  - Intersection, complement
  - Joins (natural, equi-join, theta join, semi-join)
  - Renaming:

```c
```
1. **Union and Difference**

- $R_1 \cup R_2$
- Example:
  - ActiveEmployees $\cup$ RetiredEmployees

- $R_1 - R_2$
- Example:
  - AllEmployees $-$ RetiredEmployees

What about Intersection?

- It is a derived operator
- $R_1 \cap R_2 = R_1 - (R_1 - R_2)$
- Also expressed as a join (we’ll see later)
- Example:
  - UnionizedEmployees $\cap$ RetiredEmployees

3. **Selection**

- Returns all tuples which satisfy a condition
- Notation: $s_c(R)$
- Examples:
  - $s_{\text{Salary} > 40,000}(\text{Employee})$
  - $s_{\text{Name} = \"Smith\"}(\text{Employee})$
- The condition $c$ can be $=, <, \leq, >, \geq, \neq$

Selection Example

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>DepartmentID</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>999999999</td>
<td>John</td>
<td>1</td>
<td>30,000</td>
</tr>
<tr>
<td>777777777</td>
<td>Tony</td>
<td>1</td>
<td>32,000</td>
</tr>
<tr>
<td>888888888</td>
<td>Alice</td>
<td>2</td>
<td>45,000</td>
</tr>
</tbody>
</table>

Find all employees with salary more than $40,000.

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>DepartmentID</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>888888888</td>
<td>Alice</td>
<td>2</td>
<td>45,000</td>
</tr>
</tbody>
</table>

4. **Projection**

- Eliminates columns, then removes duplicates
- Notation: $P_{A_1, A_2, \ldots, A_n}(R)$
- Example: project social-security number and names:
  - $P_{\text{SSN}, \text{Name}}(\text{Employee})$
  - Output schema: Answer(SSN, Name)

Projection Example

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>DepartmentID</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>999999999</td>
<td>John</td>
<td>1</td>
<td>30,000</td>
</tr>
<tr>
<td>777777777</td>
<td>Tony</td>
<td>1</td>
<td>32,000</td>
</tr>
<tr>
<td>888888888</td>
<td>Alice</td>
<td>2</td>
<td>45,000</td>
</tr>
</tbody>
</table>

$P_{\text{SSN}, \text{Name}}(\text{Employee})$
5. Cartesian Product

- Each tuple in R1 with each tuple in R2
- Notation: R1 x R2
- Example:
  - Employee x Dependents
- Very rare in practice; mainly used to express joins

<table>
<thead>
<tr>
<th>Employee</th>
<th>SSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>999999999</td>
</tr>
<tr>
<td>Tony</td>
<td>777777777</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependents</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emily</td>
<td></td>
</tr>
<tr>
<td>Joe</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employee x Dependents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>John</td>
</tr>
<tr>
<td>John</td>
</tr>
<tr>
<td>Tony</td>
</tr>
<tr>
<td>Tony</td>
</tr>
</tbody>
</table>

6. Relational Algebra

- Five operators:
  - Union: ∪
  - Difference: -
  - Selection: s
  - Projection: P
  - Cartesian Product: ·
- Derived or auxiliary operators:
  - Intersection, complement
  - Joins (natural, equi-join, theta join, semi-join)
  - Renaming: r

7. Renaming

- Changes the schema, not the instance
- Notation: rB1, ..., Bn (R)
- Example:
  - rLastName, SocSocNo(Employee)
  - Output schema: Answer(LastName, SocSocNo)

8. Natural Join

- Notation: R1 ⋈ R2
- Meaning: R1 ⋈ R2 = Pₜ (s_c (R1 x R2))
- Where:
  - The selection s_c checks equality of all common attributes
  - The projection eliminates the duplicate common attributes

<table>
<thead>
<tr>
<th>Employee</th>
<th>SSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>999999999</td>
</tr>
<tr>
<td>Tony</td>
<td>777777777</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lastname, SocSocNo (Employee)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lastname</td>
</tr>
<tr>
<td>John</td>
</tr>
<tr>
<td>Tony</td>
</tr>
</tbody>
</table>
Natural Join Example

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>99999999</td>
</tr>
<tr>
<td>Tony</td>
<td>77777777</td>
</tr>
</tbody>
</table>

Dependents

<table>
<thead>
<tr>
<th>SSN</th>
<th>Dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>99999999</td>
<td>Emi ly</td>
</tr>
<tr>
<td>77777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>

Employee $\bowtie$ Dependents = $P_{\text{SSN}=\text{SSN}_2}(\text{Employee} \times r_{\text{SSN}_2, \text{Dname}}(\text{Dependents}))$

Employee $\bowtie$ Dependents = $P_{\text{SSN}=\text{SSN}_2}(\text{Employee} \times \text{Dependents})$

Natural Join

$\text{S}=\begin{bmatrix} A & B & C \\ X & Y & Z \end{bmatrix}$

$\text{R} = \begin{bmatrix} A & B & C \\ X & Y & Z \end{bmatrix}$

$\text{R} \bowtie \text{S} = \begin{bmatrix} A & B & C \\ X & Y & Z \\ Z & V & W \end{bmatrix}$

Theta Join

Given the schema as $R(A, B, C, D), S(A, C, E)$, what is the schema of $R \bowtie S$?

Given $R(A, B, C), S(D, E)$, what is $R \bowtie S$?

Given $R(A, B), S(A, B)$, what is $R \bowtie S$?

Eq-join

A theta join where $q$ is an equality

$R_1 \bowtie_{A=B} R_2 = s_{A=B} (R_1 \times R_2)$

Example:

- Employee $\bowtie_{\text{SSN}=\text{SSN}_2} \text{Dependents}$

- Most useful join in practice

Semijoin

$R \bowtie S = P_{A_1, A_n \in R} (R \bowtie S)$

Where $A_1, \ldots, A_n$ are the attributes in $R$

Example:

- Employee $\bowtie \text{Dependents}$
Semijoins in Distributed Databases

- Semijoins are used in distributed databases.

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>Dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Employee \( \bowtie \) ssn=ssn(s age>71 (Dependents))

\( T = P \bowtie s \text{ age}>71 (\text{Dependents}) \)
\( R = \text{Employee} \bowtie T \)
\( \text{Answer} = R \bowtie \text{Dependents} \)

Complex RA Expressions

Operations on Bags

- A bag = a set with repeated elements.
- All operations need to be defined carefully on bags.
  - \( \{a,b,c\} \cap \{a,b,b,a,f\} = \{a,b,b,a,f\} \)
  - \( \{a,b,b,f\} - \{b,c,c,a,d\} = \{a,b,b,d\} \)
  - \( s \circ R \): preserve the number of occurrences.
  - \( P \circ (\cdot) \): no duplicate elimination.
  - Cartesian product, join: no duplicate elimination.
- Important: Relational Engines work on bags, not sets!

Reading assignment: 5.3 - 5.4

Finally: RA has Limitations!

- Cannot compute "transitive closure".

<table>
<thead>
<tr>
<th>Relationship</th>
<th>Name 1</th>
<th>Name 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Father</td>
<td>Fred</td>
<td>Mary</td>
</tr>
<tr>
<td>Spouse</td>
<td>Mary</td>
<td>Bill</td>
</tr>
<tr>
<td>Sister</td>
<td>Lou</td>
<td>Nancy</td>
</tr>
<tr>
<td>Cousin</td>
<td>Joe</td>
<td>Mary</td>
</tr>
</tbody>
</table>

- Find all direct and indirect relatives of Fred.
- Cannot express in RA!!! Need to write C program.