Simplifications:

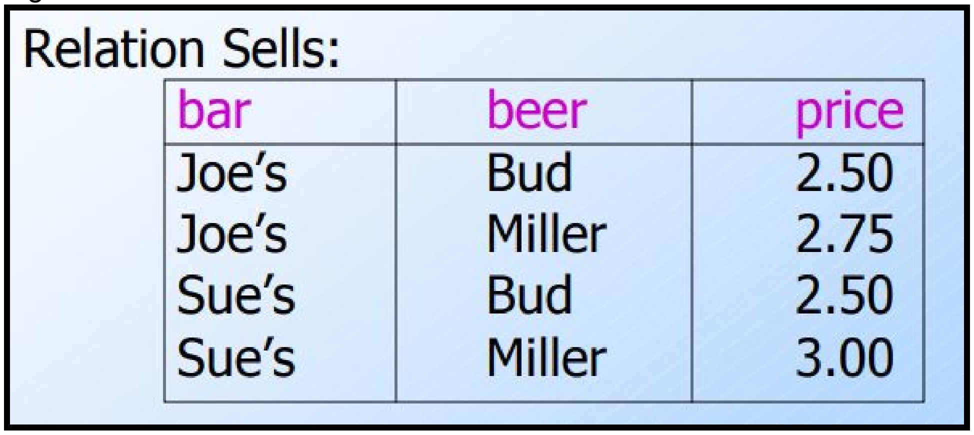
- While learning relational algebra, we will assume the following:
  - Relations are sets, so now two rows are the same.
  - Every cell has a value.
- In SQL, we will drop these assumptions.
- But for now, they simplify our queries.

Relational Algebra Basics:

- Relational algebra is an algebra whose operands are relations or variables that represent relations. Furthermore, operators are designed to do the most common things that we need to do with relations in a database. The result is an algebra that can be used as a query language for relations.
- In relation algebra, operands are tables and operators are what we can do with those tables.
  Some operators are:
  - Select
  - Project
  - Cartesian Product
  - Joins

Select:

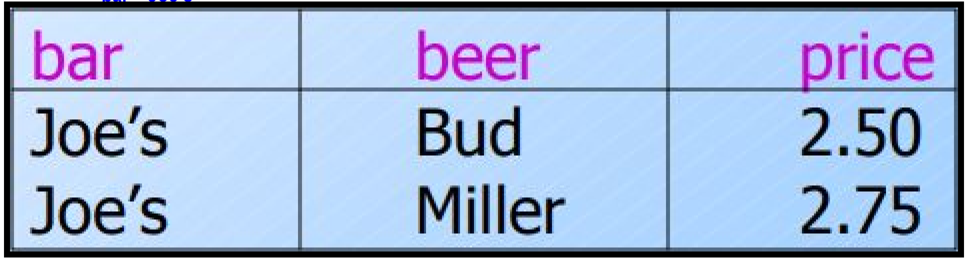
- Used for selecting rows based on some condition.
- Notation: **$\sigma_c(R)$**
- R is a table.
- Condition c is a boolean expression. It can use comparison operators and boolean operators. The operands are either constants or attributes of R.
- The result is a relation with the same schema as the operand but with only the tuples that satisfy the condition.
- E.g. Consider the relation below:



Relation Sells:

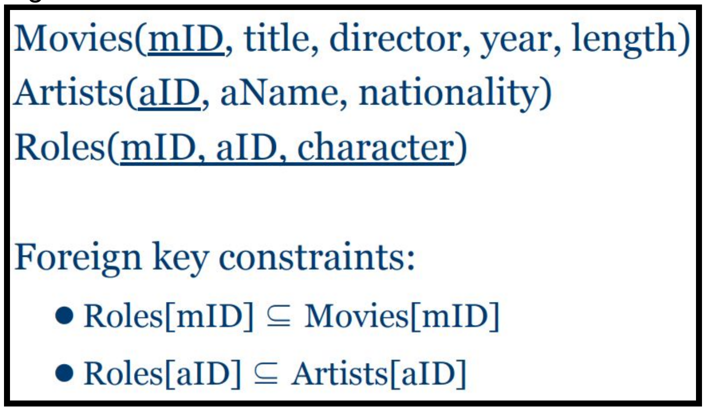
bar	beer	price
Joe’s	Bud	2.50
Joe’s	Miller	2.75
Sue’s	Bud	2.50
Sue’s	Miller	3.00

If I do **$\sigma_{bar=“Joe’s”}$(Sells)**, the output or result is:



bar	beer	price
Joe’s	Bud	2.50
Joe’s	Miller	2.75

- E.g. Consider the schema below:



Movies(mID, title, director, year, length)
Artists(aID, aName, nationality)
Roles(mID, aID, character)

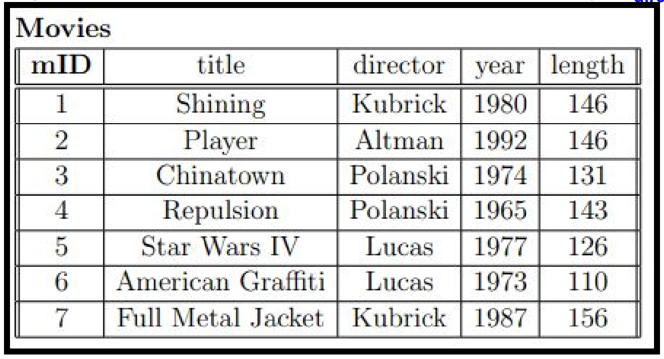
Foreign key constraints:

- Roles[mID] $\subseteq$ Movies[mID]
- Roles[aID] $\subseteq$ Artists[aID]

To find all British actors, I will do the following query: $\sigma_{nationality=\text{“British”}}$**(Artists)**.
To find all the movies from the 1970s, I will do the following query:
$\sigma_{year \geq 1970 \land year < 1980}$ **(Movies)**.

Project:

- Used for selecting columns based on some condition.
- Notation: $\pi_L(R)$
- R is a table.
- L is a subset, not necessarily a proper subset, of the attributes of R.
  I.e. L is a list of attributes that are in R. It could be 1 column or all the columns.
- The result is a relation with all the tuples from R but with only the attributes in L, and in that order.
- It's called project because it gets the columns.
- **Note:** The outcome of the project operator is a set, and thus, there can be no duplicate values.
  E.g. Consider the relation below and the query $\pi_{director}$**(Movies)**:



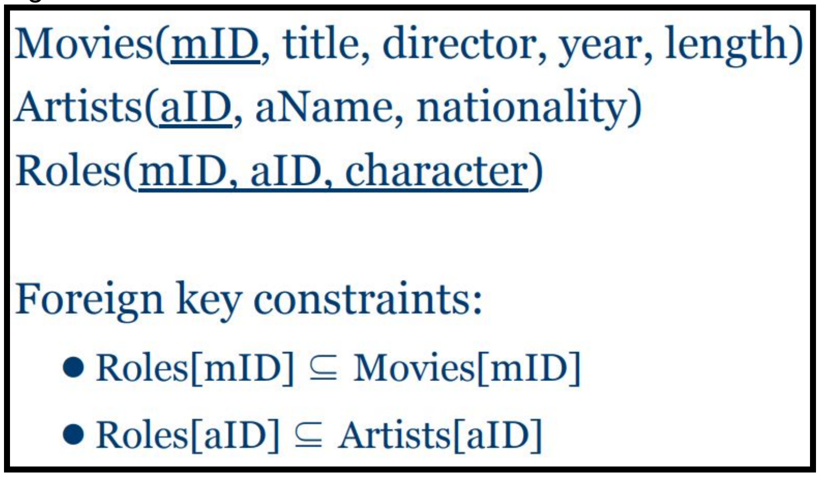
Movies

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polanski	1974	131
4	Repulsion	Polanski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156

The output of the above query is:

Director
Kubrick
Altman
Polanski
Lucas

- E.g. Consider the schema below:



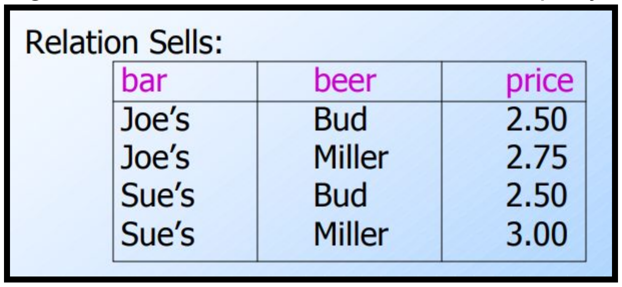
Movies(mID, title, director, year, length)
Artists(aID, aName, nationality)
Roles(mID, aID, character)

Foreign key constraints:

- Roles[mID] ⊆ Movies[mID]
- Roles[aID] ⊆ Artists[aID]

To find the names of all directors of movies from the 1970s, I will do the following query $\pi_{director}(\sigma_{year \geq 1970 \wedge year<1980}(Movies))$. This is called a **nested query**.

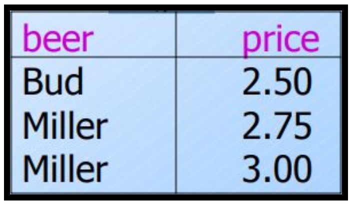
- E.g. Consider the relation below and the query $\pi_{beer,price}(Sells)$:



Relation Sells:

bar	beer	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Miller	3.00

The result relation is:



beer	price
Bud	2.50
Miller	2.75
Miller	3.00

Cartesian Product:

- Notation: **R1 x R2**
- The result is a relation with every combination of a tuple from R1 concatenated to a tuple from R2.
- Its schema is every attribute from R followed by every attribute of S, in order.
- Suppose there are m attributes in R1 and n attributes in R2. Then, **R1 x R2** has m*n tuples.
- **Note:** If an attribute occurs in both relations, it occurs twice in the result prefixed by relation name.

  E.g. Consider the relations below and the query **R1 x R2**:

  R1

A	**B**
1	2
3	4

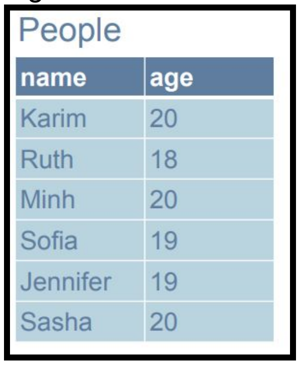
R2

B	C
3	5
4	7

The result is:

A	R1.B	R2.B	C
1	2	3	5
1	2	4	7
3	4	3	5
3	4	4	7

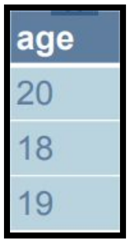
- **Note:** Projecting onto fewer attributes can remove what it was that made two tuples distinct. This is because wherever a project operation might "introduce" duplicates, only one copy of each is kept.
  E.g. Consider the relation below and the query **π_{age}(People)**:



People

name	age
Karim	20
Ruth	18
Minh	20
Sofia	19
Jennifer	19
Sasha	20

The result of the query is



age
20
18
19

- Cartesian product can be inconvenient as it can introduce nonsense tuples.

Natural Join:

- Notation: **R ⋈ S**
- The result is defined by:
  - Taking the Cartesian product.
  - Selecting to ensure equality on attributes that are in both relations (as determined by name).
  - Projecting to remove duplicate attributes.

- Some properties of natural joins are:
    1. **Commutative:** **$R \bowtie S = S \bowtie R$**
       Although attribute order may vary.
       This will matter later when we use set operations.
    2. **Associative:** **$R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$**
       So when writing n-ary joins, brackets are irrelevant.
       We can just write: **$R1 \bowtie R2 \bowtie . . . \bowtie Rn$**
- **Note:** If R and S don't have share any attribute(s) with the same name, then **$R \bowtie S$** will be the same as **R x S**.
- E.g. Consider the relations below and the query **$R \bowtie S$**:

  R

A	**B**
1	2
2	3

  S

C	**D**
5	6
7	9

  The result is:

A	**B**	**C**	**D**
1	2	5	6
1	2	7	9
2	3	5	6
2	3	7	9

  Here, since R and S don't share an attribute with the same name, **$R \bowtie S$** is the same as **R x S**.

- E.g. Consider the relations below and the query **R ⋈ S**:

R

Number	Square
1	1
2	4

S

Number	Cube
1	1
2	8

The result is:

Number	Square	Cube
1	1	1
2	4	8

- E.g. Consider the relations below and the query **R ⋈ S**:

R

Number	Square
1	1
3	9

S

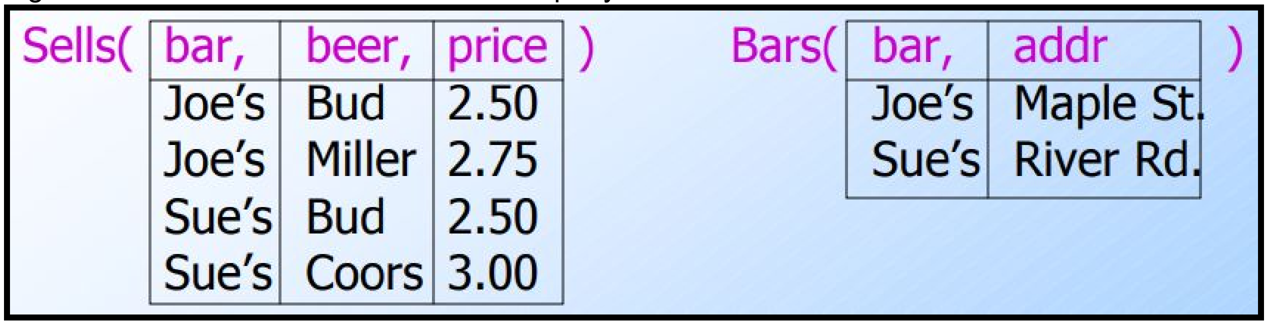
Number	Cube
1	1
2	8

The result is:

Number	Square	Cube
1	1	1

Here, because R.Number doesn't have 2 and S.Number doesn't have 3, both rows get omitted from the result.

- E.g. Consider the relations below and the query **Sells ⋈ Bars**:



Sells(

bar,	beer,	price
Joe's	Bud	2.50
Joe's	Miller	2.75
Sue's	Bud	2.50
Sue's	Coors	3.00

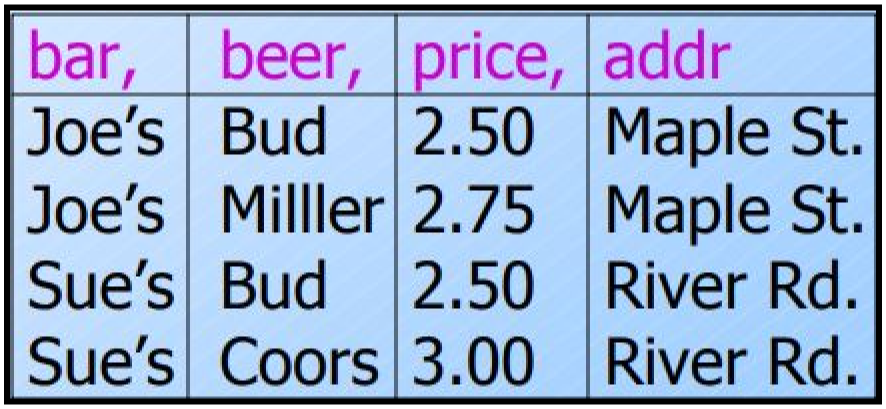
)

Bars(

bar,	addr
Joe's	Maple St.
Sue's	River Rd.

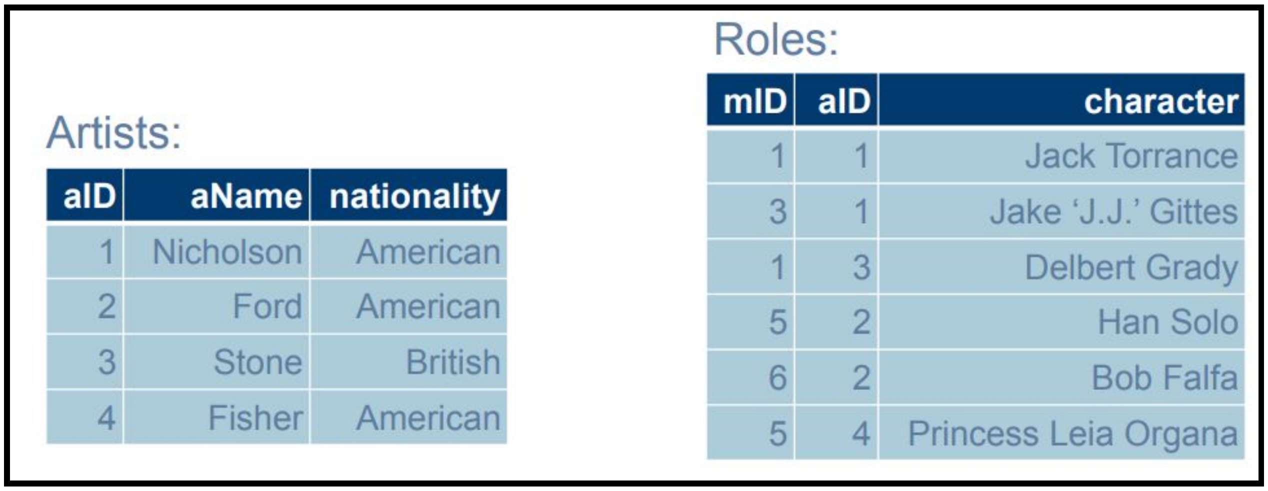
)

The result is:



bar,	beer,	price,	addr
Joe's	Bud	2.50	Maple St.
Joe's	Milller	2.75	Maple St.
Sue's	Bud	2.50	River Rd.
Sue's	Coors	3.00	River Rd.

- E.g. Consider the relations below and the question "How many tuples are in **Artists × Roles**?":



Artists:

aID	aName	nationality
1	Nicholson	American
2	Ford	American
3	Stone	British
4	Fisher	American

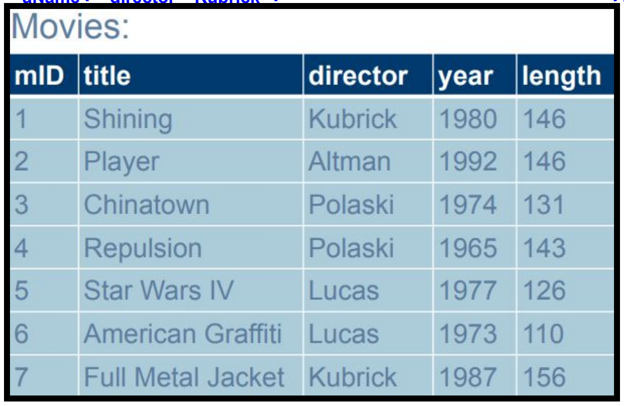
Roles:

mID	aID	character
1	1	Jack Torrance
3	1	Jake 'J.J.' Gittes
1	3	Delbert Grady
5	2	Han Solo
6	2	Bob Falfa
5	4	Princess Leia Organa

The answer is 24. There are 4 tuples in Artists and 6 in roles. 4*6 = 24.

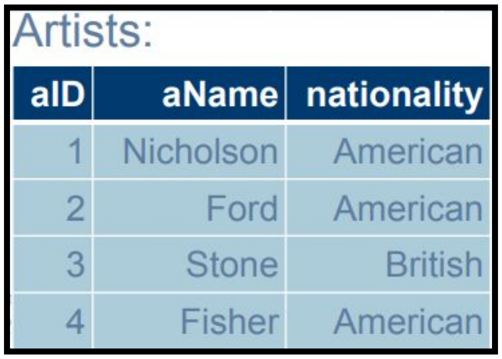
Now, with the same 2 relations from above, the answer to the question "How many tuples are in **Artists ⋈ Roles**?" is 6. There will be 2 rows with an aID of 1, 2 rows with an aID of 2, 1 row with an aID of 3 and 1 rot with an aID of 4.

- E.g. Consider the relations below. What is the result of:
  $\pi_{aName}(\sigma_{director="Kubrick"}(Artists \bowtie Roles \bowtie Movies))$?



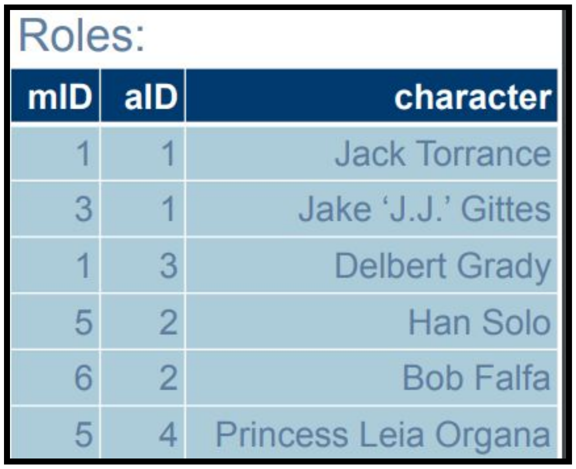
Movies:

mID	title	director	year	length
1	Shining	Kubrick	1980	146
2	Player	Altman	1992	146
3	Chinatown	Polaski	1974	131
4	Repulsion	Polaski	1965	143
5	Star Wars IV	Lucas	1977	126
6	American Graffiti	Lucas	1973	110
7	Full Metal Jacket	Kubrick	1987	156



Artists:

aID	aName	nationality
1	Nicholson	American
2	Ford	American
3	Stone	British
4	Fisher	American



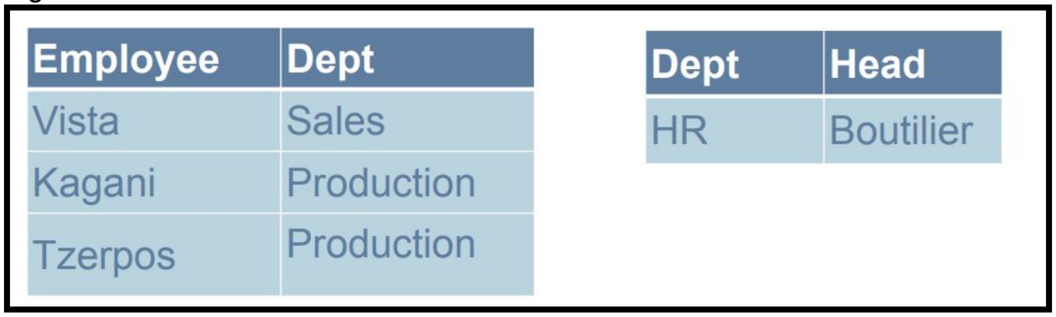
Roles:

mID	aID	character
1	1	Jack Torrance
3	1	Jake ‘J.J.’ Gittes
1	3	Delbert Grady
5	2	Han Solo
6	2	Bob Falfa
5	4	Princess Leia Organa

The answer is:

Nicholson
Stone

- Here are 3 special cases for natural join:
  1. **No tuples match:**
  - In this case, the result is a relation with no tuples.
  - E.g.

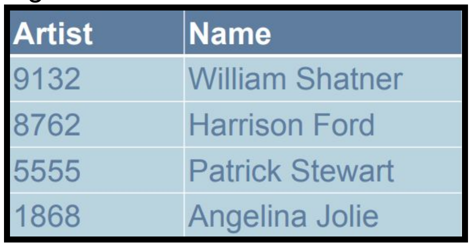


Employee	Dept
Vista	Sales
Kagani	Production
Tzerpos	Production

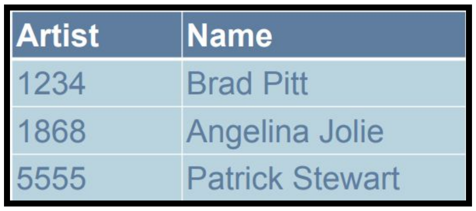
Dept	Head
HR	Boutilier

In this example, both relations have the Dept attribute, but no tuples match. Hence, the result of **Table_1 ⋈ Table_2** would be a table with no tuples.

2. **Exactly the same attributes:**
   - In this case, we're looking for tuples that are exactly the same in both tables.
   - E.g.



Artist	Name
9132	William Shatner
8762	Harrison Ford
5555	Patrick Stewart
1868	Angelina Jolie



Artist	Name
1234	Brad Pitt
1868	Angelina Jolie
5555	Patrick Stewart

Here, the result would be:

Artist	**Name**
1868	Angelina Jolie
5555	Patrick Stewart

This is because only the above 2 rows are in both tables.

3. **No attributes in common:**
   - As mentioned previously, the result of this would be the cartesian product of the two tables.

- Natural joins can over-match.
  Natural join bases the matching on attribute names, but what if two attributes have the same name, and we don't want them to have to match?
- Natural joins can also under-match.
  What if two attributes don't have the same name and we do want them to match?

Theta Join:

- It's common to use σ to check conditions after a Cartesian product.
  Theta Join makes this easier.
- Notation: $R \bowtie_{condition} S$
- The result is the same as the Cartesian product followed by select.
  In other words, $R \bowtie_{condition} S = \sigma_{condition}(R \times S)$.
- The word "theta" has no special connotation. It is an artifact of a definition in an early paper. You save just one symbol.
- You still have to write out the conditions, since they are not inferred.

Precedence:

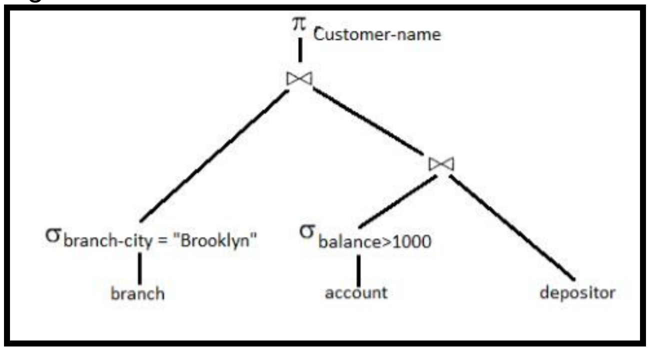
- Expressions can be composed recursively.
- It helps to annotate each subexpression, showing the attributes of its resulting relation.
- Parentheses and precedence rules define the order of evaluation.
- The precedence, from highest to lowest, is:

$$\begin{array}{l} \sigma,\ \pi,\ \rho \\ \times,\ \bowtie \\ \cap \\ \cup,\ - \end{array}$$

- Unless very sure, use brackets.

Breaking down expressions:

- Complex nested expressions can be hard to read.
- There are two alternative notations allow us to break them down:
    1. **Expression trees:**
    - Leaves are relations.
    - Interior notes are operators.
    - E.g.



    2. **Sequences of assignment statements:**
    - We can use assignment operators.
    - With assignment operators, we assign an expression to a relation.
    - Notation: **R := Expression**
    - Alternate notation: **R(A1, ..., An) := Expression**
      With this notation, we can rename the column names from the expression.
      I.e. This notation lets you name all the attributes of the new relation.
      Note: The number of columns from the expression must match the number of columns we put in the LHS.
    - **Note:** R must be a temporary variable, not one of the relations in the schema.
      I.e. You are not updating the content of a relation.
    - E.g.

$$\text{CSCoffering} := \sigma_{\text{dept='csc'}}\,\text{Offering}$$
$$\text{TookCSC(sid, grade)} := \pi_{\text{sid, grade}}(\text{CSCoffering} \bowtie \text{Took})$$
$$\text{PassedCSC(sid)} := \pi_{\text{sid}}\,\sigma_{\text{grade}>50}(\text{TookCSC})$$

    - Whether/how small to break things down is up to you. It's all for readability.
    - Assignment helps us break a problem down.
    - It also allows us to change the names of relations and attributes.

Rename:

- Notation: $\rho_{R1}(R2)$ or $\rho_{R1(A1, ..., An)}(R2)$
  The second way lets you rename all the attributes as well as the relation.
- Note that these are equivalent:
  R1(A1, ..., An) := R2
  R1 := $\rho_{R1(A1, ..., An)}(R2)$
- ρ is useful if you want to rename within an expression.

Union:

- Notation: **R U S**
- It includes all tuples that are in tables R or S. It also eliminates duplicate tuples.
- For a union operation to be valid, the following conditions must hold:
    - R and S must be the same number of attributes.

    - The attribute names of R has to match with the attribute names in S.
    - The attribute domains need to be compatible.
    - Duplicate tuples should be automatically removed.
- E.g. Consider the following relations and the query **A U B**:

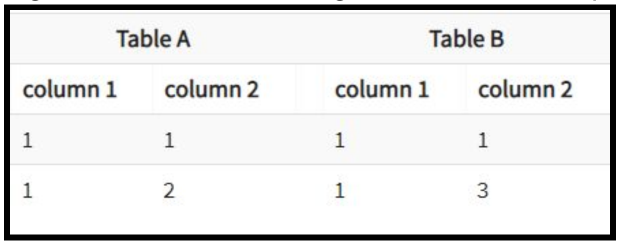


Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

The result is:

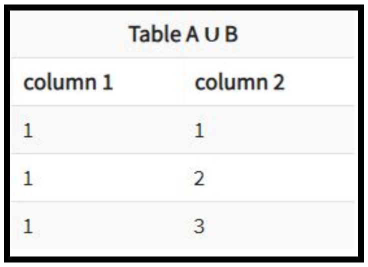


Table A U B	
column 1	column 2
1	1
1	2
1	3

Intersection:
- Notation: **R ∩ S**
- It includes all tuples that are in both tables R and S.
- For an intersection operation to be valid, the following conditions must hold:
    - The attribute names of R has to match with the attribute names in S.
    - R and S should be union compatible.
    - The result is a relation of all the tuples in both R and S.
- E.g. Consider the following relations and the query **A ∩ B**:

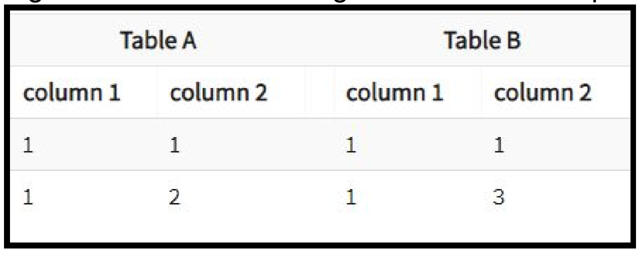


Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

The result is:

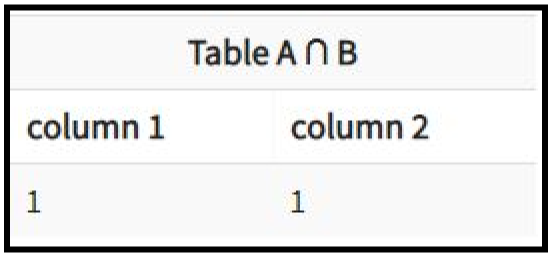


Table A ∩ B	
column 1	column 2
1	1

Difference:
- Notation: **R - S**
- It includes all tuples that are in table R but not in S.
- For a difference operation to be valid, the following conditions must hold:

  - The attribute names of R has to match with the attribute names in S.
  - R and S should be union compatible.
  - The result is a relation of all the tuples in R but not in S.
- E.g. Consider the following relations and the query **A - B**:

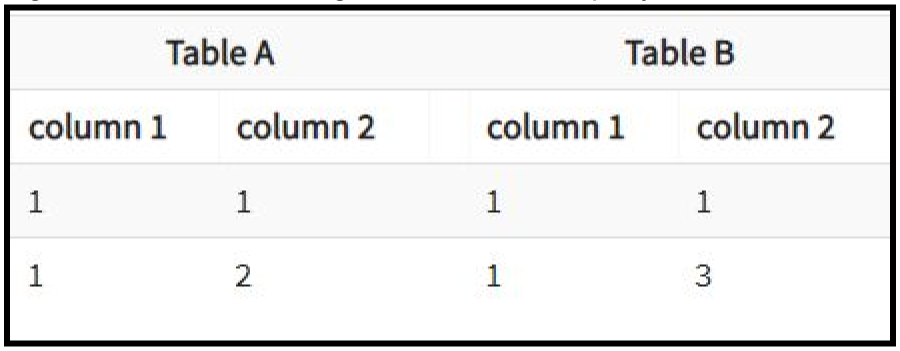


Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

The result is:

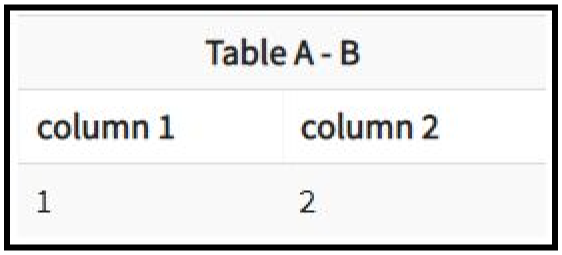


Table A - B	
column 1	column 2
1	2

Left Outer Join:
- Notation: $\mathbf{R \bowtie_L S}$
- The left outer join operation allows keeping all tuples in the left relation. However, if no matching tuple is found in the right relation, then the attributes of the right relation in the join result are filled with null values.
- E.g. Consider the relations below and the query $\mathbf{R \bowtie_L S}$:

R

Number	Square
1	1
3	9

S

Number	Cube
1	1
2	8

The result is:

Number	Square	Cube
1	1	1

3	9	-

Right Outer Join:

- Notation: $R⋈_R S$
- The right outer join operation allows keeping all tuples in the right relation. However, if no matching tuple is found in the left relation, then the attributes of the left relation in the join result are filled with null values.
- E.g. Consider the relations below and the query $R⋈_R S$:

R

Number	**Square**
1	1
3	9

S

Number	**Cube**
1	1
2	8

The result is:

Number	**Square**	**Cube**
1	1	1
2	-	8

Full Outer Join:

- Notation: $R⋈_O S$
- In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.
- E.g. Consider the relations below and the query $R⋈_O S$:

R

Number	**Square**
1	1
3	9

S

Number	**Cube**
1	1
2	8

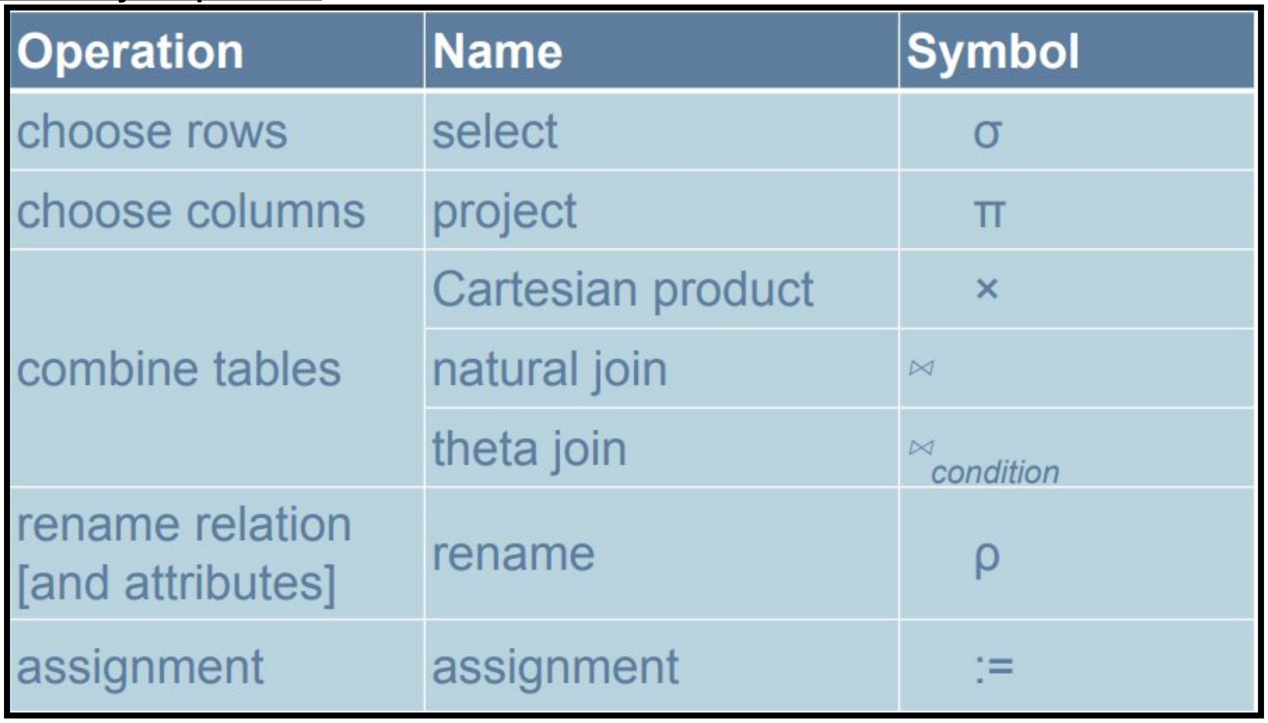
The result is:

Number	**Square**	**Cube**
1	1	1
2	-	8
3	9	-

Division:

- Notation: **R/S**
- It is used when we wish to express queries with "all" or "every".
- For integers, A/B is the largest int Q st Q x B ≤ A .
- For relations, A/B is the largest relation Q st Q × B ⊆ A.
- It is another "convenience" operator. But if you need it, it is a huge convenience. Defining a query without it is complicated.

Summary of operators:



Operation	Name	Symbol
choose rows	select	σ
choose columns	project	π
combine tables	Cartesian product	$\times$
	natural join	$\bowtie$
	theta join	$\bowtie_{condition}$
rename relation [and attributes]	rename	ρ
assignment	assignment	:=

Note: Some operations are not necessary. You can get the same effect using a combination of other operations. An example of this is theta join. We call this **syntactic sugar**.

Expressing Integrity Constraints:

- We've used this notation to express inclusion dependencies between relations R1 and R2: **R1[X] ⊆ R2[Y]**.
  Recall that the attributes in X must be a subset of the attributes in Y.
- We can use relational algebra to express other kinds of integrity constraints.
- Suppose R and S are expressions in relational algebra. We can write an integrity constraint in either of these ways:
  1. R = ∅
  2. R ⊆ S (equivalent to saying R - S = ∅)

Summary of techniques for writing queries in relational algebra:

- **Approaching the problem:**
    - Ask yourself which relations need to be involved. Ignore the rest.
    - Every time you combine relations, confirm that:
        1. Attributes that should match will be made to match and
        2. Attributes that will be made to match should match.
    - Annotate each subexpression, to show the attributes of its resulting relation.
- **Breaking down the problem:**
    - Remember that you must look one tuple at a time.
      If you need info from two different tuples, you must make a new relation where it's in one tuple.
    - Use the assignment operator to define intermediate relations.
        - Use good names for the new relations.
        - Name the attributes on the LHS each time, so you don't forget what you have in hand.
        - Add a comment explaining exactly what's in the relation.

Specific types of query:

- **Max (min is analogous):**
    - Pair tuples and find those that are not the max.
    - Then subtract from all to find the maxes.
- **"k or more":**
    - Make all combinations of k different tuples that satisfy the condition.
- **"exactly k":**
    - "k or more" - "(k+1) or more".
- **"every":**
    - Make all combos that should have occurred.
    - Subtract those that did occur to find those that didn't always. These are the failures.
    - Subtract the failures from all to get the answer.

Relational Algebra is procedural:

- A relational algebra query itself suggests a procedure for constructing the result.
  I.e. It describes how one could implement the query.
- We say that it is **procedural**.

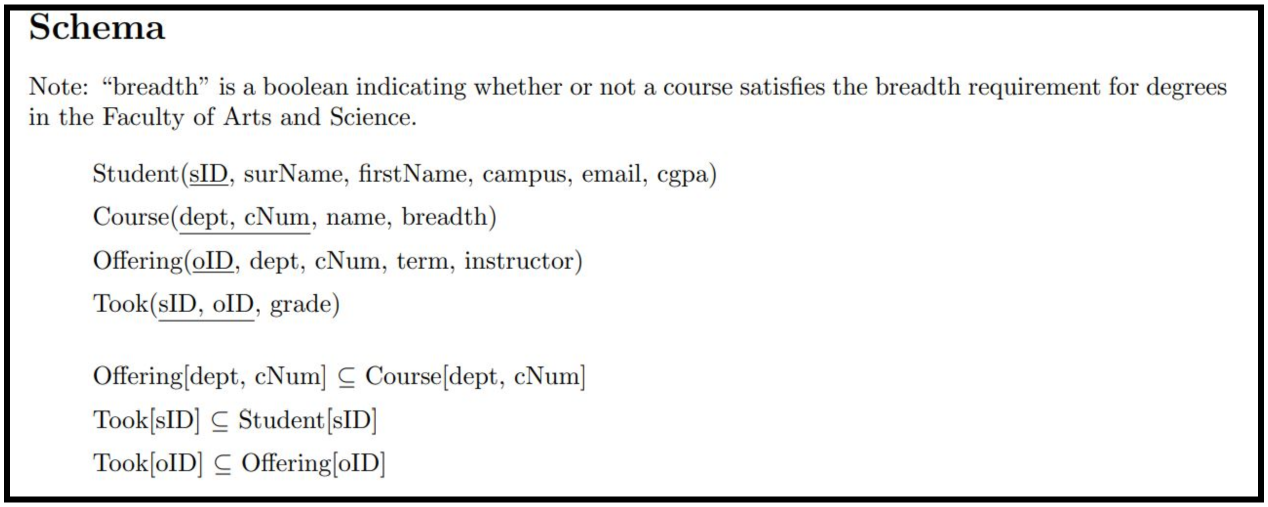
Evaluating queries:

- Any problem has multiple RA solutions.
    - Each solution suggests a "query execution plan".
    - Some may seem more efficient than others.
- In RA, we won't care about efficiency; it's an algebra.
- However, in a DBMS, where queries actually are executed, efficiency matters.
    - Which query execution plan is most efficient depends on the data in the database and what indices you have.
    - Fortunately, the DBMS optimizes our queries.
    - We can focus on what we want, not how to get it.
      I.e. Even if we write the queries in a very non-optimized way, the DBMS will optimize it for us.

Relational Calculus:

- Another abstract query language for the relational model.
- Based on first-order logic.
- RC is "declarative". The query describes what you want, but not how to get it.
- Queries look like this: **{t|t ε Movies ∧ t[director] = "Scott"}**

Examples:
Here is the schema:



Schema

Note: "breadth" is a boolean indicating whether or not a course satisfies the breadth requirement for degrees in the Faculty of Arts and Science.

Student(sID, surName, firstName, campus, email, cgpa)

Course(dept, cNum, name, breadth)

Offering(oID, dept, cNum, term, instructor)

Took(sID, oID, grade)

Offering[dept, cNum] $\subseteq$ Course[dept, cNum]

Took[sID] $\subseteq$ Student[sID]

Took[oID] $\subseteq$ Offering[oID]

- **Queries:**
  Write a query for each of the following:
  1. Student number of all students who have taken csc343.

     Soln:
     $\pi_{SID}(\sigma_{dept = \text{"CSC" and } cNum = 343}(Took \bowtie Offering))$
  2. Student number of all students who have taken csc343 and earned an A+ in it.

     Soln:
     Took_CSC343(SID) := $\pi_{SID}(\sigma_{dept = \text{"CSC" and } cNum = 343}(Took \bowtie Offering))$
     Got_A+(SID) := $\pi_{SID}(\sigma_{grade >= 90}(Took))$
     Took_CSC343_And_GotA+(SID) := TookCSC343(SID) $\cap$ GotA+(SID)
  3. The names of all such students.

     Soln:
     Took_CSC343(SID) := $\pi_{SID}(\sigma_{dept = \text{"CSC" and } cNum = 343}(Took \bowtie Offering))$
     Got_A+(SID) := $\pi_{SID}(\sigma_{grade >= 90}(Took))$
     Took_CSC343_And_GotA+(SID) := TookCSC343(SID) $\cap$ GotA+(SID)
     $\pi_{surName, firstName}$(Took_CSC343_And_GotA+ $\bowtie$ Students)
  4. The names of all students who have passed a breadth course with Professor Picky.

     Soln:
     Took_CSC343(SID) := $\pi_{SID}(\sigma_{dept = \text{"CSC" and } cNum = 343}(Took \bowtie Offering))$
     Got_A+(SID) := $\pi_{SID}(\sigma_{grade >= 90}(Took))$
     Took_CSC343_And_GotA+(SID) := TookCSC343(SID) $\cap$ GotA+(SID)
     $\pi_{surName, firstName}$(Took_CSC343_And_GotA+ $\bowtie$ Students)
  5. sID of all students who have earned some grade over 80 and some grade below 50.

     Soln:
     SID_Over_80(SID) := $\pi_{SID}(\sigma_{grade > 80}(Took))$
     SID_Under_80(SID) := $\pi_{SID}(\sigma_{grade < 50}(Took))$

SID_Over_80 ∩ SID_Under_80

6. Terms when Cook and Pitassi were both teaching something.

   Soln:

   Cook_Term(term) := $\pi_{term}(\sigma_{instructor = “Cook”}$(Offering))
   Pitassi_Term(term) := $\pi_{term}(\sigma_{instructor = “Pitassi”}$(Offering))
   Cook_Term ∩ Pitassi_Term

7. Terms when either of them was teaching csc463.

   Soln:

   Cook_Term(term) := $\pi_{term}(\sigma_{dept = “csc” \text{ and } cNum = 463 \text{ and } instructor = “Cook”}$(Offering))
   Pitassi_Term(term) := $\pi_{term}(\sigma_{dept = “csc” \text{ and } cNum = 463 \text{ and } instructor = “Pitassi”}$(Offering))
   Cook_Term U Pitassi_Term

8. sID of students who have earned a grade of 85 or more, or who have passed a course taught by Atwood.

   Soln:

   85_or_more(SID) := $\pi_{SID}(\sigma_{grade >= 85}$(Took))
   Passed_Atwood(SID) := $\pi_{SID}(\sigma_{grade >= 50 \text{ and } instructor = “Atwood”}$(Took ⋈ Offering)
   85_or_more U Passed_Atwood

9. Terms when csc369 was not offered.

   Soln:

   All_Term(Term) := π_{Term}(Offering)
   CSC369_Offered_Terms(Term) := $\pi_{Term}(\sigma_{dept=”CSC” \text{ and } cNum=369}$(Offering))
   All_Term - CSC369_Offered_Terms

10. Department and course number of courses that have never been offered.

    Soln:

    All_Courses(Dept, cNum) := $\pi_{dept, cNum}$(Course)
    Offered_Courses(Dept, cNum) := $\pi_{dept, cNum}$(Offering)
    All_Courses - Offered_Courses

11. SIDs and surnames of all pairs of students who've taken a course together.

    Soln:

    -- T1 and T2 are the same as Took.
    T1 := ρ_{T1}(Took)
    T2 := ρ_{T2}(Took)

    -- Gets pairs of sids s.t. they are taking the same class.
    -- **Note:** We use T1.sid < T2.sid and not T1.sid != T2.sid because we don't want
    -- duplicates. I.e. If student A and B are taking the class together, and we have
    -- A.sid and B.sid, we don't also want B.sid and A.sid.
    Pairs(sid1, sid2) := $\pi_{T1.sid, T2.sid}(\sigma_{T1.sid < T2.sid \text{ and } T1.oid = T2.oid}$(T1 x T2))

    -- Gets the surname of the first student.
    FirstName(sid1, sid2, name1) := $\pi_{sid1, sid2, surname}(\sigma_{sid1 = sid}$(Pairs x Student))

    -- Gets the surname of the first student.
    $\pi_{sid1, sid2, name1, surname}(\sigma_{sid2 = sid}$(FirstName x Student))

12. sID of student(s) with the highest grade in csc343, in term 20099.

    Soln:
    Takers(sid, grade) := $\pi_{sid,\ grade}$(Took $\bowtie_{dept=\text{“CSC”} \text{ and } cNum=343 \text{ and } term\ =\ 20099}$ Offering)
    NotTop(sid) := $\pi_{T1.sid}(\sigma_{T1.grade\ <\ T2.grade}(\rho_{T1}$Takers x ρ_{T2}Takers))
    Answer(sid) := π_{sid}(Takers) - NotTop
13. sID of students who have a grade of 100 at least twice.

    Soln:
    Answer(sid) := $\pi_{T1.sid}(\sigma_{T1.sid\ ==\ T2.sid \text{ and } T1.oid\ !=\ T2.oid \text{ and } T1.grade=100 \text{ and } T2.grade=100}(\rho_{T1}$Took x ρ_{T2}Took))
14. sID of students who have a grade of 100 exactly twice.

    Soln:
    At_Least_Twice(sid) := $\pi_{T1.sid}(\sigma_{T1.sid\ =\ T2.sid \text{ and } T1.oid\ !=\ T2.oid \text{ and } T1.grade=100 \text{ and } T2.grade=100}(\rho_{T1}$Took x ρ_{T2}Took))
    At_Least_Thrice(sid) := $\pi_{T1.sid}(\sigma_{T1.sid\ =\ T2.sid\ =\ T3.sid \text{ and } T1.oid\ !=\ T2.oid \text{ and } T1.oid\ !=\ T3.oid \text{ and } T2.oid\ !=\ T3.oid \text{ and } T1.grade=100 \text{ and } T2.grade=100 \text{ and } T3.grade\text{-}100}(\rho_{T1}$Took x ρ_{T2}Took x ρ_{T3}Took))
    Answer(sid) := At_Least_Twice - At_Least_Thrice

    Note: Since != isn’t transitive, we can’t do T1.oid != T2.oid != T3.oid.
15. sID of students who have a grade of 100 at most twice.

    Soln:
    Answer(sid) := π_{sid}(Student) - At_Least_Thrice
16. Department and cNum of all courses that have been taught in every term when csc448 was taught.

    Soln:
    448Terms(Term) := $\pi_{term}(\sigma_{dept=\text{“CSC”} \text{ and } cNum=448}$Offering)
    DidHappen(dept, cNum, term) := $\pi_{dept,\ cNum,\ term}$(Offering)
    ShouldHappen(dept, cNum, term) := $\pi_{dept,\ cNum}$(DidHappen) x 448Terms
    Didn’tHappen(dept, cNum, term) := ShouldHappen - DidHappen
    Answer(dept, cNum) := $\pi_{dept,\ cNum}$(DidHappen) - $\pi_{dept,\ cNum}$(Didn’tHappen)
17. Name of all students who have taken, at some point, every course Gries has taught (but not necessarily taken them from Gries)

    Soln:
    CoursesByGries(oid, dept, cNum) := $\pi_{oid,\ dept,\ cNum}(\sigma_{instructor=\text{“Gries”}}$Offering)
    ShouldHappen(sid, oid) := π_{sid}Student x π_{sid}CoursesByGries
    Didn’tHappen(sid) := π_{sid}(Student - $\pi_{sid,\ oid}$Took)
    Answer(surName, firstName) := $\pi_{surName,\ firstName}$(($\pi_{sid}$(Student) - Didn’tHappen) $\bowtie$ Student)

- **Integrity Constraints:**
  Express the following constraints using the notation R = ∅ or R - S = ∅:
  1. Courses at the 400-level cannot count for breadth.

     Soln:
     $\sigma_{\text{cNum >= 400 and cNum < 500 and Breath = True}}(\text{Course}) = \varnothing$
  2. CSC490 can only be offered at the same time as CSC454.

     Soln:
     $\text{490Terms(Term)} := \pi_{\text{Term}}(\sigma_{\text{dept = “CSC” and cNum = 490}}(\text{Offering}))$
     $\text{454Terms(Term)} := \pi_{\text{Term}}(\sigma_{\text{dept = “CSC” and cNum = 454}}(\text{Offering}))$
     $\text{490Terms - 454Terms} = \varnothing$