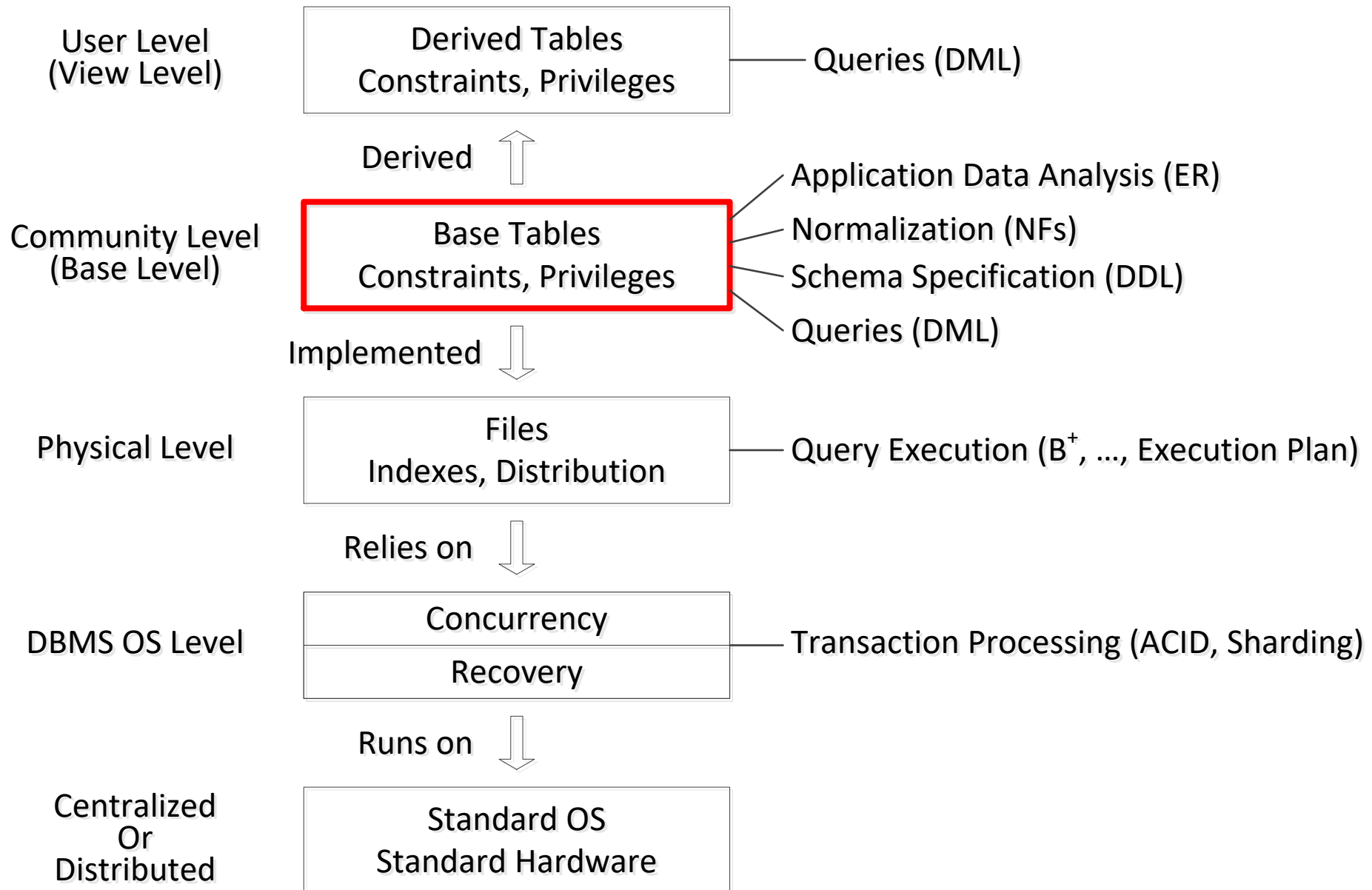


***Unit 4***  
***Relational Algebra :***  
***Data Manipulation Language for Relations***

# Relational Algebra in Context



# ***Introduction***

# ***Relational Algebra And SQL***

- ***SQL is based on relational algebra with many extensions***
  - Some necessary
  - Some unnecessary
- “Pure” relational algebra uses mathematical notation with Greek letters
- I will cover it using SQL syntax; that is in this unit I will cover relational algebra, but it will look like SQL
  - And will be really valid SQL
- Pure relational algebra is used in research, scientific papers, and some textbooks (mainly for language independence)
- Sometimes it’s good to see what the fundamental operations are.
- Skip to page 66 (optionally read up to that)

# ***Relations***

# 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, difference is frequently denoted by a symbol similar to a backslash:  $A \setminus B$
- $\times$  Cartesian product,  $A \times B = \{ (x,y) \mid x \in A \wedge y \in B \}$ ,  $A \times B \times C = \{ (x,y,z) \mid x \in A \wedge y \in B \wedge z \in C \}$ , 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 sets**
- 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***

## ***Fundamental SQL Operations***

Dennis class skip to page 66

# ***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, so do not worry about efficiency now

# 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)
- A very important property: ***Any operation on relations produces a relation***
- This is why we call our structure an ***algebra*** (meaning a structure with operations closed under those operations)
  - Example for a structure that is not an algebra: positive integers under the operation of subtraction;  $3 - 4$  is not a positive integer

## ***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 the file
  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., a condition)

|  | A | B | C | D |
|--|---|---|---|---|
|  | 5 | 5 | 7 | 4 |
|  | 4 | 5 | 4 | 4 |

## ***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 the file
  3. Check if a record satisfies some conditions on the values of the field
  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

# Selection

- Note: *If the condition is empty (there is no WHERE clause) then it is satisfied by every tuple)*
- Intuitively can think as follows:

If there is no reason to reject a tuple then accept it

# 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



# 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 is 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 is 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 programming language 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 relation, to use another name to denote it
  - Renaming of a column, to use another name to denote it

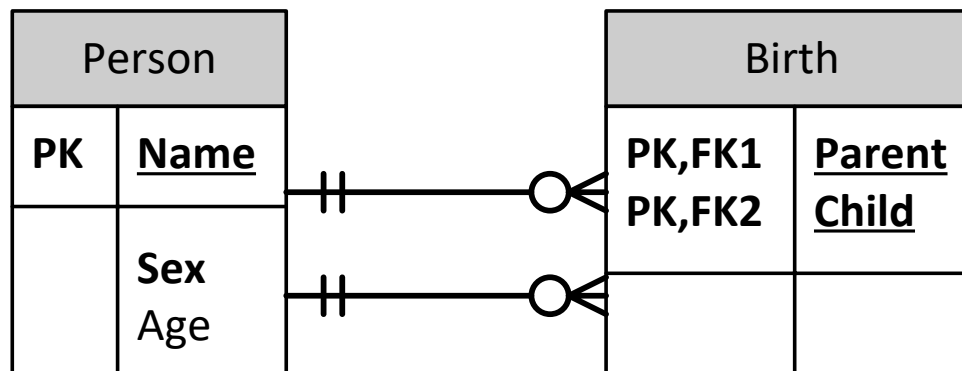
# ***Specifying Queries Using Fundamental Operations***

# ***A Small Example***

- The example consists of 2 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 the mother and the father could be listed, or two mothers, etc.)
  - 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

- 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 “understand” that Name identifies a person



# ***Microsoft Access Database***

- Microsoft Access Database with this example has been uploaded
- I suggest that you download and install Microsoft Access, but this is not required
  - You will be told how to get Microsoft Access
- I used Access because it is a very good tool for quickly demonstrating basic constructs of SQL DML, although it is not suitable for anything other than personal databases and even there it suffers from various defects
- I do not recommend using Access “for production,” but may be useful for practicing relational algebra/SQL on your own

# ***Microsoft Access Database***

- The database and our queries (other than the one with operator MINUS at the end) are on the NYU Classes web site
- Note
  - MINUS is frequently specified in commercial databases in a roundabout way
  - I leave the discussion of how it is done to when we discuss commercial databases
- Our sample Access database: People.mdb
- I ran the queries in Microsoft Access and copied and pasted them in these notes, after reformatting them
- I 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***

| 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 |

# ***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 |

# *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

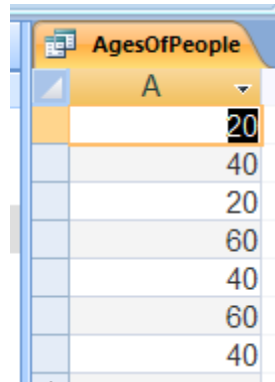
FROM Person;

|  | A  |
|--|----|
|  | 20 |
|  | 40 |
|  | 20 |
|  | 60 |
|  | 40 |
|  | 60 |
|  | 40 |

- Recall that whether duplicates are removed or not is not important (at least for the time being in our course, as we study relational algebra)

# ***The Query in Microsoft Access***

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



A screenshot of a Microsoft Access query result grid. The grid has a title bar that says 'AgesOfPeople' and a column header 'A'. The first row is highlighted in orange and contains the value '20'. The subsequent rows contain the values '40', '20', '60', '40', '60', and '40'. The grid is partially obscured by a blue sidebar on the left.

| A  |
|----|
| 20 |
| 40 |
| 20 |
| 60 |
| 40 |
| 60 |
| 40 |

## ***A Query***

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

SELECT N

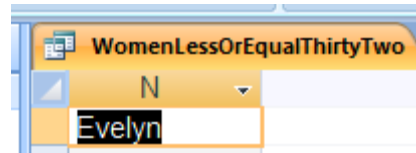
FROM Person

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 followed by a tuple in Birth
- For our purpose, the two tuples matched if N(ame) in Person is C(hild) in Birth and the S(ex) in Person is F

# *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 that **AS** was an attribute-renaming operator

# ***Cartesian Product With Condition: Matching Tuples Indicated***

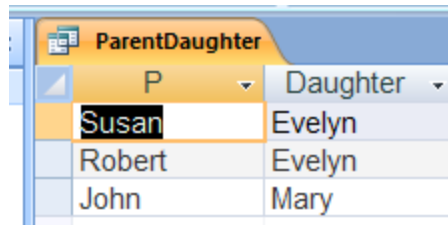
| Person | N | S | A  |
|--------|---|---|----|
| Albert | M |   | 20 |
| Dennis | M |   | 40 |
| Evelyn | F |   | 20 |
| John   | M |   | 60 |
| Mary   | F |   | 40 |
| Robert | M |   | 60 |
| Susan  | F |   | 40 |

| 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



A screenshot of a Microsoft Access query result table. The table has a title bar that says 'ParentDaughter'. Below the title bar, there are two columns: 'P' and 'Daughter'. The 'P' column has three rows of data: 'Susan', 'Robert', and 'John'. The 'Daughter' column has three rows of data: 'Evelyn', 'Evelyn', and 'Mary'. The first row is highlighted in orange.

| P      | Daughter |
|--------|----------|
| Susan  | Evelyn   |
| Robert | Evelyn   |
| John   | Mary     |

## *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(ex) of the Parent and the S(ex) 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 are referring
- Again, we use **AS** to serve as a renaming operator, this time for relations
- Note: We could have used what we have already computed: Answer(Parent, Daughter) to constraint 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';
```

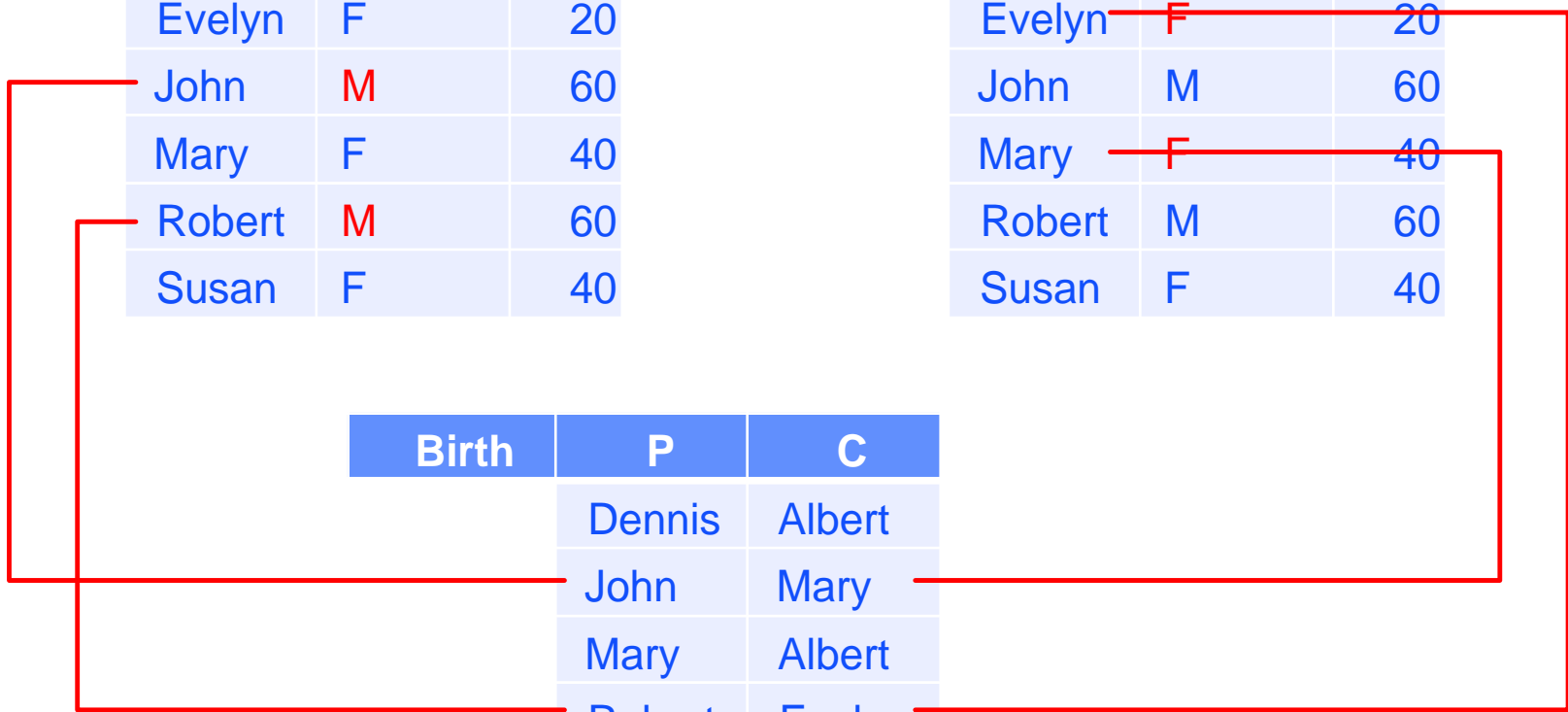
|  | Father | Daughter |
|--|--------|----------|
|  | John   | Mary     |
|  | Robert | Evelyn   |

# 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 |



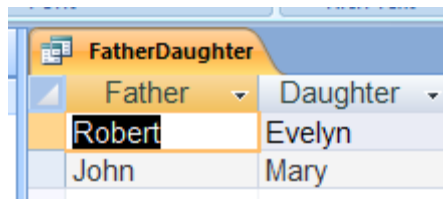
# ***Thinking Like a Programmer May Help in Writing Relational Algebra Queries***

- 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 of 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



A screenshot of a Microsoft Access query result. The query is named 'FatherDaughter'. It displays a table with two columns: 'Father' and 'Daughter'. The 'Father' column has two entries: 'Robert' and 'John'. The 'Daughter' column has two entries: 'Evelyn' and 'Mary'. The table is displayed in a grid format with a light blue header and a light gray body.

| Father | Daughter |
|--------|----------|
| Robert | Evelyn   |
| John   | Mary     |

## ***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 |

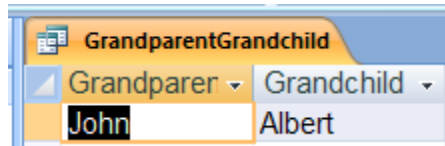
## ***Cartesian Product With Condition: Matching Tuples Indicated***

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



A screenshot of a Microsoft Access query result table. The table has a title bar that says 'GrandparentGrandchild'. Below the title bar, there are two columns: 'Grandparent' and 'Grandchild'. The 'Grandparent' column has a dropdown arrow and the value 'John'. The 'Grandchild' column has a dropdown arrow and the value 'Albert'.

| Grandparent | Grandchild |
|-------------|------------|
| John        | Albert     |

## ***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<sup>x</sup>-grandparent, Great<sup>x</sup>-grandchild), for any  $x$
- Ultimately, may want (Ancestor, Descendant)

# ***Relational Algebra Is not Universal: Cannot Compute (Ancestor, Descendant)***

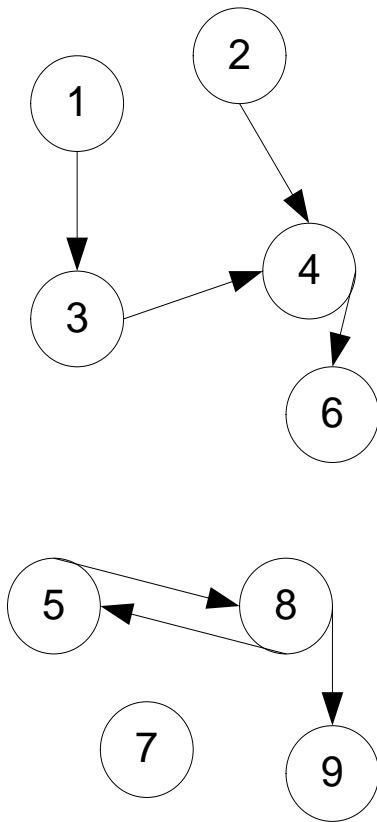
- Standard programming languages are **universal**
- This roughly means that they are as powerful as Turing machines, if unbounded amount of storage is permitted (you will never run out of memory)
- This roughly means that they can compute anything that can be computed by any computational machine we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

# ***Relational Algebra is not Universal: Cannot Compute (Ancestor, Descendant)***

- It is impossible in relational algebra (or 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
- This is not a contrived example because it shows that we cannot compute the transitive closure of a directed graph: the set of all the 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   | 1    | 3  |
| 2   | 2    | 4  |
| 3   | 3    | 4  |
| 4   | 4    | 6  |
| 5   | 5    | 8  |
| 8   | 8    | 5  |
| 8   | 8    | 9  |

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



## ***A Sample Query***

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

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

# *It Does not Matter If We Remove Duplicates*

- Removing duplicates

|  |    |   |    |   |    |
|--|----|---|----|---|----|
|  | A  |   | A  |   | A  |
|  | 20 |   | 20 |   | 60 |
|  | 40 | — | 40 | = |    |
|  | 60 |   |    |   |    |

- Not removing duplicates

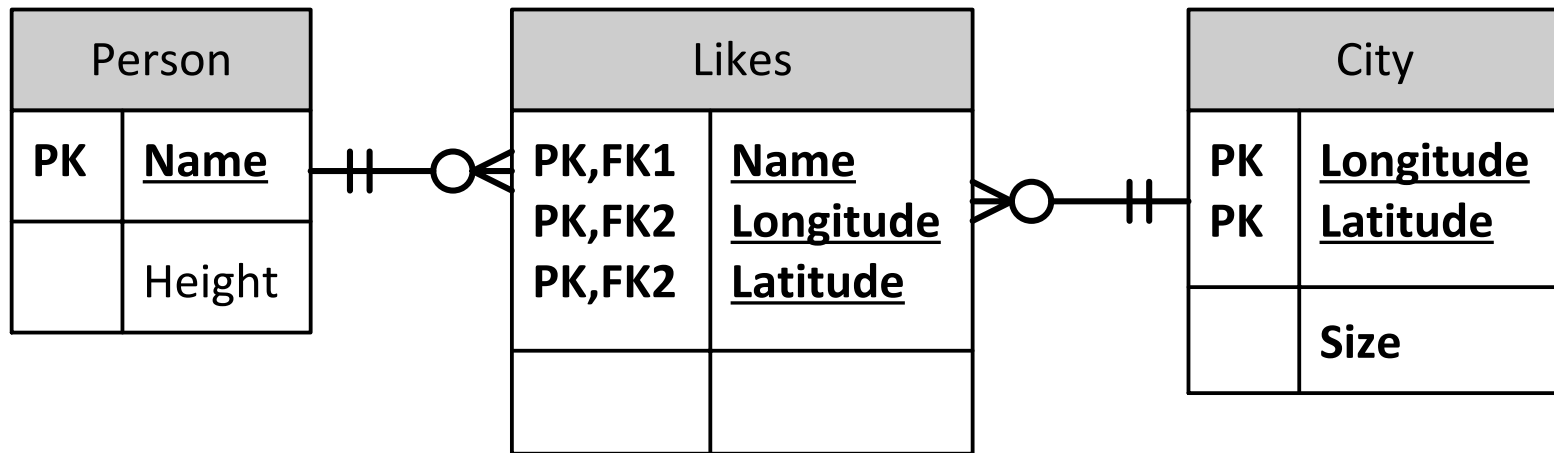
|  |    |   |    |   |    |
|--|----|---|----|---|----|
|  | A  |   | A  |   | A  |
|  | 20 |   | 20 |   | 60 |
|  | 40 |   | 40 |   | 60 |
|  | 60 | — | 40 | = |    |
|  | 60 |   |    |   |    |

# ***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!***

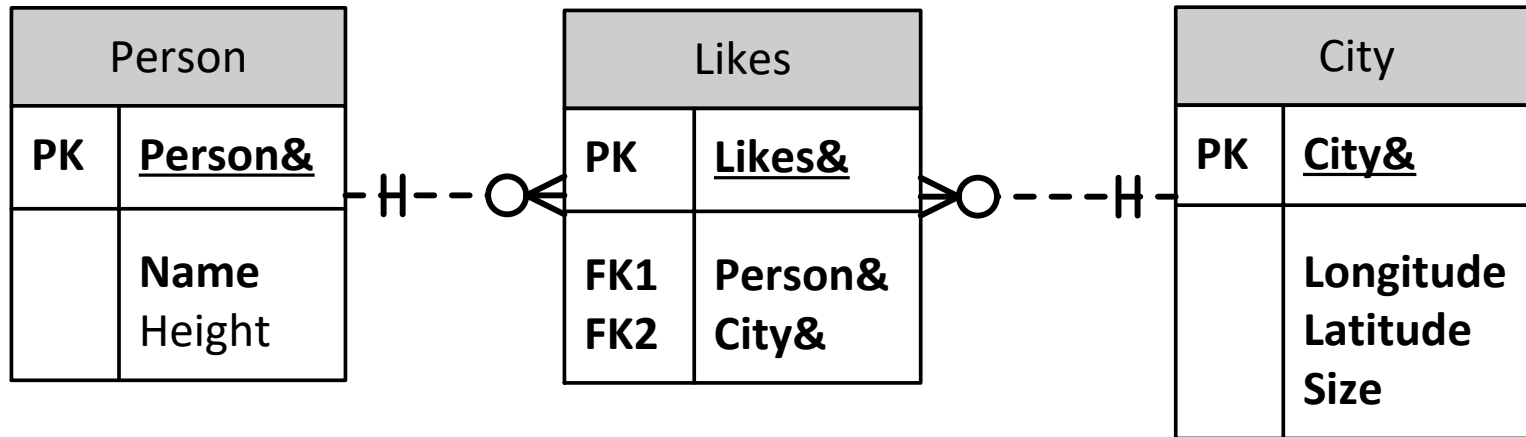
# ***Surrogate Keys And Queries***

# 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 Person.Height = 6;
```

# Querying With Surrogate Primary Keys



- 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 Person.Height = 6;
- Comment: Probably add in the specifications that
  - Name is not NULL and is UNIQUE
  - Longitude and Latitude are not NULL and (Longitude, Latitude) is UNIQUE

Unless this may no longer hold in the future

# ***Relational Algebra Using Standard Relational Algebra Mathematical Notation***



## ***Now To “Pure” Relational Algebra***

- I am giving a description 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 more theoretical resources use relational algebra instead of SQL
- This notation came first, before SQL was invented, when relational databases were just a theoretical construct

## $\pi$ : *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
```

Relational Algebra

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

|  | B  | A | D    |
|--|----|---|------|
|  | 10 | 1 | 1000 |
|  | 20 | 1 | 1000 |
|  | 20 | 1 | 1000 |

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

## $\sigma$ : 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  $\pi$

|  | 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 or Cross-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 (cross-product then restriction = join)*

| 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

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

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

## $\cup$ : *Union*

| R | A | B  |
|---|---|----|
|   | 1 | 10 |
|   | 2 | 20 |

| S | A | B  |
|---|---|----|
|   | 1 | 10 |
|   | 3 | 20 |

- SQL statement

```
SELECT *  
FROM R  
UNION  
SELECT *  
FROM S
```

### Relational Algebra

$R \cup S$

|  | A | B  |
|--|---|----|
|  | 1 | 10 |
|  | 2 | 20 |
|  | 3 | 20 |

- 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
```

### Relational Algebra

$R - S$

|  | A | B  |
|--|---|----|
|  | 2 | 20 |

- Union compatibility required



## $\cap$ : Intersection

| R | A | B  |
|---|---|----|
|   | 1 | 10 |
|   | 2 | 20 |

| S | A | B  |
|---|---|----|
|   | 1 | 10 |
|   | 3 | 20 |

- SQL statement

```
SELECT *  
FROM R  
INTERSECT  
SELECT *  
FROM S
```

### Relational Algebra

$R \cap S$

|  | A | B  |
|--|---|----|
|  | 1 | 10 |

- Union compatibility required

## ***Performance Considerations.***

- Different orders can give vastly different performance.
- Without indexes, try to do selections before joins.
- Try to do most limiting selections first.
- When indexes are involved, there are tradeoffs, because doing the select will lose the index.
- Also selectivity plays a big role (bond interest example).

# ***Key Ideas***

# ***Key Ideas***

- 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 these operations allows construction of sophisticated queries