Unit 4
Relational Algebra (Using SQL Syntax): Data Manipulation Language For Relations

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
- So it is good to know it, and I provide at the end of this unit material from which one can learn it
- But in anything practical, including commercial systems, you will be using SQL

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, $A - B = \{ x \mid x \in A \ \wedge \ x \notin B \}$
- $\times$ 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 $(A \cap B) \times (C \times A)$

  So you can write expressions and not just programs!

Relations in Relational Algebra

- Relations are sets of tuples, the latter being also called ***rows***, drawn from some domains
- 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

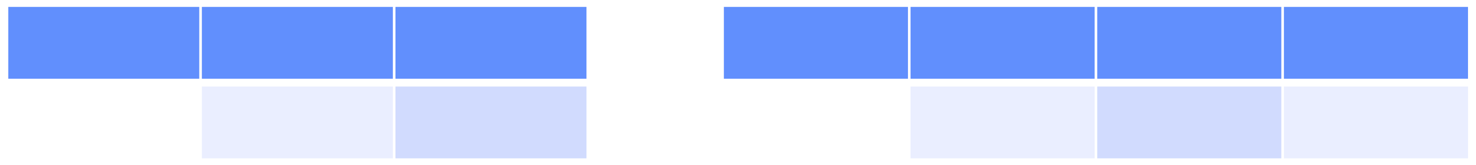
- The order of rows and whether a row appears once or many times does not matter
- The order of columns matters, but as our columns will always be labeled, we will be able to reconstruct the order even if the columns are permuted.
- The following two relations are equal:

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

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

- 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 (i.e. respective values)
  - 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;
```

(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

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#, even though they happen to be equal due to the specific equality condition
- We could, as we will see later, rename a column, to get Room#

	ID#	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

- 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
- Actually, best to assume that the column names are the same and that is what we will do from now on
- We refer to these as ***union compatibility*** of relations
- Sometimes, just the term ***compatibility*** is used

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 programming language type of operations:
  - Assignment of an expression to a new variable;

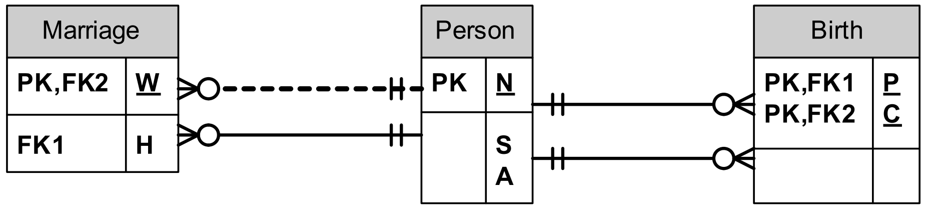
    In our case assignment of a relational expression to a relational variable.
  - Renaming of a relation, to use another name to denote it
  - Renaming of a column, to use another name to denote it

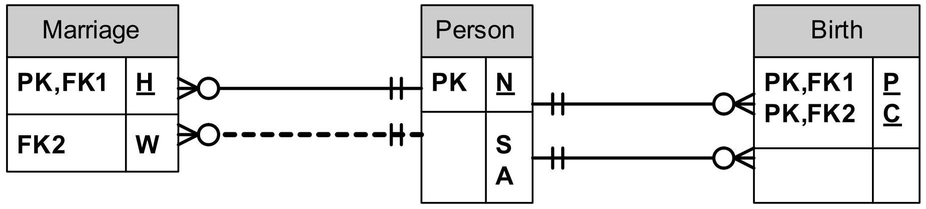
A Small Example

- The example consists of 3 relations:
  - Person(<u>Name</u>, Sex, Age)
    - This relation, whose primary key is Name, gives information about the human's sex and age
  - Birth(<u>Parent</u>, <u>Child</u>)
    - 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 could be listed.)
  - Marriage(<u>Husband</u>, Wife, Age) **or**
  - Marriage(Husband, <u>Wife</u>, Age)
    - This relation listing current marriages only, assumes that only opposite sex marriages are listed (so we can correlate simple queries based on the sex of people) 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. In the example schema, we only store marriages between a man and a woman.
  - 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

- 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

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(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)

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

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

A Query

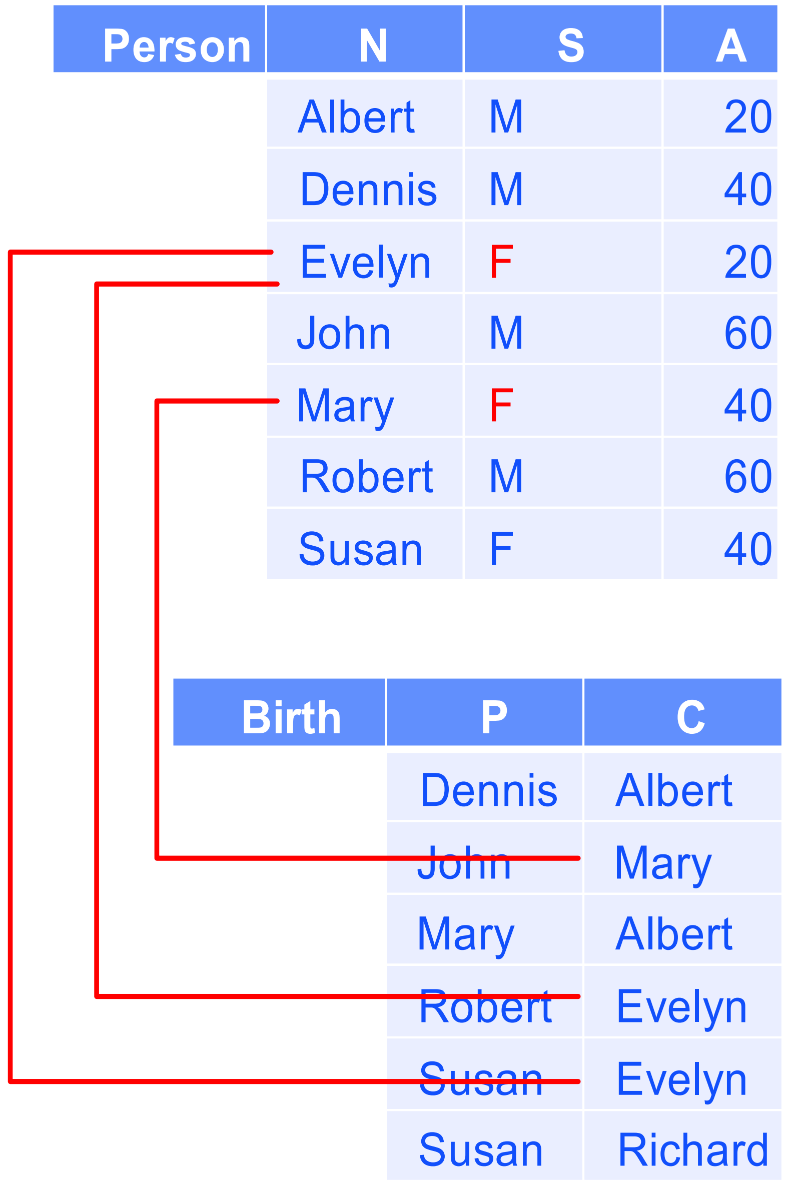
Answer:=

```
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 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

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, this time for relations

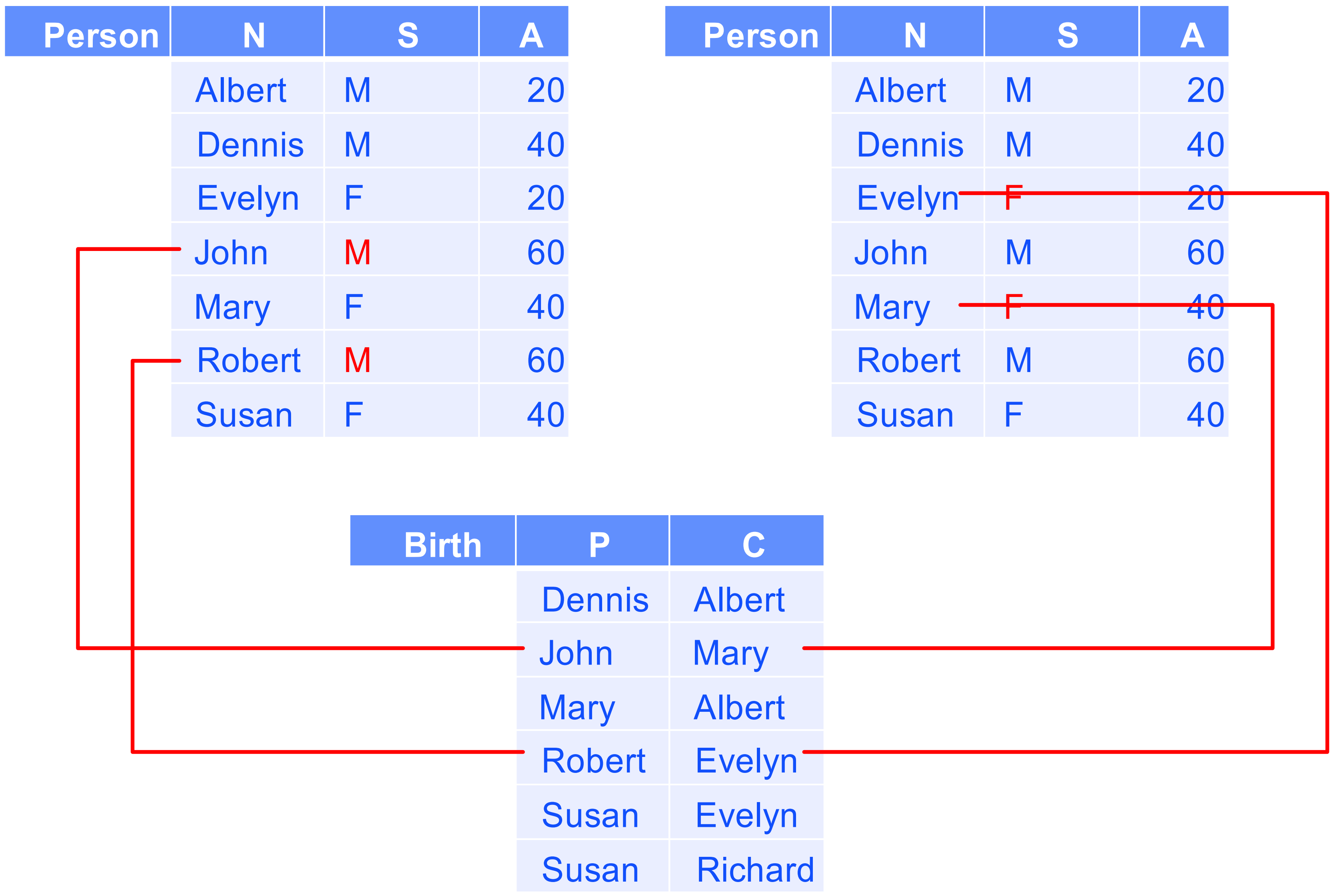
- Note: We could have used what we have already computed: Parent,Daughter

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

A Query

- Produce a relation: Answer(Father_in_law, Son_in_law).
- 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

A Query

- Produce a relation: Answer(Grandparent,Grandchild)

	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

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 (or standard SQL) to compute the relation Answer(Ancestor, Descendant)
- Why?
- 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

A Sample Query

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

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

It Does Not Matter If We Remove Duplicates

- Removing duplicates

	A
	20
	40
	60

−

	A
	20
	30

=

	A
	40
	60

- Not removing duplicates

	A
	20
	40
	20
	60
	40
	60
	40

−

	A
	20
	30

=

	A
	40
	60
	40
	60
	40

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!***

Next

- SQL as implemented in commercial databases

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

Key Ideas

- Relational algebra is not universal: cannot compute some useful answers such as transitive closure
- 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 used in research and other venues, but not commercially

Relational Algebra Using Standard Relational Algebra Mathematical Notation

Now To “Pure” Relational Algebra

- I gave a description in several slides
- 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
```

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

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

  ¬, ∧, and ∨ 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

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

	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

Relational Algebra

R ∪ 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

∩: *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