Relational Schema Design (end)
Relational Algebra
SQL (maybe)
April 18th, 2002

Boyce-Codd Normal Form

A simple condition for removing anomalies from relations:

A relation R is in BCNF if and only if:

Whenever there is a nontrivial dependency \( A_1, A_2, \ldots, A_n \rightarrow B \)
for \( R \), it is the case that \( \{ A_1, A_2, \ldots, A_n \} \)
a super-key for \( R \).

In English (though a bit vague):

Whenever a set of attributes of \( R \) is determining another attribute, should determine all the attributes of \( R \).

---

Example

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>Phone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>123-321-99</td>
<td>(201) 555-1234</td>
</tr>
<tr>
<td>Fred</td>
<td>123-321-99</td>
<td>(206) 572-4312</td>
</tr>
<tr>
<td>Joe</td>
<td>909-438-44</td>
<td>(908) 464-0028</td>
</tr>
<tr>
<td>Joe</td>
<td>909-438-44</td>
<td>(212) 555-4000</td>
</tr>
</tbody>
</table>

What are the dependencies?

SSN \( \rightarrow \) Name

What are the keys?

Is it in BCNF?

---

Decompose it into BCNF

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-321-99</td>
<td>Fred</td>
</tr>
<tr>
<td>909-438-44</td>
<td>Joe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>Phone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-321-99</td>
<td>(201) 555-1234</td>
</tr>
<tr>
<td>123-321-99</td>
<td>(206) 572-4312</td>
</tr>
<tr>
<td>909-438-44</td>
<td>(908) 464-0028</td>
</tr>
<tr>
<td>909-438-44</td>
<td>(212) 555-4000</td>
</tr>
</tbody>
</table>

---

BCNF Decomposition

Find a dependency that violates the BCNF condition:

\( A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m \)

Heuristics: choose \( B_1, B_2, \ldots, B_m \) “as large as possible”

Decompose:

Find a 2-attribute relation that is not in BCNF.

Continue until there are no BCNF violations left.

Correct Decompositions

A decomposition is lossless if we can recover:

\[ R(A,B,C) \]

\[ \{ R1(A,B), R2(A,C) \} \]

\[ R'(A,B,C) = R(A,B,C) \]

\( R' \) is in general larger than \( R \). Must ensure \( R' = R \)
Decomposition Based on BCNF is Necessarily Lossless

R(A, B, C), A → C

BCNF: R1(A,B), R2(A,C)

Some tuple (a,b,c) in R decomposes into (a,b) in R1 and (a,c) in R2
(a,b') also in R1
(a,c') also in R2

Recover tuples in R: (a,b,c), (a,b,c'), (a,b',c), (a,b',c') also in R?

Can (a,b,c') be a bogus tuple? What about (a,b',c')?

3NF: A Problem with BCNF

<table>
<thead>
<tr>
<th>Unit</th>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galaga99</td>
<td>UW</td>
<td>Database</td>
</tr>
<tr>
<td>Bingo</td>
<td>UW</td>
<td>Database</td>
</tr>
</tbody>
</table>

FD's: Unit → Company, Company → Unit

So, there is a BCNF violation, and we decompose.

Unit | Company  | Unit → Company
|------|----------|-----------------|
| Unit | Product  | No FDs

So What’s the Problem?

<table>
<thead>
<tr>
<th>Unit</th>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galaga99</td>
<td>UW</td>
<td>Database</td>
</tr>
<tr>
<td>Bingo</td>
<td>UW</td>
<td>Database</td>
</tr>
</tbody>
</table>

No problem so far. All local FD's are satisfied.

Let’s put all the data back into a single table again:

<table>
<thead>
<tr>
<th>Unit</th>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galaga99</td>
<td>UW</td>
<td>Database</td>
</tr>
<tr>
<td>Bingo</td>
<td>UW</td>
<td>Database</td>
</tr>
</tbody>
</table>

Violates the dependency: Company, Product -> Unit!

Solution: 3rd Normal Form (3NF)

A simple condition for removing anomalies from relations:

A relation R is in 3rd normal form if:

\( A_1, A_2, \ldots, A_n \rightarrow B \) for R, then \( \{ A_1, A_2, \ldots, A_n \} \) a super-key for R, or B is part of a key.

Multi-valued Dependencies

<table>
<thead>
<tr>
<th>SSN</th>
<th>Phone Number</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-321-99</td>
<td>(206) 572-4312</td>
<td>CSE-444</td>
</tr>
<tr>
<td>123-321-99</td>
<td>(206) 572-4312</td>
<td>CSE-444</td>
</tr>
<tr>
<td>123-321-99</td>
<td>(206) 432-8954</td>
<td>CSE-341</td>
</tr>
<tr>
<td>123-321-99</td>
<td>(206) 432-8954</td>
<td>CSE-341</td>
</tr>
</tbody>
</table>

The multi-valued dependencies are:

<table>
<thead>
<tr>
<th>SSN</th>
<th>Phone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>Course</td>
</tr>
</tbody>
</table>
Definition of MVDs Continued
Equivalently: the decomposition into
\[ R_1(A_1,\ldots,A_n,B_1,\ldots,B_m), \quad R_2(A_1,\ldots,A_n,C_1,\ldots,C_p) \]
is lossless
Note: an MVD \[ A_1,\ldots,A_n \rightarrow B_1,\ldots,B_m \]
Implicitly talks about “the other” attributes \[ C_1,\ldots,C_p \]
Rules for MVDs
If \[ A_1,\ldots,A_n \rightarrow B_1,\ldots,B_m \]
then \[ A_1,\ldots,A_n \rightarrow B_1,\ldots,B_m \]
Other rules in the book

4th Normal Form (4NF)

Confused by Normal Forms?

If a database doesn’t violate 4NF (BCNF) then it
doesn’t violate BCNF (3NF)!

Querying the Database

Find all the employees who earn more than
$50,000 and pay taxes in New Jersey.
We don’t want to write a program for each
query.
We design high-level query languages:
- SQL (used everywhere)
- Datalog (used by database theoreticians, their
students, friends and family)
- Relational algebra: a basic set of operations on
relations that provide the basic principles.

Querying the Database

Same as BCNF with FDs replaced by MVDs
Relational Algebra at a Glance

- Operators: relations as input, new relation as output
- Five basic RA operators:
  - Basic Set Operators
    - union, difference (no intersection, no complement)
  - Selection: $\sigma$
  - Projection: $\pi$
  - Cartesian Product: $\times$
- Derived operators:
  - Intersection, complement
  - Joins (natural, equi-join, theta join, semi-join)
- When our relations have attribute names:
  - Renaming: $\rho$

Set Operations

- Binary operations
- Union, difference, intersection
  - Intersection can be expressed in other ways

Set Operations: Union

- Union: all tuples in R1 or R2
- Notation: R1 $\cup$ R2
- R1, R2 must have the same schema
- R1 $\cup$ R2 has the same schema as R1, R2
- Example:
  - ActiveEmployees $\cup$ RetiredEmployees

Set Operations: Difference

- Difference: all tuples in R1 and not in R2
- Notation: R1 $-$ R2
- R1, R2 must have the same schema
- R1 $-$ R2 has the same schema as R1, R2
- Example
  - AllEmployees $-$ RetiredEmployees

Set Operations: Intersection

- Intersection: all tuples both in R1 and in R2
- Notation: R1 $\cap$ R2
- R1, R2 must have the same schema
- R1 $\cap$ R2 has the same schema as R1, R2
- Example
  - UnionizedEmployees $\cap$ RetiredEmployees

Selection

- Returns all tuples which satisfy a condition
- Notation: $\sigma_c(R)$
- $c$ is a condition: $=, <, >, and, or, not$
- Output schema: same as input schema
- Find all employees with salary more than $40,000$:
  - $\sigma_{\text{Salary} > 40000}(\text{Employee})$
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.

σ_{Salary > 40000}(Employee)

Projection

- Unary operation: returns certain columns
- Eliminates duplicate tuples!
- Notation: \( \Pi_{A_1, \ldots, A_n}(R) \)
- Input schema \( R(B_1, \ldots, B_m) \)
- Condition: \( \{A_1, \ldots, A_n\} \subseteq \{B_1, \ldots, B_m\} \)
- Output schema \( S(A_1, \ldots, A_n) \)
- Example: project social-security number and names:
  - \( \Pi_{SSN, Name}(Employee) \)

Projection Example

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>999999999</td>
<td>John</td>
</tr>
<tr>
<td>777777777</td>
<td>Tony</td>
</tr>
<tr>
<td>888888888</td>
<td>Alice</td>
</tr>
</tbody>
</table>

Proyection Example

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>999999999</td>
<td>John</td>
</tr>
<tr>
<td>777777777</td>
<td>Tony</td>
</tr>
<tr>
<td>888888888</td>
<td>Alice</td>
</tr>
</tbody>
</table>


Cartesian Product

- Each tuple in \( R_1 \) with each tuple in \( R_2 \)
- Notation: \( R_1 \times R_2 \)
- Input schemas \( R_1(A_1, \ldots, A_n), R_2(B_1, \ldots, B_m) \)
- Condition: \( \{A_1, \ldots, A_n\} \cap \{B_1, \ldots, B_m\} = \emptyset \)
- Output schema is \( S(A_1, \ldots, A_n, B_1, \ldots, B_m) \)
- Notation: \( R_1 \times R_2 \)
- Example: \( Employee \times Dependents \)
- Very rare in practice; but joins are very common

Cartesian Product Example

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

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Dname</th>
</tr>
</thead>
<tbody>
<tr>
<td>999999999</td>
<td>John</td>
<td>Emily</td>
</tr>
<tr>
<td>777777777</td>
<td>Tony</td>
<td>Joe</td>
</tr>
</tbody>
</table>

Renaming

- Does not change the relational instance
- Changes the relational schema only
- Notation: \( \rho_{B_1, \ldots, B_n}(R) \)
- Input schema: \( R(A_1, \ldots, A_n) \)
- Output schema: \( S(B_1, \ldots, B_n) \)
- Example:
  - \( \rho_{LastName, SocSecNo}(Employee) \)
### Renaming Example

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

\[ \rho_{\text{LastName, SocSecNo}}(\text{Employee}) \]

<table>
<thead>
<tr>
<th>LastName</th>
<th>SocSecNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>999999999</td>
</tr>
<tr>
<td>Tony</td>
<td>777777777</td>
</tr>
</tbody>
</table>

### Derived Operations

- Intersection is derived:
  - \( R_1 \cap R_2 = R_1 - (R_1 - R_2) \) why?
  - There is another way to express it (later)
- Most importantly: joins, in many variants

### Natural Join

- Notation: \( R_1 \bowtie R_2 \)
- Input Schema: \( R_1(A_1, \ldots, A_n), R_2(B_1, \ldots, B_m) \)
- Output Schema: \( S(C_1, \ldots, C_p) \)
  - Where \( \{C_1, \ldots, C_p\} = \{A_1, \ldots, A_n\} \cup \{B_1, \ldots, B_m\} \)
- Meaning: combine all pairs of tuples in \( R_1 \) and \( R_2 \) that agree on the attributes:
  - \( \{A_1, \ldots, A_n\} \cap \{B_1, \ldots, B_m\} \) (called the join attributes)
- Equivalent to a cross product followed by selection
- Example \( \text{Employee} \bowtie \text{Dependents} \)

### Natural Join Example

<table>
<thead>
<tr>
<th>Employee</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>SSN</td>
</tr>
<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></th>
</tr>
</thead>
<tbody>
<tr>
<td>SSN</td>
<td>DName</td>
</tr>
<tr>
<td>999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>

\[ \text{Employee} \bowtie \text{Dependents} = \Pi_{\text{Name, SSN, DName}} (\sigma_{\text{SSN}=\text{SSN}_2} (\text{Employee} \bowtie \rho_{\text{SSN2, Dname}} (\text{Dependents}))) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>DName</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>Tony</td>
<td>777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>

### Natural Join

- \( \text{R} = \text{S} = \text{R} \bowtie \text{S} \)

\[ \begin{array}{cccc}
A & B & C \\
X & Y & Z & U \\
Y & Z & \ \\
Z & \ & \ \\
\end{array} \quad \begin{array}{ccc}
B & C \\
Z & U \\
Z & V \\
\ \\
\end{array} \]

\[ \begin{array}{cc}
A & B & C \\
X & Z & U \\
Y & Z & U \\
Y & Z & V \\
Z & V & W \\
\end{array} \]

### Natural Join

- Given the schemas \( R(A, B, C, D), S(A, C, E) \), 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 \( R \bowtie S ? \)
Theta Join

- A join that involves a predicate
- Notation: $R_1 \bowtie_{\theta} R_2$ where $\theta$ is a condition
- Input schemas: $R_1(A_1, \ldots, A_n)$, $R_2(B_1, \ldots, B_m)$
- Output schema: $S(A_1, \ldots, A_n, B_1, \ldots, B_m)$
- It’s a derived operator:
  $R_1 \bowtie_{\theta} R_2 = \sigma_{\theta}(R_1 \times R_2)$

Equi-join

- Most frequently used in practice:
  $R_1 \bowtie_{A=B} R_2$
- Natural join is a particular case of equi-join
- A lot of research on how to do it efficiently

Semi-join

- $R \bowtie S = \Pi_{A_1, \ldots, A_n} (R \bowtie S)$
- Where the schemas are:
  - Input: $R(A_1, \ldots, A_n)$, $S(B_1, \ldots, B_m)$
  - Output: $T(A_1, \ldots, A_n)$

Semi-join

Applications in distributed databases:
- $Product(pid, cid, pname, \ldots)$ at site 1
- $Company(cid,ename,\ldots)$ at site 2
- Query: $\sigma_{price>1000}(T1) >_{\text{natural}} Company$
  - Compute as follows:
    $T1 = \sigma_{price>1000}(Product)$
    $T2 = P_{cid}(T1)$
    send $T2$ to site 2
    $T3 = T2 >_{\text{semi}} Company$
    send $T3$ to site 1
    Answer = $T3 >_{\text{semi}} T1$

Relational Algebra

- Five basic operators, many derived
- Combine operators in order to construct queries: relational algebra expressions, usually shown as trees

Complex Queries

Product (pid, name, price, category, maker-cid)
Purchase (buyer-ssn, seller-ssn, store, pid)
Company (cid, name, stock price, country)
Person (ssn, name, phone number, city)

Note:
* in Purchase: buyer-ssn, seller-ssn are foreign keys in Person, pid is foreign key in Product
* in Product maker-cid is a foreign key in Company

Find phone numbers of people who bought gizmos from Fred.
Find telephony products that somebody bought
Exercises

Product ( pid, name, price, category, maker-cid)
Purchase (buyer-ssn, seller-ssn, store, pid)
Company (cid, name, stock price, country)
Person(ssn, name, phone number, city)

Ex #1: Find people who bought telephony products.
Ex #2: Find names of people who bought American products
Ex #3: Find names of people who bought American products and did not buy French products
Ex #4: Find names of people who bought American products and they live in Seattle.
Ex #5: Find people who bought stuff from Joe or bought products from a company whose stock prices is more than $50.

Operations on Bags (and why we care)

- Union: \{a,b,c\} U \{a,b,b,h,e,f,f\} = \{a,a,b,b,h,b,c,e,f,f\}
  - add the number of occurrences
- Difference: \{a,b,b,c,e,c\} – \{b,c,c,d\} = \{a,b,b\}
  - subtract the number of occurrences
- Intersection: \{a,b,b,c,e,c\} \cap \{b,h,c,c,c,d\} = \{b,c,c\}
  - minimum of the two numbers of occurrences
- Selection: preserve the number of occurrences
- Projection: preserve the number of occurrences (no duplicate elimination)
- Cartesian product, join: no duplicate elimination

Reading assignment: 5.3

Summary of Relational Algebra

- Why bother? Can write any RA expression directly in C++/Java, seems easy.
- Two reasons:
  - Each operator admits sophisticated implementations (think of \(\sigma_{C}\))
  - Expressions in relational algebra can be rewritten: optimized

Glimpse Ahead: Efficient Implementations of Operators

- \(\sigma_{\text{age} \geq 30 \land \text{age} \leq 35} (\text{Employees})\)
  - Method 1: scan the file, test each employee
  - Method 2: use an index on \text{age}
  - Which one is better? Well, depends…
- \(\text{Employees} \bowtie \text{Relatives}\)
  - Iterate over Employees, then over Relatives
  - Iterate over Relatives, then over Employees
  - Sort Employees, Relatives, do “merge-join”
  - “hash-join”
  - etc

Glimpse Ahead: Optimizations

Product ( pid, name, price, category, maker-cid)
Purchase (buyer-ssn, seller-ssn, store, pid)
Person(ssn, name, phone number, city)

- Which is better:
  \(\sigma_{\text{price} > 100} (\text{Product}) \bowtie (\text{Purchase} \bowtie \sigma_{\text{city}=\text{sea}} (\text{Person}))\)
  \((\sigma_{\text{price} > 100} (\text{Product}) \bowtie \text{Purchase} \bowtie \sigma_{\text{city}=\text{sea}} (\text{Person}))\)
- Depends! This is the optimizer’s job…

Finally: RA has Limitations!

- Cannot compute “transitive closure”
  - Cannot express “transitive closure” in RA!!! Need to write C program
SQL Introduction

Basic form: (many many more bells and whistles in addition)

Select attributes
From relations (possibly multiple, joined)
Where conditions (selections)

Selections

Company(sticker, name, country, stockPrice)

Find all US companies whose stock is > 50:

```
SELECT *
FROM Company
WHERE country="USA" AND stockPrice > 50
```

Output schema: R(sticker, name, country, stockPrice)

Selections

What you can use in WHERE:
attribute names of the relation(s) used in the FROM
comparison operators: =, !=, <>, <, >, <=
apply arithmetic operations: stockPrice*2
operations on strings (e.g., "||" for concatenation).
Lexicographic order on strings.
Pattern matching: s LIKE p
Special stuff for comparing dates and times.

The LIKE operator

- s LIKE p: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - _ = any single character

Company(sticker, name, address, country, stockPrice)
Find all US companies whose address contains "Mountain":

```
SELECT *
FROM Company
WHERE country="USA" AND address LIKE "%Mountain%"
```
Projections
Select only a subset of the attributes

```sql
SELECT name, stockPrice
FROM Company
WHERE country='USA' AND stockPrice > 50
```

Input schema: Company(sticker, name, country, stockPrice)
Output schema: R(name, stock price)

Projections
Rename the attributes in the resulting table

```sql
SELECT name AS company, stockprice AS price
FROM Company
WHERE country='USA' AND stockPrice > 50
```

Input schema: Company(sticker, name, country, stockPrice)
Output schema: R(company, price)

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

```sql
SELECT name, stockPrice
FROM Company
WHERE country='USA' AND stockPrice > 50
ORDERBY country, name
```

Joins
Find names of people living in Seattle that bought gizmo products, and the names of the stores they bought from

```sql
SELECT pname, store
FROM Person, Purchase
WHERE pname=buyer AND city='Seattle'
AND product='gizmo'
```

Disambiguating Attributes
Find names of people buying telephony products:

```sql
SELECT Person.name
FROM Person, Purchase, Product
WHERE Person.name=Purchase.buyer
AND Product=Product.name
AND Product.category='telephony'
```

Tuple Variables
Find pairs of companies making products in the same category

```sql
SELECT product1.maker, product2.maker
FROM Product AS product1, Product AS product2
WHERE product1.category=product2.category
AND product1.maker <> product2.maker
```
Tuple Variables

Tuple variables introduced automatically by the system:

Product (name, price, category, maker)

Becomes:

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

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

Meaning (Semantics) of SQL Queries

```
SELECT a1, a2, ..., ak
FROM R1 AS x1, R2 AS x2, ..., Rn AS xn
WHERE Conditions
```

1. Nested loops:
```
Answer = {}
for x1 in R1 do
  for x2 in R2 do
    ....
    for xn in Rn do
      if Conditions then
        Answer = Answer U {(a1, ..., ak)}
  return Answer
```

2. Parallel assignment
```
Answer = {}
for all assignments x1 in R1, ..., xn in Rn do
  if Conditions then
    Answer = Answer U {(a1, ..., ak)}
return Answer
```

Doesn’t impose any order!
Like Datalog

First Unintuitive SQLism

```
SELECT a1, a2, ..., ak
FROM R1 AS x1, R2 AS x2, ..., Rn AS xn
WHERE Conditions
```

4. Translation to Relational algebra:
```
Π a1, ..., ak ( σ Conditions (R1 x R2 x ... x Rn))
```

Select-From-Where queries are precisely Select-Project-Join

Looking for $R \cap (S \cup T)$
But what happens if $T$ is empty?
Union, Intersection, Difference

```sql
(SELECT name
FROM Person
WHERE City="Seattle")
UNION
(SELECT name
FROM Person, Purchase
WHERE buyer=name AND store="The Bon")
```

Similarly, you can use INTERSECT and EXCEPT.
You must have the same attribute names (otherwise: rename).

Exercises

Product (pname, price, category, maker)
Purchase (buyer, seller, store, product)
Company (cname, stock price, country)
Person (pename, phone number, city)

- Ex #1: Find people who bought telephony products.
- Ex #2: Find names of people who bought American products
- Ex #3: Find names of people who bought American products and did not buy French products
- Ex #4: Find names of people who bought American products and they live in Seattle.
- Ex #5: Find people who bought stuff from Joe or bought products from a company whose stock prices is more than $50.

Subqueries

A subquery producing a single tuple:

```sql
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.

Can say the same thing without a subquery:

```sql
SELECT Purchase.product
FROM Purchase, Person
WHERE buyer = name AND ssn = "123456789"
```

Is this query equivalent to the previous one?

Subqueries Returning Relations

Find companies who manufacture products bought by Joe Blow.

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

Here the subquery returns a set of values

Subqueries Returning Relations

Equivalent to:

```sql
SELECT Company.name
FROM Company, Product, Purchase
WHERE Company.name=maker
AND Product.name = product
AND buyer = "Joe Blow"
```

Is this query equivalent to the previous one?
Subqueries Returning Relations

You can also use: 
- \( s > \text{ALL} \ R \)
- \( s > \text{ANY} \ R \)
- \( \exists x \ R \)

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

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")
```

**Question for Database Fans and their Friends**

- Can we express this query as a single **SELECT-FROM-WHERE** query, without subqueries?

  - Hint: show that all SFW queries are **monotone** (figure out what this means). A query with **ALL** is not monotone

Conditions on Tuples

**SELECT** Company.name
**FROM** Company, Product
**WHERE** Company.name=maker
**AND** (Product.name, price) IN (SELECT product, price
**FROM** Purchase
**WHERE** buyer = "Joe Blow");

Correlated Queries

**Movie** (\( \text{title, year, director, length} \))

Find movies whose title appears more than once.

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

Complex Correlated Query

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

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

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

Removing Duplicates

**SELECT** DISTINCT Company.name
**FROM** Company, Product
**WHERE** Company.name=maker
**AND** (Product.name, price) IN (SELECT product, price
**FROM** Purchase
**WHERE** buyer = "Joe Blow");

Powerful, but much harder to optimize!
Conserving Duplicates

The UNION, INTERSECTION and EXCEPT operators operate as sets, not bags.

(\textbf{SELECT} name
\textbf{FROM} Person
\textbf{WHERE} City="Seattle")
UNION \textbf{ALL}

(\textbf{SELECT} name
\textbf{FROM} Person, Purchase
\textbf{WHERE} buyer=name AND store="The Bon")