Database Systems

Session 5 – Main Theme

Relational Algebra, Relational Calculus, and SQL

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Agenda

1. Session Overview
2. Relational Algebra and Relational Calculus
3. Relational Algebra Using SQL Syntax
5. Summary and Conclusion

Session Agenda

- Session Overview
- Relational Algebra and Relational Calculus
- Relational Algebra Using SQL Syntax
- Summary & Conclusion

What is the class about?

- Course description and syllabus:
  - http://www.nyu.edu/classes/jcf/CSCI-GA.2433-001
  - http://cs.nyu.edu/courses/summer22/CSCI-GA.2433-001/
- Textbooks:
  - ***Fundamentals of Database Systems (7th Edition)***

Ramez Elmasri and Shamkant Navathe
Addition Wesley
ISBN-10: 0133970779, ISBN-13: 978-0133970777 7th Edition (06/18/15)

Icons / Metaphors

Information

Common Realization

Knowledge/Competency Pattern

Governance

Alignment

Solution Approach

Agenda

1. Session Overview
2. Relational Algebra and Relational Calculus
3. Relational Algebra Using SQL Syntax
5. Summary and Conclusion

Agenda

- Unary Relational Operations: SELECT and PROJECT
- Relational Algebra Operations from Set Theory
- Binary Relational Operations: JOIN and DIVISION
- Additional Relational Operations
- Examples of Queries in Relational Algebra
- The Tuple Relational Calculus
- The Domain Relational Calculus

The Relational Algebra and Relational Calculus

- **Relational algebra**
  - Basic set of operations for the relational model
- **Relational algebra expression**
  - Sequence of relational algebra operations
- **Relational calculus**
  - Higher-level declarative language for specifying relational queries

Unary Relational Operations: SELECT and PROJECT (1/3)

- The SELECT Operation
  - Subset of the tuples from a relation that satisfies a selection condition:

$$\sigma_{<\text{selection condition}>}(R)$$

    - Boolean expression contains clauses of the form <attribute name> <comparison op> <constant value>
    - *or*
    - <attribute name> <comparison op> <attribute name>

Unary Relational Operations: SELECT and PROJECT (2/3)

- Example:

$$\sigma_{(\text{Dno}=4 \text{ AND } \text{Salary}>25000) \text{ OR } (\text{Dno}=5 \text{ AND } \text{Salary}>30000)}(\text{EMPLOYEE})$$

- <selection condition> applied independently to each individual tuple *t* in *R*
  - If condition evaluates to TRUE, tuple selected
- Boolean conditions **AND**, **OR**, and **NOT**
- **Unary**
  - Applied to a single relation

Unary Relational Operations: SELECT and PROJECT (3/3)

- **Selectivity**
  - Fraction of tuples selected by a selection condition
- SELECT operation commutative
- **Cascade** SELECT operations into a single operation with **AND** condition

The PROJECT Operation

- Selects columns from table and discards the other columns:

$$\pi_{<\text{attribute list}>}(R)$$

- **Degree**
  - Number of attributes in <attribute list>
- **Duplicate elimination**
  - Result of PROJECT operation is a set of distinct tuples

Sequences of Operations and the RENAME Operation

- **In-line** expression:

$$\pi_{\text{Fname, Lname, Salary}}(\sigma_{\text{Dno}=5}(\text{EMPLOYEE}))$$

- Sequence of operations:

$$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{DEP5\_EMPS})$$

- **Rename** attributes in intermediate results
  - RENAME operation

$$\rho_{S(B1, B2, \ldots, Bn)}(R) \quad \text{or} \quad \rho_{S}(R) \quad \text{or} \quad \rho_{(B1, B2, \ldots, Bn)}(R)$$

Relational Algebra Operations from Set Theory (1/2)

- **UNION**, **INTERSECTION**, and **MINUS**
  - Merge the elements of two sets in various ways
  - Binary operations
  - Relations must have the same type of tuples
- UNION
  - *R* U *S*
  - Includes all tuples that are either in *R* or in *S* or in both *R* and *S*
  - Duplicate tuples eliminated

Relational Algebra Operations from Set Theory (2/2)

- INTERSECTION
  - $R \cap S$
  - Includes all tuples that are in both *R* and *S*
- SET DIFFERENCE (or MINUS)
  - $R - S$
  - Includes all tuples that are in *R* but not in *S*

The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

- **CARTESIAN PRODUCT**
  - **CROSS PRODUCT** or **CROSS JOIN**
  - Denoted by ×
  - Binary set operation
  - Relations do not have to be union compatible
  - Useful when followed by a selection that matches values of attributes

Binary Relational Operations: JOIN and DIVISION (1/2)

- The **JOIN** Operation
  - Denoted by ⋈
  - Combine related tuples from two relations into single "longer" tuples
  - General join condition of the form <condition> **AND** <condition> **AND**...**AND** <condition>
  - Example:

$$\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE}$$
$$\text{RESULT} \leftarrow \pi_{\text{Dname, Lname, Fname}}(\text{DEPT\_MGR})$$

Binary Relational Operations: JOIN and DIVISION (2/2)

- **THETA** JOIN
  - Each <condition> of the form A_i θ B_j
  - A_i is an attribute of R
  - B_j is an attribute of S
  - A_i and B_j have the same domain
  - θ (theta) is one of the comparison operators:
    - {=, <, ≤, >, ≥, ≠}

Variations of JOIN: The EQUIJOIN and NATURAL JOIN (1/2)

- **EQUIJOIN**
  - Only = comparison operator used
  - Always have one or more pairs of attributes that have identical values in every tuple
- **NATURAL JOIN**
  - Denoted by *
  - Removes second (superfluous) attribute in an EQUIJOIN condition

Variations of JOIN: The EQUIJOIN and NATURAL JOIN (2/2)

- **Join selectivity**
  - Expected size of join result divided by the maximum size $n_R * n_S$
- **Inner joins**
  - Type of match and combine operation
  - Defined formally as a combination of CARTESIAN PRODUCT and SELECTION

A Complete Set of Relational Algebra Operations

- Set of relational algebra operations {σ, π, U, ρ, –, ×} is a **complete set**
  - Any relational algebra operation can be expressed as a sequence of operations from this set

The DIVISION Operation

- Denoted by ÷
- Example: retrieve the names of employees who work on all the projects that 'John Smith' works on
- Apply to relations $R(Z) \div S(X)$
  - Attributes of R are a subset of the attributes of S

Operations of Relational Algebra (1/2)

Table 6.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{\text{<selection condition>}}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{\text{<attribute list>}}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{\text{<join condition>}} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{\text{<join condition>}} R_2$, OR $R_1 \bowtie_{(\text{<join attributes 1>}), (\text{<join attributes 2>})} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 *_{\text{<join condition>}} R_2$, OR $R_1 *_{(\text{<join attributes 1>}), (\text{<join attributes 2>})} R_2$ OR $R_1 * R_2$

Operations of Relational Algebra (2/2)

Table 6.1 Operations of Relational Algebra

UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

Notation for Query Trees

- **Query tree**
  - Represents the input relations of query as leaf nodes of the tree
  - Represents the relational algebra operations as internal nodes

Sample Query Tree for Relational Algebra Expression

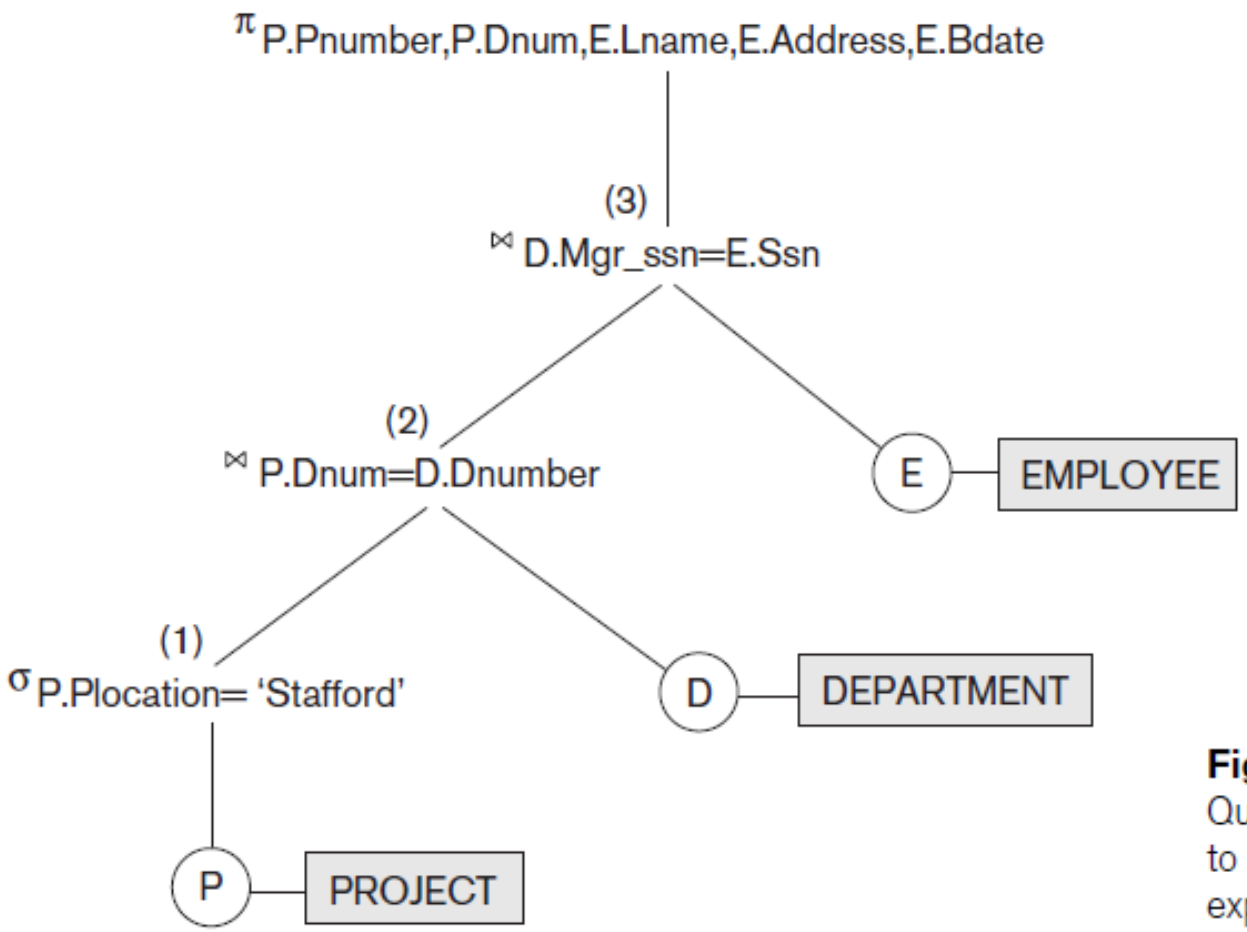


Figure 6.9
Query tree corresponding to the relational algebra expression for Q2.

Additional Relational Operations (1/2)

- **Generalized projection**
  - Allows functions of attributes to be included in the projection list

$$\pi_{F1, F2, ..., Fn}(R)$$

- **Aggregate functions** and **grouping**
  - Common functions applied to collections of numeric values
  - Include SUM, AVERAGE, MAXIMUM, and MINIMUM

Additional Relational Operations (2/2)

- Group tuples by the value of some of their attributes
  - Apply aggregate function independently to each group

$$<\text{grouping attributes}>\ \Im\ <\text{function list}>\ (R)$$

Sample Aggregate Function Operation

Figure 6.10
The aggregate function operation.

a. $\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}(\text{Dno } \mathfrak{I} \text{ COUNT Ssn, AVERAGE Salary}(\text{EMPLOYEE}))$.

b. $\text{Dno } \mathfrak{I} \text{ COUNT Ssn, AVERAGE Salary}(\text{EMPLOYEE})$.

c. $\mathfrak{I} \text{ COUNT Ssn, AVERAGE Salary}(\text{EMPLOYEE})$.

R

(a)

Dno	No_of_employees	Average_sal
5	4	33250
4	3	31000
1	1	55000

(b)

Dno	Count_ssn	Average_salary
5	4	33250
4	3	31000
1	1	55000

(c)

Count_ssn	Average_salary
8	35125

[8]Note that this is an arbitrary notation we are suggesting. There is no standard notation.

Recursive Closure Operations

- Operation applied to a **recursive relationship** between tuples of same type

$$\text{BORG\_SSN} \leftarrow \pi_{\text{Ssn}}(\sigma_{\text{Fname='James' AND Lname='Borg'}}(\text{EMPLOYEE}))$$
$$\text{SUPERVISION}(\text{Ssn1}, \text{Ssn2}) \leftarrow \pi_{\text{Ssn,Super\_ssn}}(\text{EMPLOYEE})$$
$$\text{RESULT1}(\text{Ssn}) \leftarrow \pi_{\text{Ssn1}}(\text{SUPERVISION} \bowtie_{\text{Ssn2=Ssn}} \text{BORG\_SSN})$$

OUTER JOIN Operations

- **Outer joins** ⟕
  - Keep all tuples in *R*, or all those in *S*, or all those in both relations regardless of whether or not they have matching tuples in the other relation
  - **Types**
    - **LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN**
  - Example:

$$\text{TEMP} \leftarrow (\text{EMPLOYEE} \ ⟕_{\text{Ssn=Mgr\_ssn}} \text{DEPARTMENT})$$

$$\text{RESULT} \leftarrow \pi_{\text{Fname, Minit, Lname, Dname}}(\text{TEMP})$$

The OUTER UNION Operation

- Take union of tuples from two relations that have some common attributes
  - Not union (type) compatible
- **Partially compatible**
  - All tuples from both relations included in the result
  - Tuples with the same value combination will appear only once

Examples of Queries in Relational Algebra (1/3)

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

$$\text{RESEARCH\_DEPT} \leftarrow \sigma_{\text{Dname='Research'}}(\text{DEPARTMENT})$$
$$\text{RESEARCH\_EMPS} \leftarrow (\text{RESEARCH\_DEPT} \bowtie_{\text{Dnumber=Dno}} \text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Address}}(\text{RESEARCH\_EMPS})$$

As a single in-line expression, this query becomes:

$$\pi_{\text{Fname, Lname, Address}} (\sigma_{\text{Dname='Research'}}(\text{DEPARTMENT} \bowtie_{\text{Dnumber=Dno}} (\text{EMPLOYEE}))$$

Examples of Queries in Relational Algebra (2/3)

Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

$$\text{STAFFORD\_PROJS} \leftarrow \sigma_{\text{Plocation}=\text{'Stafford'}}(\text{PROJECT})$$
$$\text{CONTR\_DEPTS} \leftarrow (\text{STAFFORD\_PROJS} \bowtie_{\text{Dnum}=\text{Dnumber}} \text{DEPARTMENT})$$
$$\text{PROJ\_DEPT\_MGRS} \leftarrow (\text{CONTR\_DEPTS} \bowtie_{\text{Mgr\_ssn}=\text{Ssn}} \text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(\text{PROJ\_DEPT\_MGRS})$$

Query 3. Find the names of employees who work on *all* the projects controlled by department number 5.

$$\text{DEPT5\_PROJS} \leftarrow \rho_{(\text{Pno})}(\pi_{\text{Pnumber}}(\sigma_{\text{Dnum}=5}(\text{PROJECT})))$$
$$\text{EMP\_PROJ} \leftarrow \rho_{(\text{Ssn, Pno})}(\pi_{\text{Essn, Pno}}(\text{WORKS\_ON}))$$
$$\text{RESULT\_EMP\_SSNS} \leftarrow \text{EMP\_PROJ} \div \text{DEPT5\_PROJS}$$
$$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(\text{RESULT\_EMP\_SSNS} * \text{EMPLOYEE})$$

Examples of Queries in Relational Algebra (3/3)

Query 6. Retrieve the names of employees who have no dependents.

This is an example of the type of query that uses the MINUS (SET DIFFERENCE) operation.

$$\text{ALL\_EMPS} \leftarrow \pi_{\text{Ssn}}(\text{EMPLOYEE})$$
$$\text{EMPS\_WITH\_DEPS(Ssn)} \leftarrow \pi_{\text{Essn}}(\text{DEPENDENT})$$
$$\text{EMPS\_WITHOUT\_DEPS} \leftarrow (\text{ALL\_EMPS} - \text{EMPS\_WITH\_DEPS})$$
$$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(\text{EMPS\_WITHOUT\_DEPS} * \text{EMPLOYEE})$$

Query 7. List the names of managers who have at least one dependent.

$$\text{MGRS(Ssn)} \leftarrow \pi_{\text{Mgr\_ssn}}(\text{DEPARTMENT})$$
$$\text{EMPS\_WITH\_DEPS(Ssn)} \leftarrow \pi_{\text{Essn}}(\text{DEPENDENT})$$
$$\text{MGRS\_WITH\_DEPS} \leftarrow (\text{MGRS} \cap \text{EMPS\_WITH\_DEPS})$$
$$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(\text{MGRS\_WITH\_DEPS} * \text{EMPLOYEE})$$

The Tuple Relational Calculus

- Declarative expression
  - Specify a retrieval request
  - Non-procedural language
- Any retrieval that can be specified in basic relational algebra
  - Can also be specified in relational calculus

Tuple Variables and Range Relations

- **Tuple variables**
  - Ranges over a particular database relation
- **Satisfy** COND(*t*):
- Specify: $\{t \mid \text{COND}(t)\}$
  - **Range relation** *R* of *t*
  - Select particular combinations of tuples
  - Set of attributes to be retrieved (**requested attributes**)

Expressions and Formulas in Tuple Relational Calculus

- General expression of tuple relational calculus is of the form:

$$\{t_1.A_j, t_2.A_k, \ldots, t_n.A_m \mid \text{COND}(t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m})\}$$

- **Truth value** of an atom
  - Evaluates to either TRUE or FALSE for a specific combination of tuples
- **Formula** (Boolean condition)
  - Made up of one or more atoms connected via logical operators **AND**, **OR**, and **NOT**

Existential and Universal Quantifiers

- **Universal quantifier** (inverted “A”)
- **Existential quantifier** (mirrored “E”)
- Define a tuple variable in a formula as **free** or **bound**

Sample Queries in Tuple Relational Calculus

Query 1. List the name and address of all employees who work for the 'Research' department.

Q1: {*t*.Fname, *t*.Lname, *t*.Address | EMPLOYEE(*t*) **AND** (∃*d*)(DEPARTMENT(*d*)
AND *d*.Dname='Research' **AND** *d*.Dnumber=*t*.Dno)}

Query 4. Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as manager of the controlling department for the project.

Q4: { *p*.Pnumber | PROJECT(*p*) **AND** (((∃*e*)(∃*w*)(EMPLOYEE(*e*)
AND WORKS_ON(*w*) **AND** *w*.Pno=*p*.Pnumber
AND e.Lname='Smith' **AND** e.Ssn=*w*.Essn))
OR
((∃*m*)(∃*d*)(EMPLOYEE(*m*) **AND** DEPARTMENT(*d*)
AND *p*.Dnum=*d*.Dnumber **AND** *d*.Mgr_ssn=*m*.Ssn
AND *m*.Lname='Smith'))))}

Notation for Query Graphs

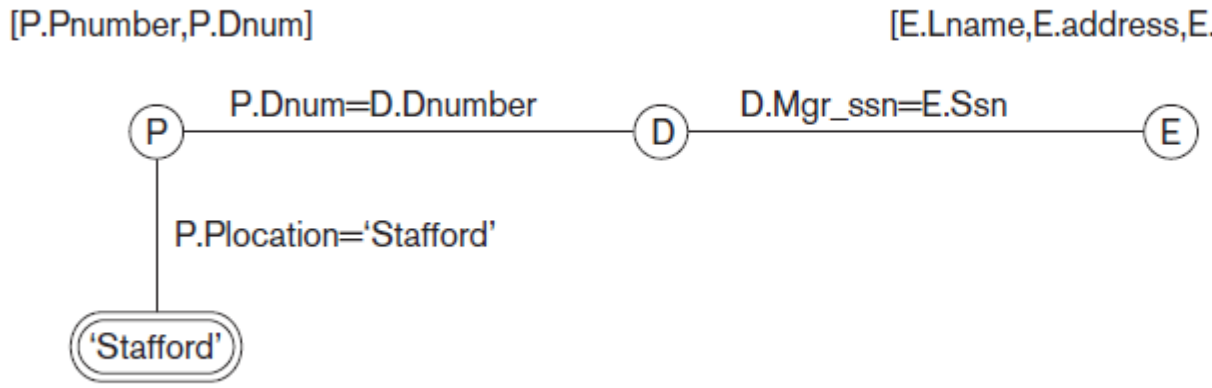


Figure 6.13
Query graph for Q2.

Transforming the Universal and Existential Quantifiers

- Transform one type of quantifier into other with negation (preceded by **NOT**)
  - **AND** and **OR** replace one another
  - Negated formula becomes un-negated
  - Un-negated formula becomes negated

Using the Universal Quantifier in Queries

Query 3. List the names of employees who work on *all* the projects controlled by department number 5. One way to specify this query is to use the universal quantifier as shown:

Q3: $\{e.\text{Lname}, e.\text{Fname} \mid \text{EMPLOYEE}(e)$ **AND** $((\forall x)(\textbf{NOT}(\text{PROJECT}(x))$ **OR NOT** $(x.\text{Dnum}=5)$ **OR** $((\exists w)(\text{WORKS\_ON}(w)$ **AND** $w.\text{Essn}=e.\text{Ssn}$ **AND** $x.\text{Pnumber}=w.\text{Pno}))))\}$

Q3A: $\{e.\text{Lname}, e.\text{Fname} \mid \text{EMPLOYEE}(e)$ **AND** $(\textbf{NOT}\ (\exists x)\ (\text{PROJECT}(x)$ **AND** $(x.\text{Dnum}=5)$ AND $(\textbf{NOT}\ (\exists w)(\text{WORKS\_ON}(w)$ **AND** $w.\text{Essn}=e.\text{Ssn}$ **AND** $x.\text{Pnumber}=w.\text{Pno}))))\}$

Safe Expressions

- Guaranteed to yield a finite number of tuples as its result
  - Otherwise expression is called **unsafe**
- Expression is **safe**
  - If all values in its result are from the domain of the expression

The Domain Relational Calculus (1/2)

- Differs from tuple calculus in type of variables used in formulas
  - Variables range over single values from domains of attributes
- Formula is made up of **atoms**
  - Evaluate to either TRUE or FALSE for a specific set of values
    - Called the **truth values** of the atoms

The Domain Relational Calculus (2/2)

- QBE language
  - Based on domain relational calculus

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

Q1: $\{q, s, v \mid (\exists z)\ (\exists l)\ (\exists m)\ (\text{EMPLOYEE}(qrstuvwxyz)\ \textbf{AND}\ \text{DEPARTMENT}(lmno)\ \textbf{AND}\ l='\text{Research}'\ \textbf{AND}\ m=z)\}$

Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, birth date, and address.

Q2: $\{i, k, s, u, v \mid (\exists j)(\exists m)(\exists n)(\exists t)(\text{PROJECT}(hijk)\ \textbf{AND}\ \text{EMPLOYEE}(qrstuvwxyz)\ \textbf{AND}\ \text{DEPARTMENT}(lmno)\ \textbf{AND}\ k=m\ \textbf{AND}\ n=t\ \textbf{AND}\ j='\text{Stafford}')\}$

Summary

- Formal languages for relational model of data:
  - Relational algebra: operations, unary and binary operators
  - Some queries cannot be stated with basic relational algebra operations
    - But are important for practical use
- Relational calculus
  - Based predicate calculus

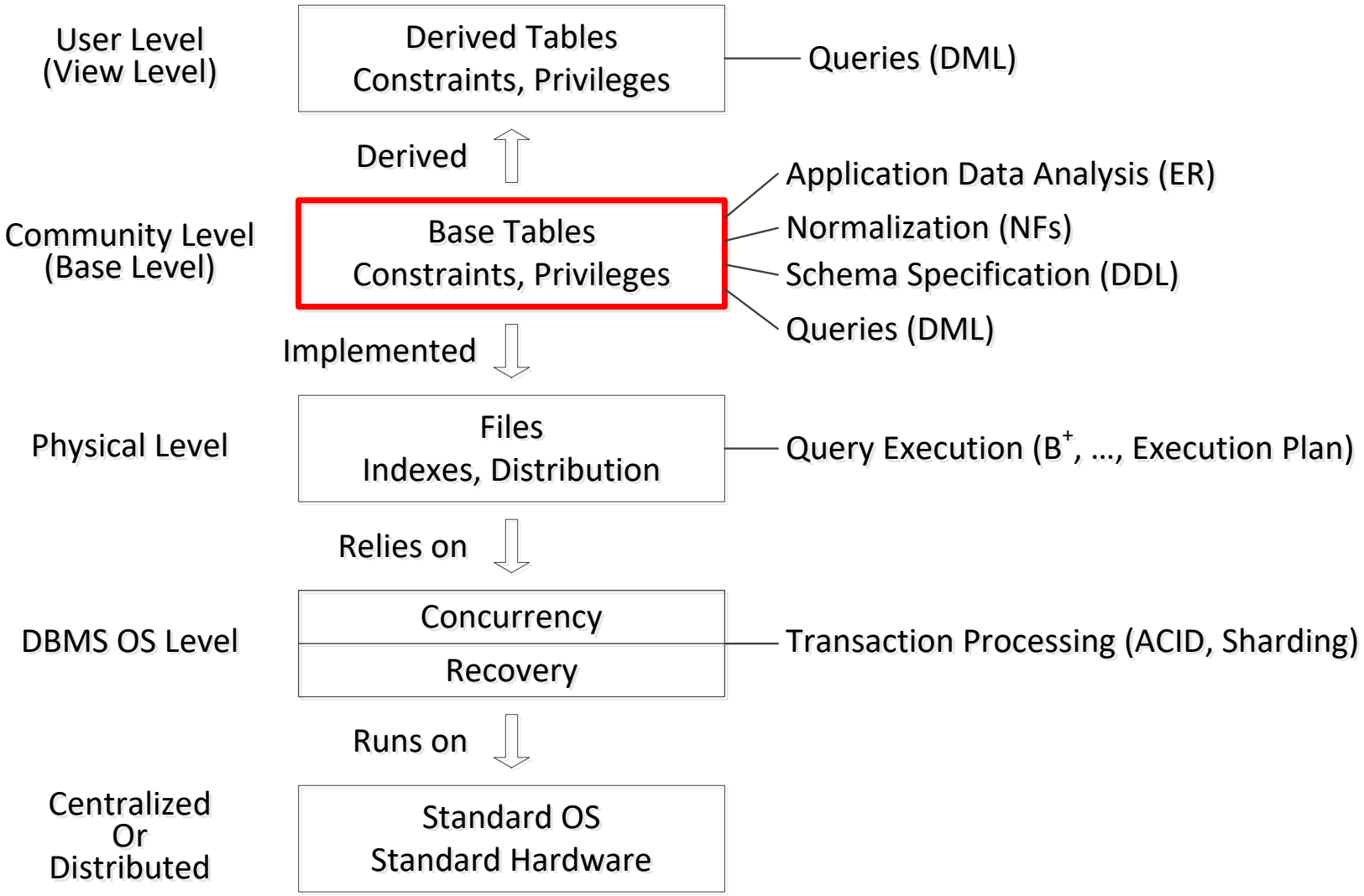
Agenda

1. Session Overview
2. Relational Algebra and Relational Calculus
3. Relational Algebra Using SQL Syntax
4. Summary and Conclusion

Agenda

- Relational Algebra and SQL
- Basic Syntax Comparison
- Sets and Operations on Relations
- Relations in Relational Algebra
- Empty Relations
- Relational Algebra vs. Full SQL
- Operations on Relations
  - » Projection
  - » Selection
  - » Cartesian Product
  - » Union
  - » Difference
  - » Intersection
- From Relational Algebra to Queries (with Examples)
- Microsoft Access Case Study
- Pure Relational Algebra

Relational Algebra in Context



Querying Relational Databases (DQL: Data Query Language)

- Three approaches for DQL in a decreasing order of desirability and increasing order of ease of learning
  1. ***Symbolic logic***: predicate logic

     Here are the conditions that the answer must satisfy: DBMS figures out how to produce an answer satisfying the conditions

  2. ***Relational algebra***: well-defined expressions using fundamental relational, algebraic operations

     Here is the expression to produce the answer; DBMS evaluates the expression

  3. ***Procedural language***: a program to be executed

     Here is the program: DBMS runs the program

- ***SQL is (2) with frequently sub-optimal syntax, and where useful it adds (3) and additional interfaces to standard programming languages***

SQL Fundamental Operations - Relational Algebra with Sweetened Syntax

- ***SQL DQL is based on relational algebra with many extensions***
  - Some necessary
  - Some unnecessary
- "Pure" relational algebra uses mathematical notation with Greek letters
- It is covered here using SQL syntax; that is this unit covers relational algebra, but it looks like SQL
  - And will be valid SQL
- ***All the "fundamental" operations of SQL are in this unit***
  - Of course, this is a slight exaggeration, but not far from the truth
- To emphasize the core of SQL, the term "relational algebra" is frequently used for the subset of SQL discussed in this unit
- In the next unit: other important SQL operations

Key Differences Between SQL And "Pure" Relational Algebra

- SQL data model is a ***multiset*** not a set; still rows in tables (we sometimes continue calling relations)
  - » Still no order among rows: no such thing as 1st row
  - » We can (if we want to) count how many times a particular row appears in the table
  - » We can remove/not remove duplicates as we specify (most of the time)
  - » There are some operators that specifically pay attention to duplicates
  - » We ***must*** know whether duplicates are removed (and how) for each SQL operation; luckily, easy
- Many redundant operators (relational algebra had only one: intersection)
- SQL provides statistical operators, such as AVG (average)
  - » Can be performed on subsets of rows; e.g. average salary per company branch

Key Differences Between Relational Algebra And SQL

- Every domain is “enhanced” with a special element: NULL
  - Very strange semantics for handling these elements
- “Pretty printing” of output: sorting, and similar
- Operations for
  - Inserting
  - Deleting
  - Changing/updating (sometimes not easily reducible to deleting and inserting)

Basic Syntax Comparison (1/2)

Relational Algebra	SQL
$\pi_{a, b}$	SELECT a, b
$\sigma_{(d > e) \wedge (f = g)}$	WHERE d > e AND f = g
$p \times q$	FROM p, q
$\pi_{a, b}\, \sigma_{(d > e) \wedge (f = g)} (p \times q)$	SELECT a, b FROM p, q WHERE d > e AND f = g; {must always have SELECT even if all attributes are kept, can be written as: SELECT *}
renaming	AS {or blank space}
p := result	INSERT INTO p result {assuming p was empty}
$\pi_{a, b}$ (p) (assume a, b are the only attributes)	SELECT * FROM p;

Basic Syntax Comparison (2/2)

Relational Algebra	SQL
$p \cup q$	SELECT * FROM p UNION SELECT * FROM q
$p - q$	SELECT * FROM p EXCEPT SELECT * FROM q Sometimes, instead, we have DELETE FROM
$p \cap q$	SELECT * FROM p INTERSECT SELECT * FROM q

Sets And Operations On Them

- If A, B, and C are sets, then we have the operations
- $\cup$ Union, $A \cup B = \{ x \mid x \in A \ \vee \ x \in B \}$
- $\cap$ Intersection, $A \cap B = \{ x \mid x \in A \ \wedge \ x \in B \}$
- – Difference, $A - B = \{ x \mid x \in A \ \wedge \ x \notin B \}$
- In mathematics, set difference is frequently denoted by a symbol similar to a backslash: $A \setminus B = A - B$
- $\times$ Cartesian product, $A \times B = \{ (x,y) \mid x \in A \ \wedge \ y \in B \}$, $A \times B \times C = \{ (x,y,z) \mid x \in A \ \wedge \ y \in B \ \wedge \ z \in B \}$, etc.
- The above operations form an ***algebra***, that is you can perform operations on results of operations, such as $(A \cap B) \times (C \times A)$ and such operations always produce se
- So you can write expressions and not just programs!

Relations in Relational Algebra

- Relations are sets of **tuples**, which we will also call ***rows***, drawn from some domains
- These domains do not include NULLs
- Relational algebra deals with relations (which look like tables with fixed number of columns and varying number of rows)
- We assume that each domain is linearly ordered, so for each *x* and *y* from the domain, one of the following holds
  - » $x < y$
  - » $x = y$
  - » $x < y$
- Frequently, such comparisons will be meaningful even if *x* and *y* are drawn from different columns
  - » For example, one column deals with income and another with expenditure: we may want to compare them

Reminder: Relations in Relational Algebra

- The order of rows and whether a row appears once or many times does not matter
- The order of columns matters, but as our columns will always be labeled, we will be able to reconstruct the order even if the columns are permuted.
- The following two relations are equal:

R	A	B
	1	10
	2	20

R	B	A
	20	2
	10	1
	20	2
	20	2

Many Empty Relations

- In set theory, there is only one empty set
- For us, it is more convenient to think that for each relation schema, that for specific choice of column names and domains, there is a different empty relation
- And of, course, two empty relations with different number of columns must be different
- So for instance the two relations below are different

- The above needs to be stated more precisely to be “completely correct,” but as this will be intuitively clear, we do not need to worry about this too much

Relational Algebra Versus Full SQL

- Relational algebra is restricted to querying the database
- Does not have support for
  - » Primary keys
  - » Foreign keys
  - » Inserting data
  - » Deleting data
  - » Updating data
  - » Indexing
  - » Recovery
  - » Concurrency
  - » Security
  - » …
- Does not care about efficiency, only about specifications of what is needed

Operations on relations

- There are several fundamental operations on relations
- We will describe them in turn:
  - » Projection
  - » Selection
  - » Cartesian product
  - » Union
  - » Difference
  - » Intersection (technically not fundamental)
- The very important property: ***Any operation on relations produces a relation***
- This is why we call this an ***algebra***

Projection: Choice Of Columns

R	A	B	C	D
	1	10	100	1000
	1	20	100	1000
	1	20	200	1000

- SQL statement

```
SELECT B, A, D
FROM R
```

	B	A	D
	10	1	1000
	20	1	1000
	20	1	1000

- We could have removed the duplicate row, but did not have to

Intuitive Explanation (Formally not Meaningful but Very Useful)

- R is a file of records
- Each record is tuple
- Each record consists of fields (values of attributes)

- Execute the following “program”

1. Create a new empty file
2. Loop on the records on file R
3. Keep only some specific fields of each record and append this “modified” record to the new file

Selection: Choice Of Rows

R	A	B	C	D
	5	5	7	4
	5	6	5	7
	4	5	4	4
	5	5	5	5
	4	6	5	3
	4	4	3	4
	4	4	4	5
	4	6	4	6

- SQL statement:

  SELECT * (this means all columns)

  FROM R

  WHERE A <= C AND D = 4; (this is a predicate, i.e., condition)

	A	B	C	D
	5	5	7	4
	4	5	4	4

Selection: Choice of Rows

- SQL statement:

```
SELECT *
FROM R
WHERE A <= C AND D = 4;
```

- We can consider

  WHERE condition;

  as telling us that ***we should not accept a tuple whose values violate condition***

- Therefore, ***if there is no***

  WHERE condition;

  no condition can be violated, and then ***all the tuples are "good"*** and passed to the SELECT

Some Interesting Cases

- What should be the result of

```
SELECT *
FROM R
WHERE 1 = 2;
```

- What should be the result of

```
SELECT *
FROM R
WHERE 1 = 1;
```

- What should be the result of

```
SELECT *
FROM R
WHERE ;
```

Intuitive Explanation (Formally not Meaningful but Very Useful)

- R is a file of records
- Each record is tuple
- Each record consists of fields (values of attributes)

- Execute the following “program”

1. Create a new empty file
2. Loop on the records of file R
3. Check if a record satisfies some conditions on the values of the field; if there is no WHERE condition, every tuple satisfies that condition
4. If the conditions are satisfied append the record to the new file, otherwise discard it

Selection

- In general, the condition (predicate) can be specified by a Boolean formula with

  NOT, AND, OR on atomic conditions, where a condition is:

  - a comparison between two column names,
  - a comparison between a column name and a constant
  - Technically, a constant should be put in quotes
  - Even a number, such as 4, perhaps should be put in quotes, as '4', so that it is distinguished from a column name, but as we will *never* use numbers for column names, this not necessary
- Note that "not equal" is written generally as one (or both) of the two
  - <>
  - !=

Cartesian Product

R	A	B
	1	10
	2	10
	2	20

S	C	B	D
	40	10	10
	50	20	10

- SQL statement

  SELECT A, R.B, C, S.B, D

  FROM R, S; (comma stands for Cartesian product)

	A	R.B	C	S.B	D
	1	10	40	10	10
	1	10	50	20	10
	2	10	40	10	10
	2	10	50	20	10
	2	20	40	10	10
	2	20	50	20	10

Intuitive Explanation (Formally Not Meaningful But Very Useful)

- R and S are files of records
- Each record is tuple
- Each record consists of fields (values of attributes)

- Execute the following “program”

1. Create a new empty file
2. Outer loop: Read one-by-one the records of file R
3. Inner loop: Read one-by-one the records of file S
4. Combine the record from R with the record from S
5. Append to the new file the new “combined” record

A Typical Use Of Cartesian Product

R	Size	Room#
	140	1010
	150	1020
	140	1030

S	ID#	Room#	YOB
	40	1010	1982
	50	1020	1985

- SQL statement:

```
SELECT ID#, R.Room#, Size
FROM R, S
WHERE R.Room# = S.Room#;
```

	ID#	R.Room#	Size
	40	1010	140
	50	1020	150

A Typical Use Of Cartesian Product

- After the Cartesian product, we got

	Size	R.Room#	ID#	S.Room #	YOB
	140	1010	40	1010	1982
	140	1010	50	1020	1985
	150	1020	40	1010	1982
	150	1020	50	1020	1985
	140	1030	40	1010	1982
	140	1030	50	1020	1985

- This allowed us to correlate the information from the two original tables by examining each tuple in turn

A Typical Use Of Cartesian Product

- This example showed how to correlate information from two tables
  - » The first table had information about rooms and their sizes
  - » The second table had information about employees including the rooms they sit in
  - » The resulting table allows us to find out what are the sizes of the rooms the employees sit in
- We had to specify R.Room# or S.Room# in SELECT, even though they happen to be equal because we need to specify from which relation a specific column in the output is drawn
- We could, as we will see later, rename a column, to get Room#

	ID#	Room#	Size
	40	1010	140
	50	1020	150

WHERE Can Formally Examine Only One Tuple at a Time

- WHERE examines only one tuple at a time
- Therefore if we want to correlate information from more than one relation by looking at several tuples from several relations we have to “prepare” something that is a single tuple
  - » Or even if we want to look at two (or more) tuples from the same relation
- We take the cartesian product of the relevant relations
  - » Or even the cartesian product of two (or more) copies of the same relation
- A single tuple in the resulting relation allows us to correlate several tuples from the original relations

Union

R	A	B
	1	10
	2	20

S	A	B
	1	10
	3	20

- SQL statement

```
(SELECT *
FROM R)
UNION
(SELECT *
FROM S);
```

	A	B
	1	10
	2	20
	3	20

- Note: We happened to choose to remove duplicate rows
- Note: we ***could not*** just write R UNION S (syntax quirk)

Union Compatibility

- We require same -arity (number of columns), otherwise the result is not a relation
- Also, the operation “probably” should make sense, that is the values in corresponding columns should be drawn from the same domains
- We refer to these as ***union compatibility*** of relations
- Sometimes, just the term ***compatibility*** is used
- The names of the columns in the result are taken from the first relation
- Actually, in this unit for clarity, it is best to assume that the column names are the same and that is what we will do from now on

Difference

R	A	B
	1	10
	2	20

S	A	B
	1	10
	3	20

- SQL statement

```
(SELECT *
FROM R)
MINUS
(SELECT *
FROM S);
```

	A	B
	2	20

- Union compatibility required
- EXCEPT is a synonym for MINUS

Intersection

R	A	B
	1	10
	2	20

S	A	B
	1	10
	3	20

- SQL statement

```
(SELECT *
FROM R)
INTERSECT
(SELECT *
FROM S);
```

	A	B
	1	10

- Union compatibility required
- Can be computed using differences only: R – (R – S)

From Relational Algebra to Queries

- These operations allow us to define a large number of interesting queries for relational databases.
- In order to be able to formulate our examples, we will assume standard “bookkeeping” type of operations:
  - » Assignment of an expression to a new variable;

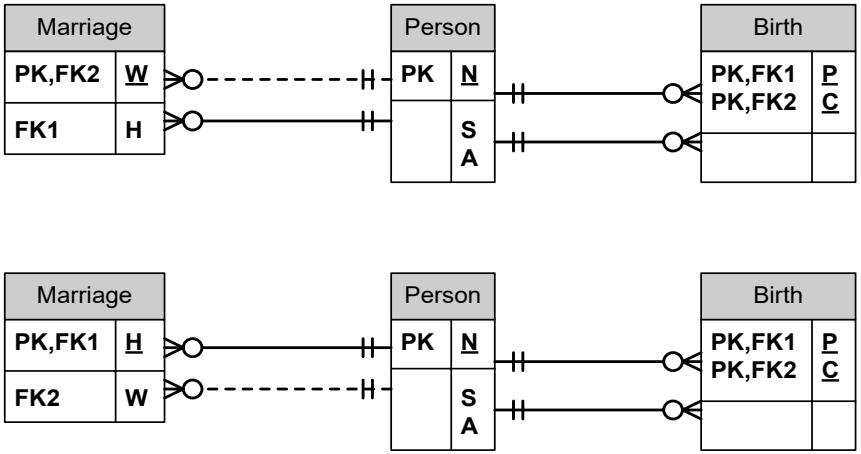
    In our case assignment of a relational expression to a relational variable.
  - » Renaming of a relations, to use another name to denote it
  - » Renaming of a column, to use another name to denote it

A Small Example

- The example consists of 3 relations:
  - Person(Name,Sex,Age)
    - This relation, whose primary key is Name, gives information about the human's sex and age
  - Birth(Parent,Child)
    - This relation, whose primary key is the pair Parent,Child, with both being foreign keys referring to Person gives information about who is a parent of whom. (Both mother and father would be generally listed)
  - Marriage(Husband,Wife, Age) **or**
  - Marriage(Husband,Wife, Age)
    - This relation listing current marriages only, requires choosing which spouse will serve as primary key. For our exercise, it does not matter what the choice is. Both Husband and Wife are foreign keys referring to Person. Age specifies how long the marriage has lasted.
  - For each attribute above, we will frequently use its first letter to refer to it, to save space in the slides, unless it creates an ambiguity
  - Some ages do not make sense, but this is fine for our example

Relational Implementation (1/2)

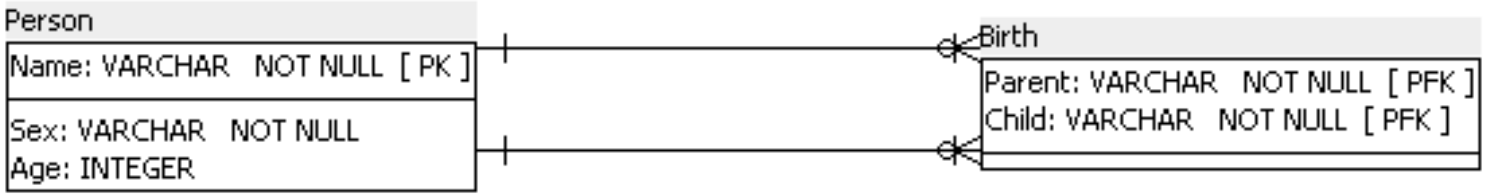
- Two options for selecting the primary key of Marriage
- The design is not necessarily good, but nice and simple for learning relational algebra



- Because we want to focus on relational algebra, which does not understand keys, we will not specify keys in this unit

Relational Implementation (2/2)

- The design is not necessarily good in practice as many people may have the same name, but nice and simple for learning relational algebra



- Because we want to focus on relational algebra, which does not understand keys, we will not specify keys in this unit when we implement the database, though we do “understand” that Name identifies a person

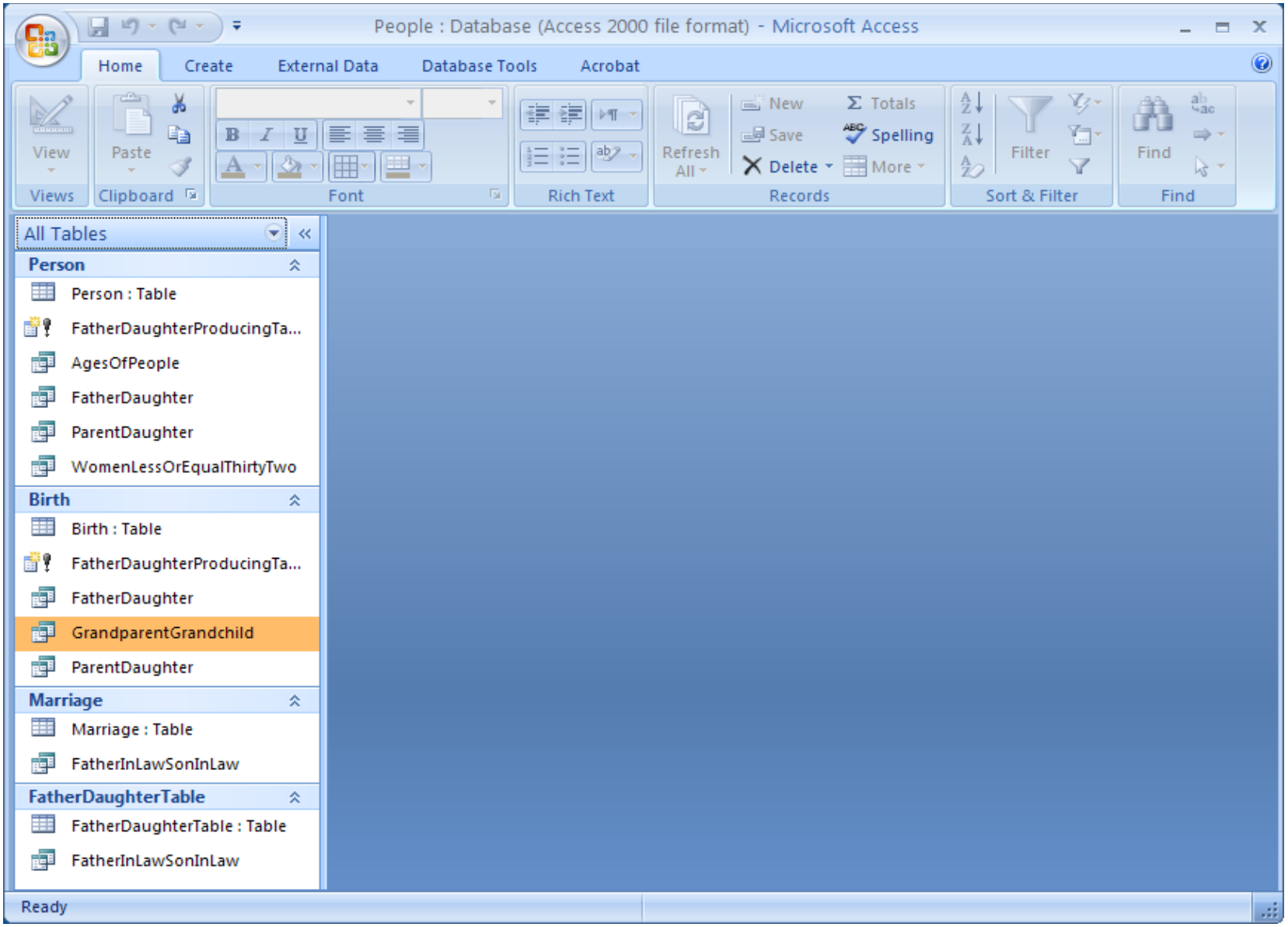
Microsoft Access Database

- Microsoft Access is a very good tool for quickly demonstrating basic constructs of SQL DQL, although it is not suitable for anything other than personal databases and even there is suffers from various defects
- Note
  - MINUS is sometimes specified in commercial databases in a roundabout way, for example in MySQL
  - The next unit will get into how it is done
- The queries were ran in Microsoft Access and copied and pasted them in these notes, after reformatting them
  - It is more suitable for producing good visualisations
- As useful, the results of the queries were copied and pasted as screen shots

Microsoft Access Database

- The database and our queries (other than the one with MINUS at the end) are the appropriate “extras” directory on the class web in “slides”
  - MINUS is frequently specified in commercial databases in a roundabout way
  - We will cover how it is done when we discuss commercial databases
- Our sample Access database: People.mdb
- The queries in Microsoft Access are copied and pasted in these notes, after reformatting them
- Included copied and pasted screen shots of the results of the queries so that you can correlate the queries with the names of the resulting tables

Our Database With Sample Queries - Open In Microsoft Access



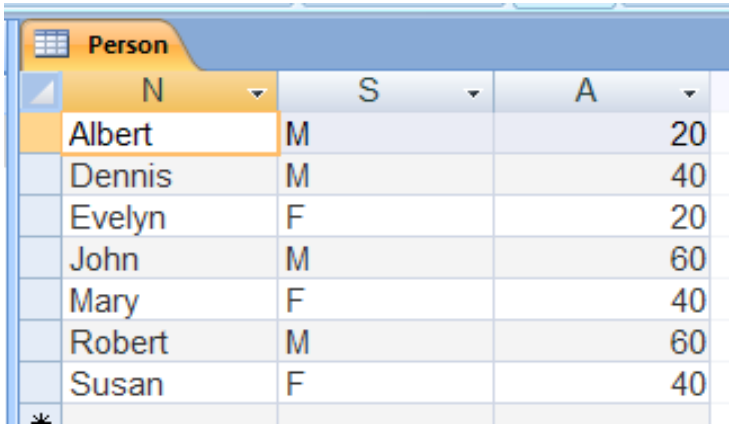
Our Database

Person	N	S	A
	Albert	M	20
	Dennis	M	40
	Evelyn	F	20
	John	M	60
	Mary	F	40
	Robert	M	60
	Susan	F	40

Birth	P	C
	Dennis	Albert
	John	Mary
	Mary	Albert
	Robert	Evelyn
	Susan	Evelyn
	Susan	Richard

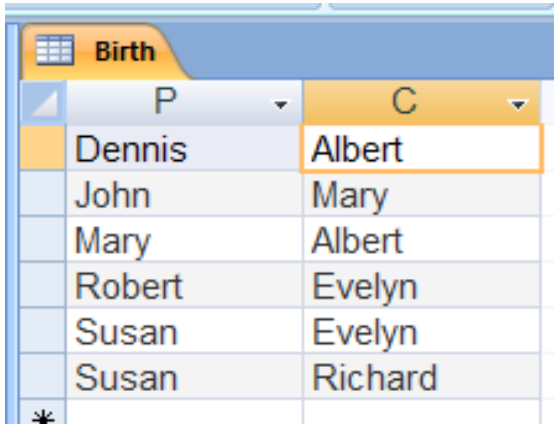
Marriage	H	W	A
	Dennis	Mary	20
	Robert	Susan	30

Our Instance In Microsoft Access



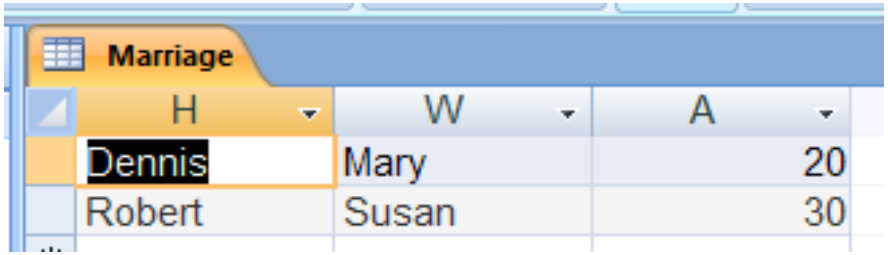
Person

N	S	A
Albert	M	20
Dennis	M	40
Evelyn	F	20
John	M	60
Mary	F	40
Robert	M	60
Susan	F	40



Birth

P	C
Dennis	Albert
John	Mary
Mary	Albert
Robert	Evelyn
Susan	Evelyn
Susan	Richard



Marriage

H	W	A
Dennis	Mary	20
Robert	Susan	30

A Query

- Produce the relation Answer(A) consisting of all ages of people
- Note that all the information required can be obtained from looking at a single relation, Person
- Answer:=

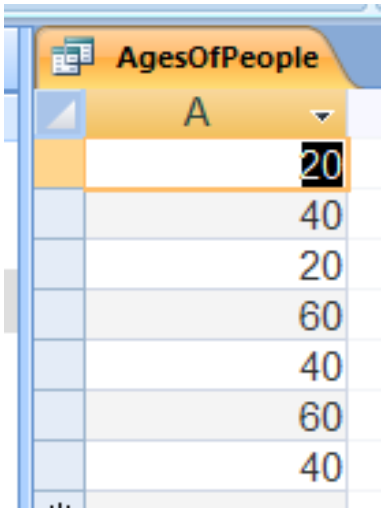
  SELECT A

  FROM Person;
- Recall that whether duplicates are removed is not important (at least for the time being in our course)

	A
	20
	40
	20
	60
	40
	60
	40

The Query In Microsoft Access

- The actual query was copied and pasted from Microsoft Access and reformatted for readability
- The result is below



Our Database

Person	N	S	A
	Albert	M	20
	Dennis	M	40
	Evelyn	F	20
	John	M	60
	Mary	F	40
	Robert	M	60
	Susan	F	40

Birth	P	C
	Dennis	Albert
	John	Mary
	Mary	Albert
	Robert	Evelyn
	Susan	Evelyn
	Susan	Richard

Marriage	H	W	A
	Dennis	Mary	20
	Robert	Susan	30

A Query

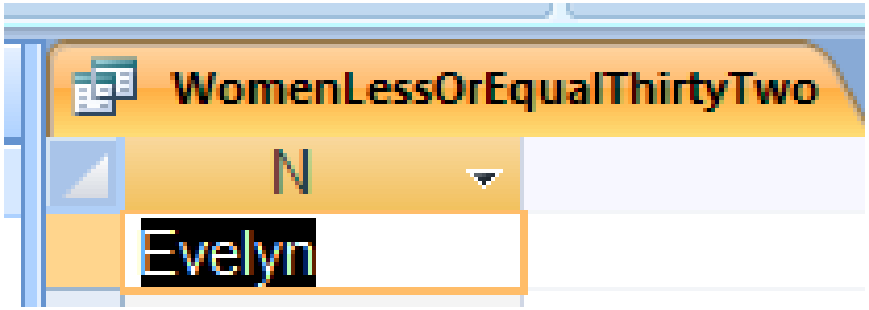
- Produce the relation Answer(N) consisting of all women who are less or equal than 32 years old.
- Note that all the information required can be obtained from looking at a single relation, Person
- Answer:=

```
SELECT N
FROM Person
WHERE A <= 32 AND S ='F';
```

	N
	Evelyn

The Query In Microsoft Access

- The actual query was copied and pasted from Microsoft Access and reformatted for readability
- The result is below



A Query

- Produce a relation Answer(P, Daughter) with the obvious meaning
- Here, even though the answer comes only from the single relation Birth, we still have to check in the relation Person what the S of the C is
- To do that, we create the Cartesian product of the two relations: Person and Birth. This gives us “long tuples,” consisting of a tuple in Person and a tuple in Birth
- For our purpose, the two tuples matched if N in Person is C in Birth and the S of the N is F

Our Database

Person	N	S	A
	Albert	M	20
	Dennis	M	40
	Evelyn	F	20
	John	M	60
	Mary	F	40
	Robert	M	60
	Susan	F	40

Birth	P	C
	Dennis	Albert
	John	Mary
	Mary	Albert
	Robert	Evelyn
	Susan	Evelyn
	Susan	Richard

Marriage	H	W	A
	Dennis	Mary	20
	Robert	Susan	30

A Query

Answer:=

```
SELECT P, C AS Daughter
FROM Person, Birth
WHERE C = N AND S = 'F';
```

	P	Daughter
	John	Mary
	Robert	Evelyn
	Susan	Evelyn

- Note: AS is the attribute renaming operator

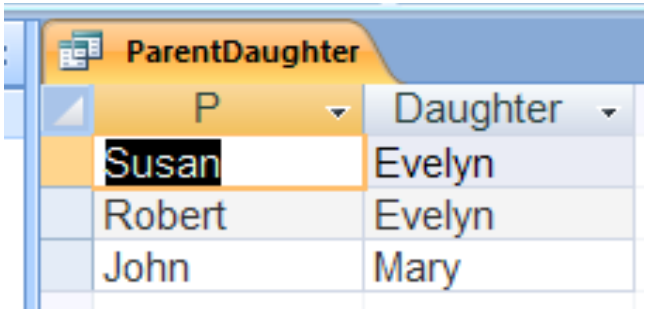
Cartesian Product With Condition: Matching Tuples Indicated

Person	N	S	A
	Albert	M	20
	Dennis	M	40
	Evelyn	F	20
	John	M	60
	Mary	F	40
	Robert	M	60
	Susan	F	40

Birth	P	C
	Dennis	Albert
	John	Mary
	Mary	Albert
	Robert	Evelyn
	Susan	Evelyn
	Susan	Richard

The Query In Microsoft Access

- The actual query was copied and pasted from Microsoft Access and reformatted for readability
- The result is below



Our Database

Person	N	S	A
	Albert	M	20
	Dennis	M	40
	Evelyn	F	20
	John	M	60
	Mary	F	40
	Robert	M	60
	Susan	F	40

Birth	P	C
	Dennis	Albert
	John	Mary
	Mary	Albert
	Robert	Evelyn
	Susan	Evelyn
	Susan	Richard

Marriage	H	W	A
	Dennis	Mary	20
	Robert	Susan	30

A Query

- Produce a relation Answer(Father, Daughter) with the obvious meaning.
- Here we have to simultaneously look at two copies of the relation Person, as we have to determine both the S of the Parent and the S of the C
- We need to have ***two distinct copies*** of Person in our SQL query
- But, they have to have different names so we can specify to which we refer
- Again, we use AS as a renaming operator, these time for relations
- Note: We could have used what we have already computed: Answer(Parent,Daughter) and only to "contrain" the S(ex) of the Parent

A Query

- Answer :=

  SELECT P AS Father, C AS Daughter

  FROM Person, Birth, Person AS Person1

  WHERE P = Person.N AND C = Person1.N

  AND Person.S = ‘M’ AND Person1.S = ‘F’;

	Father	Daughter
	John	Mary
	Robert	Evelyn

Cartesian Product With Condition: Matching Tuples Indicated

Person	N	S	A
	Albert	M	20
	Dennis	M	40
	Evelyn	F	20
	John	M	60
	Mary	F	40
	Robert	M	60
	Susan	F	40

Person	N	S	A
	Albert	M	20
	Dennis	M	40
	Evelyn	F	20
	John	M	60
	Mary	F	40
	Robert	M	60
	Susan	F	40

Birth	P	C
	Dennis	Albert
	John	Mary
	Mary	Albert
	Robert	Evelyn
	Susan	Evelyn
	Susan	Richard

Thinking Like a Programmer May Help in Writing Relational Algebra Queries

- We can solve the problem working with files Person and Birth

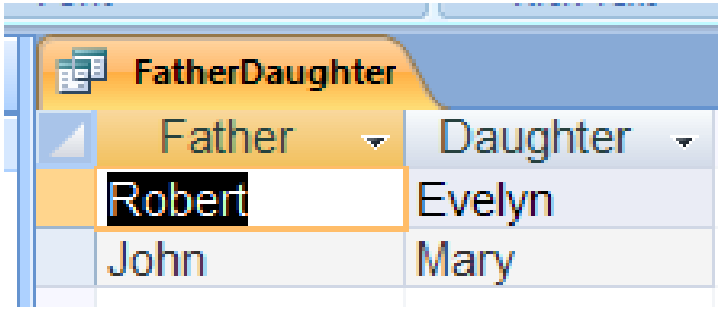
- for i = first to last record in Person
  for j = first to last record in Birth
  for k = first to last record in Person
  examine/combine the records i, j, and k

- We had two variables looping over Person and one variable looping over Birth

- That's why, loosely speaking, in relational algebra we needed two copies of Person and one copy of Birth
  - » We do not have "looping variables" such as i, j, and k.
  - » We need to create the set of the "combinations" and need different names for the two copies of Person

The Query In Microsoft Access

- The actual query was copied and pasted from Microsoft Access and reformatted for readability
- The result is below



Our Database

Person	N	S	A
	Albert	M	20
	Dennis	M	40
	Evelyn	F	20
	John	M	60
	Mary	F	40
	Robert	M	60
	Susan	F	40

Birth	P	C
	Dennis	Albert
	John	Mary
	Mary	Albert
	Robert	Evelyn
	Susan	Evelyn
	Susan	Richard

Marriage	H	W	A
	Dennis	Mary	20
	Robert	Susan	30

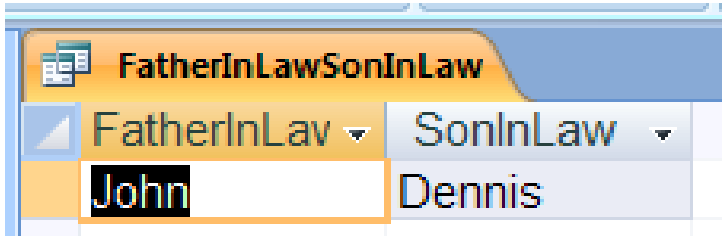
A Query

- Produce a relation: Answer(Father_in_law, Son_in_law).
- A classroom exercise, but you can see the solution in the posted database.
- Hint: you need to compute the Cartesian product of several relations if you start from scratch, or of two relations if you use the previously computed (Father, Daughter) relation

	F_I_L	S_I_L
	John	Dennis

The Query In Microsoft Access

- The actual query was copied and pasted from Microsoft Access and reformatted for readability
- The result is below



Our Database

Person	N	S	A
	Albert	M	20
	Dennis	M	40
	Evelyn	F	20
	John	M	60
	Mary	F	40
	Robert	M	60
	Susan	F	40

Birth	P	C
	Dennis	Albert
	John	Mary
	Mary	Albert
	Robert	Evelyn
	Susan	Evelyn
	Susan	Richard

Marriage	H	W	A
	Dennis	Mary	20
	Robert	Susan	30

A Query

- Produce a relation: Answer(Grandparent,Grandchild)
- Answer :=

```
SELECT Birth.P AS G_P, Birth1.C AS G_C
FROM Birth, Birth AS Birth1
WHERE Birth.C = Birth1.P;
```

	G_P	G_C
	John	Albert

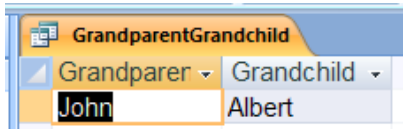
Cartesian Product With Condition: Matching Tuples Indicated

Birth	P	C
	Dennis	Albert
	John	Mary
	Mary	Albert
	Robert	Evelyn
	Susan	Evelyn
	Susan	Richard

Birth	P	C
	Dennis	Albert
	John	Mary
	Mary	Albert
	Robert	Evelyn
	Susan	Evelyn
	Susan	Richard

The Query in Microsoft Access

- The actual query was copied and pasted from Microsoft Access and reformatted for readability
- The result is below



Further Distance

- How to compute (Great-grandparent,Great-grandchild)?
- Easy, just take the Cartesian product of the (Grandparent, Grandchild) table with (Parent,Child) table and specify equality on the “intermediate” person
- How to compute (Great-great-grandparent,Great-great-grandchild)?
- Easy, just take the Cartesian product of the (Grandparent, Grandchild) table with itself and specify equality on the “intermediate” person
- Similarly, can compute (Great^x-grandparent,Great^x-grandchild), for any *x*
- Ultimately, may want (Ancestor,Descendant)

Relational Algebra Is Not Universal:
Cannot Compute (Ancestor,Descendant)

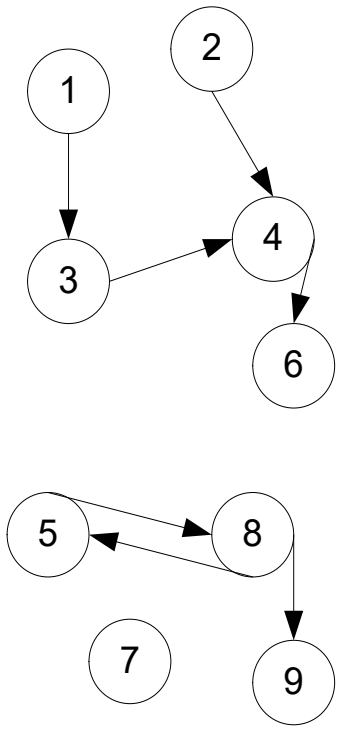
- Standard programming languages are ***universal***
- This roughly means that they are as powerful as Turing machines, if unbounded amount of storage is permitted (you will never run out of memory)
- This roughly means that they can compute anything that can be computed by any computational machine we can (at least currently) imagine
- Relational algebra is weaker than a standard programming language
- It is impossible in relational algebra (or standard SQL) to compute the relation Answer(Ancestor, Descendant)

Relational Algebra Is Not Universal: Cannot Compute (Ancestor,Descendant)

- It is impossible in relational algebra (and early versions of SQL), but we will see how to do it in Oracle later in the course) to compute the relation Answer(Ancestor, Descendant)
- Why can't we do it using the operations we have so far?
- The proof is a reasonably simple, but uses cumbersome induction.
- The general idea is:
  - » Any relational algebra query is limited in how many relations or copies of relations it can refer to
  - » Computing arbitrary (ancestor, descendant) pairs cannot be done, if the query is limited in advance as to the number of relations and copies of relations (including intermediate results) it can specify
  - » From an intuitive standpoint consider: given a fixed-length arithmetic formula with integers, can you compute arbitrarily large integers?
- This is not a contrived example because it shows that we cannot compute the transitive closure of a directed graph: the set of all paths in the graph

Relational Algebra Is Not Universal: Cannot Compute Transitive Closures

- Given **Arc** we would like to compute **Path (transitive closure)** but cannot do it for arbitrary-size graphs using early SQL but can do it starting with the 1999 SQL standard and in Oracle for DAGs only



Arc	From	To
	1	3
	2	4
	3	4
	4	6
	5	8
	8	5
	8	9

Path	From	To
	1	3
	1	4
	1	6
	2	4
	2	6
	3	4
	3	6
	4	6
	5	8
	5	9
	8	5
	8	9

Our Database

Person	N	S	A
	Albert	M	20
	Dennis	M	40
	Evelyn	F	20
	John	M	60
	Mary	F	40
	Robert	M	60
	Susan	F	40

Birth	P	C
	Dennis	Albert
	John	Mary
	Mary	Albert
	Robert	Evelyn
	Susan	Evelyn
	Susan	Richard

Marriage	H	W	A
	Dennis	Mary	20
	Robert	Susan	30

A Sample Query

- Produce a relation Answer(A) consisting of all ages of males that are not (also) ages of females

```
SELECT A
FROM Person
WHERE S = 'M'
MINUS
SELECT A
FROM Person
WHERE S = 'F';
```

- Note that the following is not correct

```
SELECT A
FROM Person
WHERE S != 'F';
```

The Query in Microsoft Access

- We do not show this here, as it is done in a roundabout way, and we will do it later
- Same problem with MySQL

It Does not Matter If We Remove Duplicates

- Removing duplicates

	A
	20
	40
	60

–

	A
	20
	40

=

	A
	60

- Not removing duplicates

	A
	20
	40
	60
	60

–

	A
	20
	40
	40

=

	A
	60
	60

It Does not Matter If We Remove Duplicates

- The resulting set contains precisely ages: 60
- So, we do not have to be concerned with whether the implementation removes duplicates from the result or not
- In both cases we can answer correctly
  - Is 50 a number that is an age of a marriage but not of a person?
  - Is 40 a number that is an age of a marriage but not of a person?

- Just like we do not have to be concerned with whether it sorts (orders) the result

- This is the consequence of us ***not*** insisting that an element in a set appears only once, as we discussed earlier
- ***Note, if we had said that an element in a set appears once, we would have had to spend effort removing duplicates!***

More About Minus

- Additional exercise: find all parents who do not have daughters

Our Database

Person	N	S	A
	Albert	M	20
	Dennis	M	40
	Evelyn	F	20
	John	M	60
	Mary	F	40
	Robert	M	60
	Susan	F	40

Birth	P	C
	Dennis	Albert
	John	Mary
	Mary	Albert
	Robert	Evelyn
	Susan	Evelyn
	Susan	Richard

Marriage	H	W	A
	Dennis	Mary	20
	Robert	Susan	30

A Sample Query

- Produce a relation Answer(A) consisting of all ages of visitors that are not ages of marriages

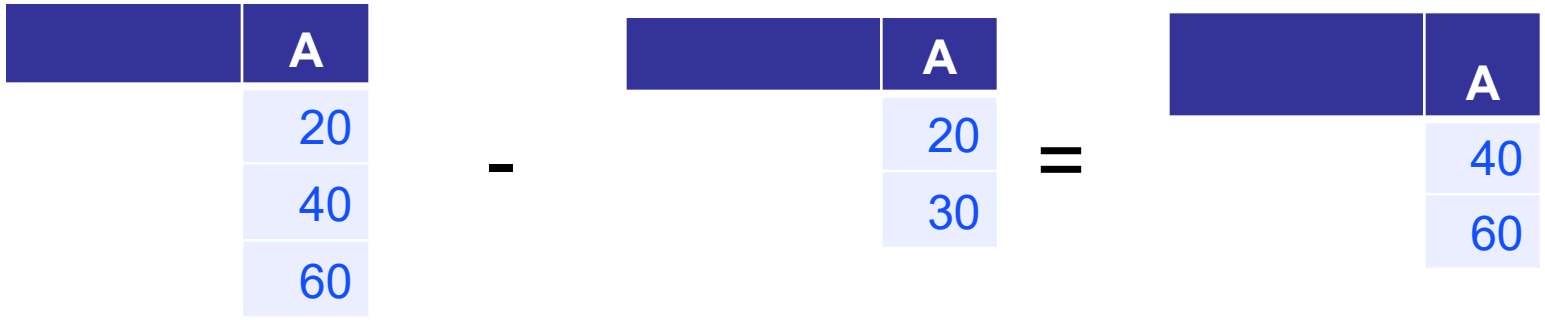
```
SELECT
A FROM Person
MINUS
SELECT
A FROM MARRIAGE;
```

The Query In Microsoft Access

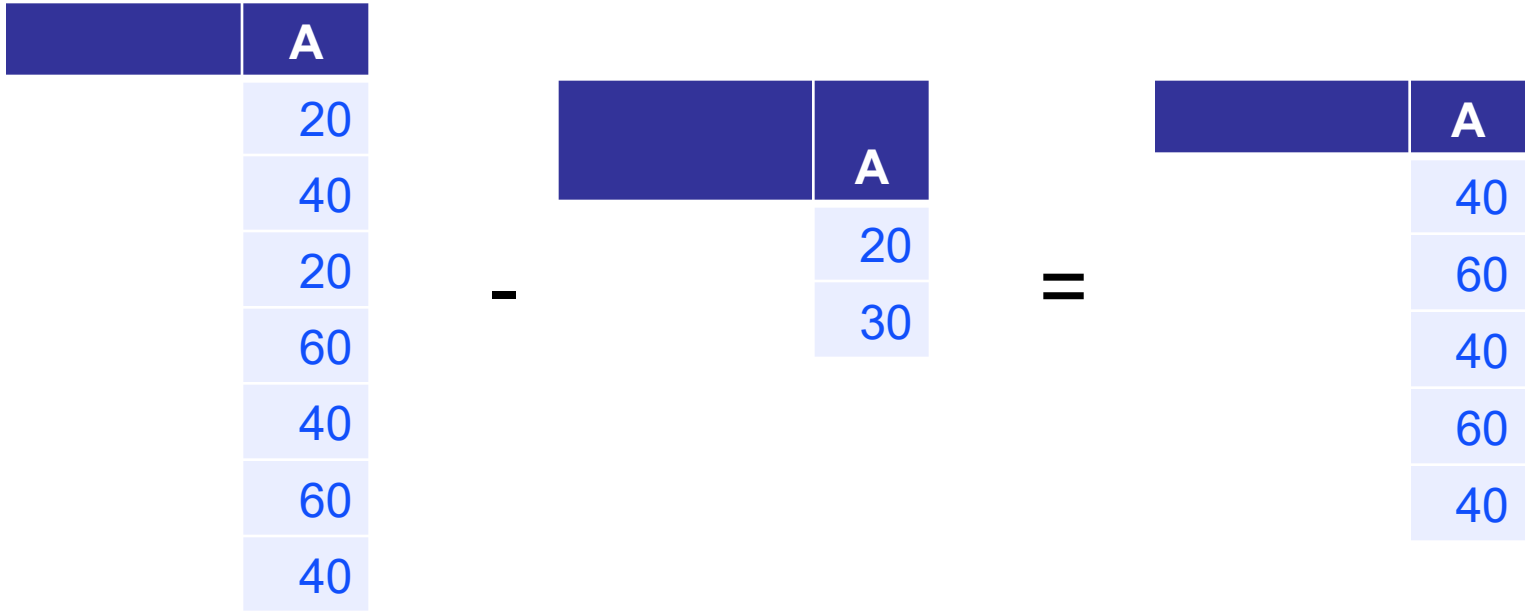
- We do not show this here, as it is done in a roundabout way and we will do it later

It Does Not Matter If We Remove Duplicates

- Removing duplicates



- Not removing duplicates



It Does Not Matter If We Remove Duplicates

- The resulting set contains precisely ages: 40, 60
- So we do not have to be concerned with whether the implementation removes duplicates from the result or not
- In both cases we can answer correctly
  - Is 50 a number that is an age of a marriage but not of a person
  - Is 40 a number that is an age of a marriage but not of a person
- Just like we do not have to be concerned with whether it sorts (orders) the result
- This is the consequence of us not insisting that an element in a set appears only once, as we discussed earlier
- ***Note, if we had said that an element in a set appears once, we would have to spend effort removing duplicates!***

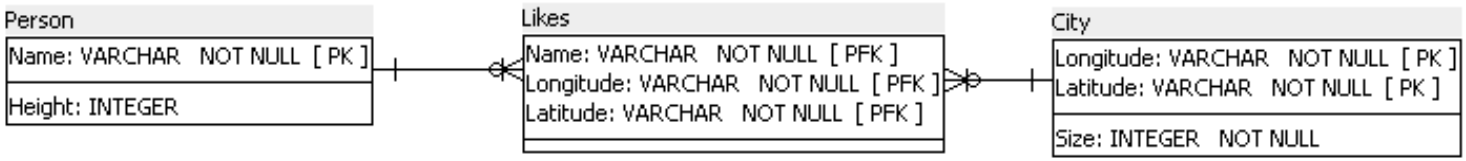
Note On Some Common Operations That We Do Not Cover In This Unit

- Informally, what we have done could be called “joins”
- You may have heard about operations such as
  - Join On
  - Inner Join
  - EquiJoin
- They are all easily implementable using our “template” of

  SELECT …
  FROM …
  WHERE …

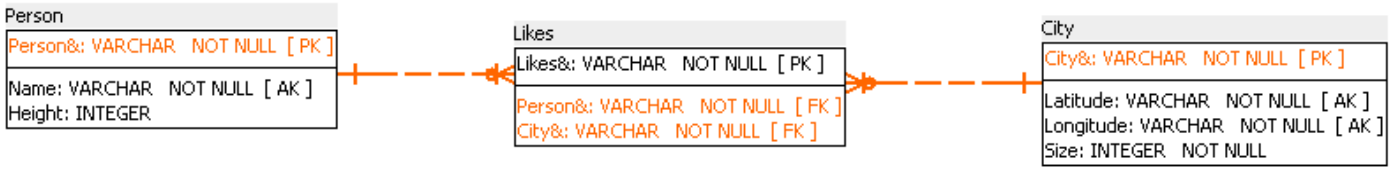
  So, no need to use them, and do not use such syntax in your assignments in our class unless instructed otherwise
- Left, Right, Outer joins are implementable using our operations after we learn more about NULLs
  - We will learn about them later

Querying With Natural Primary Keys



- Find the sizes of the cities that persons of height 6 like
- SELECT Size
  FROM Person, Likes, City
  WHERE Person.Name = Likes.Name AND
  Likes.Longitude = City.Longitude AND Likes.Latitude =
  City.Latitude AND Height = 6;

Querying With Surrogate Primary Keys



- Name is UNIQUE in Person
- (Longitude, Latitude) is UNIQUE in City
- We have not looked at AK before; it stands for Alternate Key and that's what SQL Power Architect calls UNIQUE

- Find the sizes of the cities that persons of height 6 like
- SELECT Size
  FROM Person, Likes, City
  WHERE Person.Person& = Likes.Person& AND Likes.City& = City.City& AND Height = 6;

Now To "Pure" Relational Algebra

- This was described in several slides
  - Just the basic operations, as before
  - Other operations can be derived from the basic ones
- But it is really the same as before, just the notation is more mathematical
- Looks like mathematical expressions, not snippets of programs
- It is useful to know this because many articles use this instead of SQL
- This notation came first, before SQL was invented, and when relational databases were just a theoretical construct

π: Projection: Choice Of Columns

R	A	B	C	D
	1	10	100	1000
	1	20	100	1000
	1	20	200	1000

- SQL statement Relational Algebra

```
SELECT B, A, D
FROM R
```

	B	A	D
	10	1	1000
	20	1	1000
	20	1	1000

$\pi_{B,A,D}(R)$

- We could have removed the duplicate row, but did not have to

σ: Selection: Choice Of Rows

R	A	B	C	D
	5	5	7	4
	5	6	5	7
	4	5	4	4
	5	5	5	5
	4	6	5	3
	4	4	3	4
	4	4	4	5
	4	6	4	6

- SQL statement:

```
SELECT *
FROM R
WHERE A <= C AND D = 4;
```

Relational Algebra

$\sigma_{A \leq C \wedge D=4}(R)$ Note: no need for π

	A	B	C	D
	5	5	7	4
	4	5	4	4

Selection

- In general, the condition (predicate) can be specified by a Boolean formula with

  $\neg$, $\wedge$, and $\vee$ on atomic conditions, where a condition is:

  - » a comparison between two column names,
  - » a comparison between a column name and a constant
  - » Technically, a constant should be put in quotes
  - » Even a number, such as 4, perhaps should be put in quotes, as ‘4’ so that it is distinguished from a column name, but as we will *never* use numbers for column names, this not necessary

×: Cartesian Product

R	A	B
	1	10
	2	10
	2	20

S	C	B	D
	40	10	10
	50	20	10

- SQL statement

```
SELECT A, R.B, C, S.B, D
FROM R, S
```

Relational Algebra

$R \times S$

	A	R.B	C	S.B	D
	1	10	40	10	10
	1	10	50	20	10
	2	10	40	10	10
	2	10	50	20	10
	2	20	40	10	10
	2	20	50	20	10

A Typical Use Of Cartesian Product

R	Size	Room#
	140	1010
	150	1020
	140	1030

S	ID#	Room#	YOB
	40	1010	1982
	50	1020	1985

- SQL statement:

```
SELECT ID#, R.Room#, Size
FROM R, S
WHERE R.Room# = S.Room#
```

Relational Algebra

$$\pi_{\text{ID\#, R.Room\#, Size}}(\sigma_{\text{R.B = S.B}}(R \times S))$$

	ID#	R.Room#	Size
	40	1010	140
	50	1020	150

∪: Union

R	A	B
	1	10
	2	20

S	A	B
	1	10
	3	20

- SQL statement

```
(SELECT *
FROM R)
UNION
(SELECT *
FROM S)
```

	A	B
	1	10
	2	20
	3	20

Relational Algebra

$R \cup S$

- Note: We happened to choose to remove duplicate rows
- Union compatibility required

–: Difference

R	A	B
	1	10
	2	20

S	A	B
	1	10
	3	20

- SQL statement

```
(SELECT *
FROM R)
MINUS
(SELECT *
FROM S)
```

	A	B
	2	20

Relational Algebra

R – S

- Union compatibility required

∩: Intersection

R	A	B
	1	10
	2	20

S	A	B
	1	10
	3	20

- SQL statement

```
(SELECT *
FROM R)
INTERSECT
(SELECT *
FROM S)
```

	A	B
	1	10

Relational Algebra

$R \cap S$

- Union compatibility required

Addendum 1

- We want to express a very natural condition

  IF antecedent THEN consequent

- Example:

  IF the person works THEN the person has an ssn
  - ssn stands for Social Security Number

- But we only have NOT, OR, AND

- We will discuss in the context of the WHERE condition, but the issue is of general importance

Choice Of Rows *An Example Statement*

Student	N_Number	Works	Has_SSN
	N1	No	No
	N2	No	Yes
	N3	Yes	No
	N4	Yes	Yes

- SQL statement (not standard SQL syntax):

```
SELECT N_Number
FROM R
WHERE (IF Works = 'Yes' THEN Has_SSN = 'Yes');
```

- Which rows are in the answer from the following?

Answer	N_Number
	N1
	N2
	N3
	N4

Choice Of Rows - The Negation Of The Example Statement

Student	N_Number	Works	Has_SSN
	N1	No	No
	N2	No	Yes
	N3	Yes	No
	N4	Yes	Yes

- SQL statement (not standard SQL syntax):

```
SELECT N_Number
FROM R
WHERE ( NOT(IF Works = 'Yes' THEN Has_SSN = 'Yes') );
```

- Which rows are in the answer from the following?

Answer	N_Number
	N1
	N2
	N3
	N4

Choice Of Rows - Answer To The Negated Statement

Student	N_Number	Works	Has_SSN
	N1	No	No
	N2	No	Yes
	N3	Yes	No
	N4	Yes	Yes

- SQL statement (not standard SQL syntax):

```
SELECT N_Number
FROM R
WHERE ( NOT(IF Works = 'Yes' THEN Has_SSN = 'Yes') );
```

- Rows in the Answer, intuitively,

Answer	N_Number
	N3

- Because N3 is the only one violating the original condition
  - N3 Works is satisfied but N3 Has_SSN is not satisfied

Choice Of Rows - Answer To The Original Statement

Student	N_Number	Works	Has_SSN
	N1	No	No
	N2	No	Yes
	N3	Yes	No
	N4	Yes	Yes

- SQL statement (not standard SQL syntax):

```
SELECT N_Number
FROM R
WHERE (IF Works = 'Yes' THEN Has_SSN = 'Yes');
```

- Rows in the Answer, ***complement*** of the previous Answer

Answer	N_Number
	N1
	N2
	N4

Choice Of Rows - Example Statement Using Given Operations

Student	N_Number	Works	Has_SSN
	N1	No	No
	N2	No	Yes
	N3	Yes	No
	N4	Yes	Yes

- SQL statement (standard SQL syntax):

```
SELECT N_Number
FROM R
WHERE Works = 'No' OR Has_SSN = 'Yes';
```

- Produces the correct Answer

Answer	N_Number
	N1
	N2
	N4

Conclusion

- We want to express a very natural condition

  IF antecedent THEN consequent

- Example:

  IF (the person works) THEN (the person has an ssn)

- But we only have NOT, OR, AND

  A person does not work, or the person has an ssn

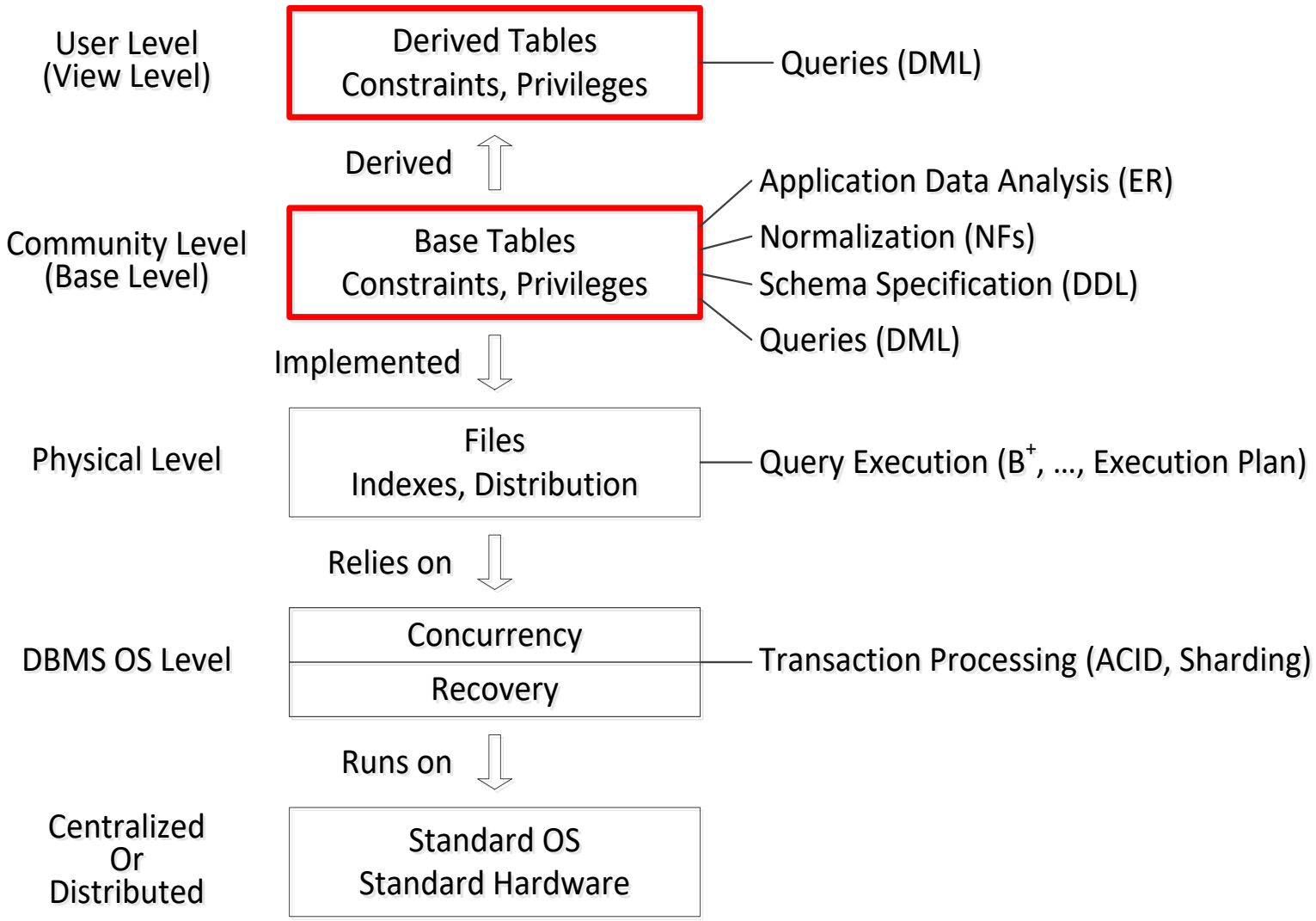
- NOT (the person works) OR (the person has an ssn)

- **NOT antecedent OR consequent**

Addendum 2 – Predicate Calculus vs. SQL

- SQL is Essentially Relational Algebra ++

DQL and DML in Context

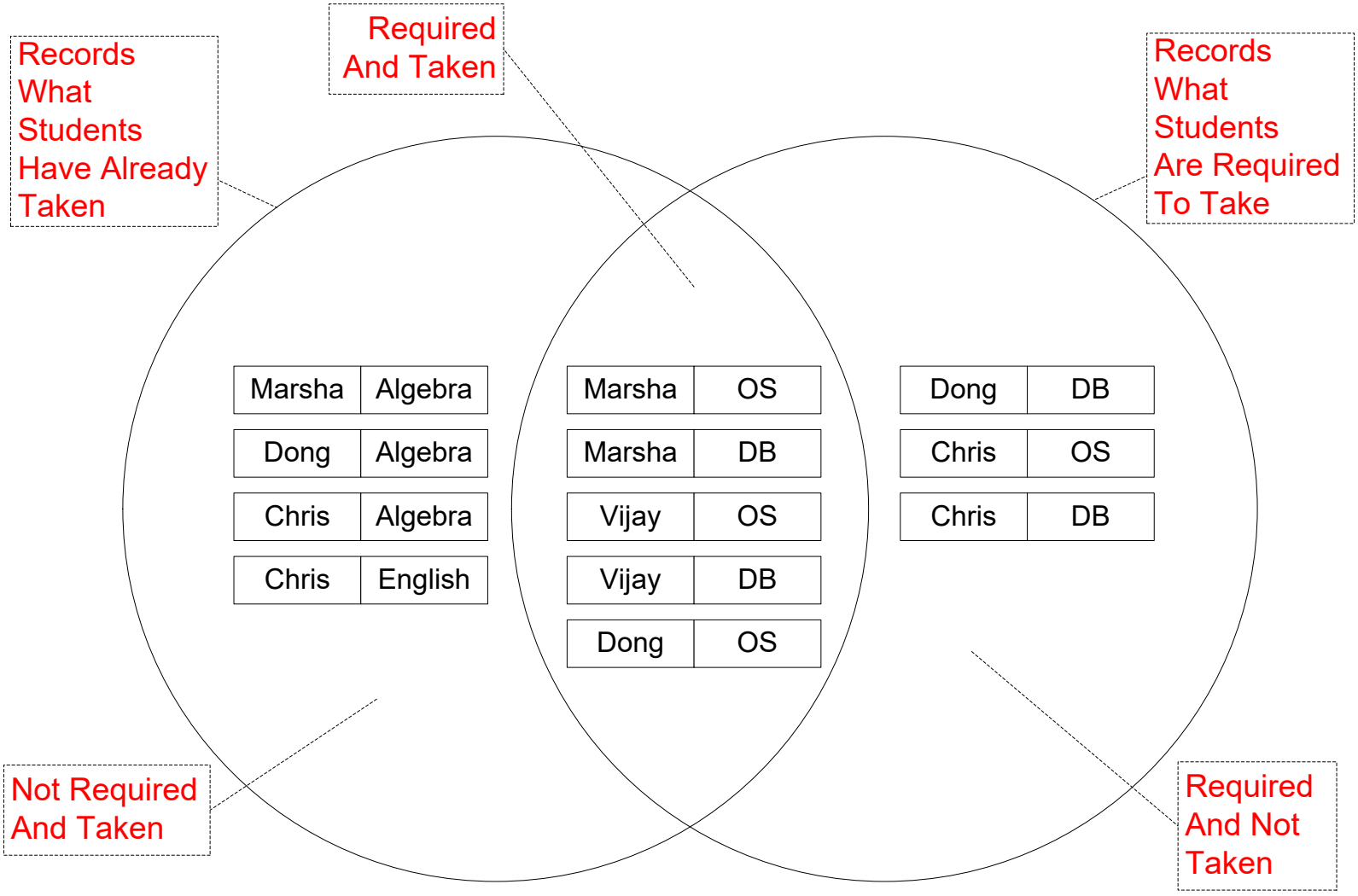


Back to Our Example

Took	Person	Course
	Marsha	OS
	Marsha	DB
	Marsha	Algebra
	Vijay	OS
	Vijay	DB
	Dong	OS
	Dong	Algebra
	Chris	Algebra
	Chris	English

Required	Course
	OS
	DB

Helpful Venn Diagram



Very Basic Predicate Calculus (Symbolic Logic)

- This is very important conceptually but rarely covered in Database Management Systems classes
- We covered (something you already knew) propositional calculus
  - » It deals with NOT, AND, OR
- We will cover (first-order) predicate calculus, also called first-order logic
  - » It extends propositional calculus to also deal with THERE EXISTS, FOR ALL
- We will learn why the operation of division (query asking about all) was so difficult to implement and we used double negations (implemented sometimes as double MINUS)
- But we will also see how to obtain such queries easily

Very Basic Predicate Calculus (First-Order Logic)

- To remind/review/learn about ***existential quantifiers***

$$\{y \mid \exists x[A(x, y)]\}$$

means the set of *y* for which there exists an *x* such that *A*(*x*,*y*) is true

- Example: the set of all integers that are squares
  The set of integers *y* for which there exist an integer x such that *y* is equal to x^2

$$\{y \mid \exists x[y = x^2]\}$$

Very Basic Predicate Calculus (First-Order Logic)

- To remind/review/learn about ***universal quantifiers***

$$\{y \mid \forall x[A(x, y)]\}$$

means the set of *y* for which for all *x*, *A*(*x*,*y*) is true

- Example: the set of all integers that are not squares

The set of integers *y* for which for every integer *x*, *y* is not equal to x^2

$$\{y \mid \forall x[y \neq x^2]\}$$

Can Convert a Universal Quantifier Into an Existential Quantifier

$$\{y \mid \forall x[y \neq x^2]\}$$

is the same as

$$\{y \mid \neg\exists x[y = x^2]\}$$

- Easy to understand intuitively why this is true

Expressing Implication Using a Negation and a Disjunction

- This you probably know, but let's review why $\alpha \rightarrow \beta$ is the same as (is equivalent to) $\neg\alpha \vee \beta$
- It is easier to think about the negation of $\alpha \rightarrow \beta$
- This intuitively means that α is true and β is false
- Or $\alpha \wedge \neg\beta$ is true
- The negation of this is $\neg(\alpha \wedge \neg\beta)$, that is $\neg\alpha \vee \beta$
- So, we have shown that

$$\alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$$

Some Useful Formulas

$$\alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$$

$$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$$

$$\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$$

$$\forall x[A(x)] \equiv \neg\exists x[\neg A(x)]$$

$$\exists x[A(x)] \equiv \neg\forall x[\neg A(x)]$$

Asking About Some

- *s* contains all the persons (students), *r* contains all the required courses, *t* lists which students took which courses
- List all Persons, who took ***at least one*** Required Course
- The result can be expressed using a logical formula with an ***existential quantifier***

$$\{p \mid p \in s \wedge \exists c[c \in r \wedge (p,c) \in t]\}$$

- We want every Person for which there is a Course in Required, such that (Person, Course) is in Took
- The standard SELECT … FROM … WHERE … easily expresses the existential quantifier above

```
SELECT Took.p
FROM Required, Took
WHERE Required.c = Took.c;
```

Asking About None

- List all Persons, who Took ***no*** Courses that are Required
- The result can be expressed using a logical formula with a ***universal quantifier***

$$\{p \mid p \in s \wedge \forall c[c \in r \rightarrow (p,c) \notin t]\}$$

- We can read this as
  We want every Person such that for every Course if Course is in Required then (Person, Course) is not in Took
- It is difficult to implement a universal quantifier in SQL (relational algebra)
  - » Difficult, but not impossible
  - » We have done that, though we did not think about that in a systematic way

Expressing Asking About None Using an Existential Quantifier

- We will start with our original expression and replace it by equivalent expressions
- On the right, we provide a justification for the step

$$\{p \mid p \in s \wedge \forall c[c \in r \rightarrow (p,c) \notin t]\}$$

$$\{p \mid p \in s \wedge \forall c[c \notin r \vee (p,c) \notin t]\} \qquad \alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$$

$$\{p \mid p \in s \wedge \neg\exists c\neg[c \notin r \vee (p,c) \notin t]\} \qquad \forall x[A(x)] \equiv \neg\exists x[\neg A(x)]$$

$$\{p \mid p \in s \wedge \neg\exists c[c \in r \wedge (p,c) \in t]\} \qquad \neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$$

- This what we did when we computed the query earlier using minus, which is really a negation in this context
- Using predicate calculus is easier than using relational algebra

Expressing Asking About None Using an Existential Quantifier (Highlighted)

- We will start with our original expression and replace it by equivalent expressions
- On the right, we provide a justification for the step

$$\{p \mid p \in s \wedge \forall c[c \in r \rightarrow (p,c) \notin t]\}$$

$$\{p \mid p \in s \wedge \forall c[c \notin r \vee (p,c) \notin t]\} \qquad \alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$$

$$\{p \mid p \in s \wedge \neg\exists c\neg[c \notin r \vee (p,c) \notin t]\} \qquad \forall x[A(x)] \equiv \neg\exists x[\neg A(x)]$$

$$\{p \mid p \in s \wedge \neg\exists c[c \in r \wedge (p,c) \in t]\} \qquad \neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$$

- This what we did when we computed the query earlier using a SELECT and minus, which is really a negation in this context

Asking About All

- List all Persons, who Took ***at least all*** the Courses that are Required
- The result can be expressed using a logical ***formula*** with a universal quantifier:

$$\{p \mid p \in s \wedge \forall c[c \in r \rightarrow (p,c) \in t]\}$$

- We can read this as

  We want every Person such that for every Course if Course is in Required then (Person, Course) is in Took

- It is not easy to express this using the standard SELECT … FROM … WHERE …

Expressing Asking About All Using an Existential Quantifier

- We will start with our original expression and replace it by equivalent expressions
- On the right, we provide a justification for the step

$$\{p \mid p \in s \wedge \forall c[c \in r \rightarrow (p,c) \in t]\}$$

$$\{p \mid p \in s \wedge \forall c[c \notin r \vee (p,c) \in t]\} \qquad \alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$$

$$\{p \mid p \in s \wedge \neg\exists c\neg[c \notin r \vee (p,c) \in t]\} \qquad \forall x[A(x)] \equiv \neg\exists x[\neg A(x)]$$

$$\{p \mid p \in s \wedge \neg\exists c[c \in r \wedge (p,c) \notin t]\} \qquad \neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$$

- This what we did when we computed division earlier, using “double negation”
- It was also probably difficult to derive
- Here we have obtained the result “automatically”

Expressing Asking About All Using an Existential Quantifier

- We will start with our original expression and replace it by equivalent expressions
- On the right, we provide a justification for the step

$$\{p \mid p \in s \wedge \forall c[c \in r \rightarrow (p,c) \in t]\}$$

$$\{p \mid p \in s \wedge \forall c[c \notin r \vee (p,c) \in t]\} \qquad \alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$$

$$\{p \mid p \in s \wedge \neg\exists c\neg[c \notin r \vee (p,c) \in t]\} \qquad \forall x[A(x)] \equiv \neg\exists x[\neg A(x)]$$

$$\{p \mid p \in s \wedge \neg\exists c[c \in r \wedge (p,c) \notin t]\} \qquad \neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$$

- This what we did when we computed division earlier, using "double negation"
- And we can easily convert the last formula into a sequence of simple SQL queries, which we discussed earlier in the class

SQL and Predicate Calculus

- SQL (and relational algebra) implement the existential quantifier essentially directly
- SQL (and relational algebra) do not implement the universal quantifier directly, so it needs to be expressed essentially as we did, using the existential quantifier
- There were proposals to actually use predicate calculus and not relational algebra to query databases
- But it was believed that programmers would not want to write logical expressions

- Omitting some minor technicalities, relational algebra is equivalent in expressive power to predicate calculus
- Proof of the equivalence is rather easy, though tedious, using transformations like what we have just used
- But predicate calculus is more natural (I think), once you learn it

Conclusion

- Predicate calculus is worth knowing
- It is useful in many settings

Summary (1/2)

- A relation is a set of rows in a table with labeled columns
- Relational algebra as the basis for SQL
- Basic operations:
  - » Union (requires union compatibility)
  - » Difference (requires union compatibility)
  - » Intersection (requires union compatibility); technically not a basic operation
  - » Selection of rows
  - » Selection of columns
  - » Cartesian product
- These operations define an algebra: given an expression on relations, the result is a relation (this is a “closed” system)
- Combining this operations allows production of sophisticated queries

Summary (2/2)

- Relational algebra is not universal: We can write programs producing queries that cannot be specified using relational algebra (cannot compute some useful answers such as transitive closure)
- Using surrogate keys
- We focused on relational algebra specified using SQL syntax, as this is more important in practice
- The other, “more mathematical” notation came first and is frequently used in research and other venues, but not commercially

Agenda

1. Session Overview
2. Relational Algebra and Relational Calculus
3. Relational Algebra Using SQL Syntax
4. Summary and Conclusion

Summary

- Relational algebra and relational calculus are formal languages for the relational model of data
- A relation is a set of rows in a table with labeled columns
- Relational algebra and associated operations are the basis for SQL
- These relational operations define an algebra
- Relational algebra is not universal as it is possible to write programs producing queries that cannot be specified using relational algebra
- Relational algebra can be specified using SQL syntax
- The other, “more mathematical” notation came first and is used in research and other venues, but not commercially

Assignments & Readings

- Readings
  - Slides and Handouts posted on the course web site
  - Textbook: Chapters 6, and 8

- Assignment #5
  - Textbook exercises: 6.10, 6.16, 8.20, 8.27, 8.28, 8.33
- Project Framework Setup (ongoing)

Next Session: Standard Query Language (SQL) Features

- SQL as implemented in commercial databases

Any Questions?