COMP353 Databases

Relational Algebra (RA)
for Relational Data Model

Relational Algebra (RA)

- *Database Query languages* are specialized languages to ask for information (**queries**) in DB.
- **Relational Algebra (RA)** is a query language associated with the relational data model.
- Queries in RA are expressions using a collection of operators on relations in the DB.
- The input(s) and output of a RA query are relations
- A query is ***evaluated*** using the ***current instance*** of the input relations to produce the output

Operations in "standard" RA

- The well-known set operations
  - √ Union ($\cup$)
  - √ Intersection ($\cap$)
  - √ Difference ($-$)
- Special DB operations that select "parts" of a relation instance
  - Selection (σ) – selects some rows (tuples) & discards the rest
  - Projection (π) – selects some columns (attributes) & discards the rest
- Operations that "combine" the tuples from the argument relations
  - √ Cartesian product ($\times$) – pairs the tuples in all possible ways
  - Join ($\bowtie$) – pairs particular tuples from the two input relations
- A unary operation to ***rename*** relations, called Rename (ρ)

Note: The output of a RA expression is an "unnamed" relation/set, i.e., **RA** expressions return **sets**, whereas **SQL** returns **multisets** (**bags**)

Compatibility Requirement

- We can apply the **set operators** of **union, intersection,** and **difference** to instances of relations **R** and **S** if **R** and **S** are *compatible*, that is they have "the same" schemas.
- **Definition**: Relations $S(A_1,\ldots,A_n)$ and $R(B_1,\ldots,B_m)$ are compatible if:
  (1) $n=m$ and
  (2) type(A_i) = type(B_i) (or compatible types), for all $1 \le i \le n$.

Set Operations on Relations

Let **R** and **S** be relation schemas, and **r** and **s** be any instances of them.

- The **union** of **r** and **s** is the set of all tuples that appear in either one or both. Each tuple **t** appears only once in the union, even if it appears in both; $r \cup s = \{t \mid t \in r \lor t \in s\}$
- The **intersection** of **r** and **s**, is the set of all tuples that appear in both; $r \cap s = \{t \mid t \in r \land t \in s\}$
- The **difference** of **r** and **s**, is the set of all tuples that appear in **r** but not in **s**; $r - s = \{t \mid t \in r \land t \notin s\}$
  - Commutative operations; **r Op s = s Op r**
    Note: Set difference (–) is not commutative, i.e., ($r - s \neq s - r$)

Example

Relation Schema: Star (<u>name</u>, address, gender, birthdate)

Instance r of Star:

Name	Address	Gender	Birthdate
Carrie Fisher	123 Maple	F	9/9/99
Mark Hamill	456 Oak rd.	M	8/8/88

Instance s of Star:

Name	Address	Gender	Birthdate
Carrie Fisher	123 Maple	F	9/9/99
Harrison Ford	789 Palm rd.	M	7/7/77

$r \cup s$:

Name	Address	Gender	Birthdate
Carrie Fisher	123 Maple	F	9/9/99
Mark Hamill	456 Oak rd.	M	8/8/88
Harrison Ford	789 Palm rd.	M	7/7/77

Example

Relation Schema: **Star** (name, address, gender, birthdate)

Instance r of Star:

Name	Address	Gender	Birthdate
Carrie Fisher	123 Maple	F	9/9/99
Mark Hamill	456 Oak rd.	M	8/8/88

Instance s of Star:

Name	Address	Gender	Birthdate
Carrie Fisher	123 Maple	F	9/9/99
Harrison Ford	789 Palm rd.	M	7/7/77

$r \cap s$:

Name	Address	Gender	Birthdate
Carrie Fisher	123 Maple	F	9/9/99

Example

Relation Schema: **Star** (name, address, gender, birthdate)

Instance r of Star:

Name	Address	Gender	Birthdate
Carrie Fisher	123 Maple	F	9/9/99
Mark Hamill	456 Oak rd.	M	8/8/88

Instance s of Star:

Name	Address	Gender	Birthdate
Carrie Fisher	123 Maple	F	9/9/99
Harrison Ford	789 Palm rd.	M	7/7/77

$r - s$:

Name	Address	Gender	Birthdate
Mark Hamill	456 Oak rd.	M	8/8/88

Example

Relation Schema: **Star** (name, address, gender, birthdate)

Instance r of Star:

Name	Address	Gender	Birthdate
Carrie Fisher	123 Maple	F	9/9/99
Mark Hamill	456 Oak rd.	M	8/8/88

Instance s of Star:

Name	Address	Gender	Birthdate
Carrie Fisher	123 Maple	F	9/9/99
Harrison Ford	789 Palm rd.	M	7/7/77

$s - r$:

Name	Address	Gender	Birthdate
Harrison Ford	789 Palm rd.	M	7/7/77

Projection (π)

- Let **R** be a relation schema.
- The **projection** operation (π) is used to produce, from any instance **r** of **R,** a new relation that includes listed "columns" of **R**
- The output of $\pi_{A1, A2,\ldots,Aj}$ (**r**) is a relation with columns $A_1, A_2,\ldots, A_j$, in this order.
- Note: The subscript of π is a *list*, which defines the structure of the output as the ordered tuple ($A_1, A_2,\ldots, A_j$).

Example

Relation Schema: **Movie**(title, year, length, filmType, studioName, producer)

Instance movie Of Movie:

title	year	length	filmType	studioName	producer
Star wars	1977	124	color	Fox	12345
Mighty Ducks	1991	104	color	Disney	67890
Wayne's World	1992	95	color	Paramount	99999

Query: $\pi_{\text{title, year, length}}$(movie)

title	year	length
Star wars	1977	124
Mighty Ducks	1991	104
Wayne's World	1992	95

Example

Relation Schema: **Movie**(title, year, length, filmType, studioName, producer)

Instance movie Of Movie:

title	year	length	filmType	studioName	producer
Star wars	1977	124	color	Fox	12345
Mighty Ducks	1991	104	color	Disney	67890
Wayne's World	1992	95	color	Paramount	99999

Query: π_{filmType}(movie)

Result:

filmType
color

Selection (σ)

- The **selection** operator (σ), applied to an instance **r** of relation **R**, returns a subset of **r**
- We denote this operation/query by $\sigma_C(r)$
- The output includes tuples satisfying condition **C**
- The schema of the output is the same as **R**

Example

Relation Schema: **Movie**(<u>title</u>, <u>year</u>, length, filmType, studioName, producer)

Instance movie of Movie:

title	year	length	filmType	studioName	producer
Star wars	1977	124	color	Fox	12345
Mighty Ducks	1991	104	color	Disney	67890
Wayne's World	1992	95	color	Paramount	99999

Query: $\sigma_{length \geq 100}(movie)$

Result:

title	year	length	filmType	studioName	producer
Star wars	1977	124	color	Fox	12345
Mighty Ducks	1991	104	color	Disney	67890

Example

Relation: **Movie**(<u>title</u>, <u>year</u>, length, filmType, studioName, producer)

Instance movie of Movie:

title	year	length	filmType	studioName	producer
Star wars	1977	124	color	Fox	12345
Mighty Ducks	1991	104	color	Disney	67890
Wayne's World	1992	95	color	Paramount	99999

Query: $\sigma_{length \geq 100 \text{ AND } studioName = \text{'Fox'}}(movie)$

Result:

title	year	length	filmType	studioName	producer
Star wars	1977	124	color	Fox	12345

Cartesian Product ($\times$)

- Let **R** and **S** be relation schemas, and **r** and **s** be any instances of **R** and **S**, respectively.
- The **Cartesian Product** of **r** and **s** is the set of all tuples obtained by "concatenating" the tuples in **r** and **s**. Formally, $r \times s = \{ t_1.t_2 \mid t_1 \in r \wedge t_2 \in s \}$
- The schema of result is the "union" of **R** and **S**
  - If **R** and **S** have some attributes in common, we need to invent new names for identical names, e.g., use **R.B** and **S.B**, if **B** appears in both **R** and **S**

Example

Instance r of R:

A	B
1	2
3	4

Instance s of S:

B	C	D
2	5	6
4	7	8
9	10	11

r $\times$ s:

A	R.B	S.B	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

Theta-join (θ)

- Suppose **R** and **S** are relation schemas, **r** is an instance of **R**, and **s** is an instance of **S**. The **theta-join** of **r** and **s** is the set of all tuples obtained from concatenating all $t_1 \in$ **r** and $t_2 \in$ **s**, such that t_1 and t_2 *satisfy some condition* **C**
- We denote **θ**-join by $r \bowtie_C s$
- The schema of the result is the same as the schema of **R** $\times$ **S** (i.e., the union of **R** and **S**)
- **C** is a Boolean expression, simple or complex, as in operation σ

Example

Instance r of R:

A	B	C
1	2	3
6	5	8
9	7	11

Instance s of S:

B	C	D
2	3	4
2	3	5
7	8	10

$r \bowtie_{A<D} s$:

A	R.B	R.C	S.B	S.C	D
1	2	3	2	3	4
1	2	3	2	3	5
1	2	3	7	8	10
6	5	8	7	8	10
9	7	11	7	8	10

Equi-join

- The **equi-join** operator, is a special case of θ-join, in which we may only use the equality relation (=) in condition **C**
- It is denoted as $\mathbf{r} \bowtie_{\mathbf{c}} \mathbf{s}$ (i.e., the same as θ-join)
- The schema of the output is the same as that of θ-join

Example

Instance r of R:

A	B	C
1	2	3
6	5	8
9	7	11

Instance s of S:

B	C	D
2	3	4
2	3	5
7	8	10

$r \bowtie_{R.C=S.C} s$:

A	R.B	R.C	S.B	S.C	D
1	2	3	2	3	4
1	2	3	2	3	5
6	5	8	7	8	10

Natural Join ($\bowtie$)

- **Natural join**, is a special case of equi-join, where the equalities are not explicitly specified, rather they are assumed implicitly on the common attributes of **R** and **S**
- We denote this natural join operation by $\mathbf{r} \bowtie \mathbf{s}$
- The schema of the output is similar to that of equi-join, except that each common attribute appears only once.

Note: If **R** and **S** do not have any common attribute, then the join operation becomes Cartesian product.

Example

Instance r of R:

A	B	C
1	2	3
6	2	8
9	7	3

Instance s of S:

B	C	D
2	3	4
2	3	5
7	8	10

$r \bowtie s$:

A	B	C	D
1	2	3	4
1	2	3	5

Expressing Queries in RA

- Every standard RA operation has relation(s) as argument(s) and produces a relation (set) as the output

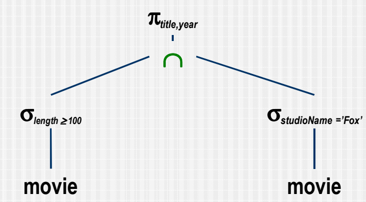
  (Exception is the sort operator τ)
- This property of RA operations (that inputs and output are relations) make it possible to formulate/express any query by composing/nesting/grouping queries.
- We can use parentheses for *grouping, in order* to improve clarity and readability

Example: RA Query

- Relation schema:
  Movie (<u>title</u>, <u>year</u>, length, filmType, studioName)
- Query: List the *title* and *year* of every movie made by **Fox** studio whose length is at least 100 minutes?
- One way to express this query in RA is:
  $\pi_{title,year}(\sigma_{studioName = 'Fox' \text{ and } length >= 100}(\textbf{movie}))$
- Another way:
  - Select those **movie** tuples that have *length* ≥100
  - Select those **movie** tuples that have *studioName* = 'Fox'
  - Find the intersection of the above two results
  - Project on the attributes ***title*** and ***year***

Example: RA Query

$\pi_{title,year}(\sigma_{studioName = 'Fox' \text{ and } length >= 100}(\textbf{movie}))$



$\pi_{title,year}(\sigma_{length \geq 100}(\textbf{movie}) \cap \sigma_{studioName = 'Fox'}(\textbf{movie}))$

Example: RA Query

- Relation schema:
  Movie (<u>title</u>, <u>year</u>, length, filmType, studioName)
  StarsIn (<u>title</u>, <u>year</u>, <u>starName</u>)
- Query: List the **names** of the stars of movies of length ≥ 100 minutes long.
- One expression in RA for this query:
  - Select **movie** tuples of ***length* ≥ 100**
  - Join the result with relation **StarsIn**
  - Project on the attribute ***starName***
- ***Exp1***: $\pi_{starName}(\sigma_{length \geq 100}(\textbf{movie}) \bowtie \textbf{starsIn})$
- ***Another solution***: $\pi_{starName}(\sigma_{length \geq 100}(\textbf{movie} \bowtie \textbf{starsIn}))$

Renaming Operator (ρ)

- To **control** the names of the attributes of the relations used by algebra operations, we may need to explicitly **rename** relations. Can do this for *convenience* as well
- **Renaming Operator** is denoted as $\rho_{s(A1,A2,\ldots,An)}(\mathbf{r})$
- The result is a copy of the input relation instance **r**, but named as **s,** and the attributes A1, A2, ..., An, in that order.
- Use $\rho_s(\mathbf{r})$ to give relation **r**, a new name **s**
  In this case, the schema of **s** will be the same as that of **r**.

Example

- Query: $\pi_{starName}(\sigma_{length \geq 100}(\textbf{movie}) \bowtie \textbf{starsIn})$
- This query can be rewritten in 2 steps as follows:
  1. $\rho_{M(\text{title, year, length, filmType, studioName})}(\sigma_{length \geq 100}(\textbf{movie}))$
     or even simpler as: $\rho_M(\sigma_{length \geq 100}(\textbf{movie}))$ if used in the same formula
  2. Or use M := $\sigma_{length \geq 100}$(movie) as a separate formula and then formulate the query as: $\pi_{starName}(\textbf{M} \bowtie \textbf{starsIn})$
- Consider **takes**(<u>sid</u>, <u>cid</u>, grade)
- Query: Find ID of every student who has taken at least 2 courses.
- $\pi_{takes.sid}(\sigma_{(takes.sid = T.sid) \text{ and } (takes.cid \neq T.cid)}(takes \times \rho_T(takes)))$

Dependent and Independent Operations

- Some RA operations can be expressed based on other operations. Examples include:
  - $\mathbf{r} \cap \mathbf{s} = \mathbf{r} - (\mathbf{r} - \mathbf{s})$
  - $\mathbf{r} \bowtie_C \mathbf{s} = \sigma_C(\mathbf{r} \times \mathbf{s})$
  - $\mathbf{r} \bowtie \mathbf{s} = \pi_L(\sigma_{r.A1 = s.A1 \text{ AND} \ldots \text{ AND } r.An = s.An}(\mathbf{r} \times \mathbf{s}))$,
    where **L** is the list of attributes in **R** *followed* by those attributes in **S** that are not in **R,** and **A1**,..., **An** are the common attributes of **R** and **S**

Relational Algebra with Bag Semantics

- Relations stored in DB are called **base** relations/tables.
- Base relations are normally sets; no duplicates.
- In some situations, e.g., during query processing, it is allowed for relations to have duplicate tuples.
- If duplicates are allowed in a collection, it is called ***bag/multiset***.

Instance r of R:

A	B	C
1	2	3
6	5	8
6	5	8
1	2	3
9	7	11

Here, r is a bag

Why Bags?

- **1. Faster projection operations**
  - Bag projection is faster, since otherwise returning distinct values is expensive (as we need sorting for duplicate elimination.

    Another example**:** *Computing the bag union* (**r** $\cup^B$ **s**) *is much cheaper* than computing the standard set union **r** $\cup$ **s**.

    Formally, if **r** and **s** have *n* and m tuples, then the bag and set union operations will cost $O(n+m)$ and $O(n*m)$, respectively.
- **2. Correct computation** with some aggregation
  - For example, to compute the *average* of values for attribute A in the previous relation, we must consider the bag of those values

Set Operations on Bags

- **r** $\cup^B$ **s**, the **bag union** of **r** and **s**, is the bag of tuples that are in **r**, in **s**, or in both. If a tuple *t* appears *n* times in **r**, and *m* times in **s**, then *t* appears *n+m* times in bag **r** $\cup^B$ **s**

  $$r \cup^B s = \{ t{:}k \mid t{:}n \in r \wedge t{:}m \in s \wedge k = n+m \}$$
- **r** $\cap^B$ **s**, the **bag intersection** of **r** and **s**, is the bag of tuples that appear in both **r** and **s.** If a tuple *t* appears *n* times in **r**, and *m* times in **s**, then the number of occurrences of t in bag **r** $\cap^B$ **s** is *min(n,m)*

  $$r \cap^B s = \{ t{:}k \mid t{:}n \in r \wedge t{:}m \in s \wedge k = min(n,m) \}$$
- **r** $-^B$ **s**, the **bag difference** of **r** and **s** is defined as follows:

  $$r -^B s = \{ t{:}k \mid t{:}n \in r \wedge t{:}m \in s \wedge k = max(0, n-m) \}$$
  $$s -^B r = \{ t{:}k \mid t{:}n \in r \wedge t{:}m \in s \wedge k = max(0, m-n) \}$$

Example

Bag r:

A	B
1	2
3	4
1	2
1	2

Bag s:

A	B
1	2
3	4
3	4
5	6

r $\cup^B$ s:

A	B
1	2
3	4
1	2
1	2
1	2
3	4
3	4
5	6

Example

Bag r:

A	B
1	2
3	4
1	2
1	2

Bag s:

A	B
1	2
3	4
3	4
5	6

r $\cap^B$ s:

A	B
1	2
3	4

Example

Bag r:

A	B
1	2
3	4
1	2
1	2

Bag s:

A	B
1	2
3	4
3	4
5	6

r $-^B$ s:

A	B
1	2
1	2

Example

Bag r:

A	B
1	2
3	4
1	2
1	2

Bag s:

A	B
1	2
3	4
3	4
5	6

$s -^{B} r$:

A	B
3	4
5	6

Bag Projection π^B

- Let **R** be a relation scheme, and **r** be a collection of tuples over **R**, which could have duplicates.
  The **bag projection** operator is used to produce, from **r,** a bag of tuples over some of **R.**
- Even when **r** does not have duplicates, we may get duplicates when projecting on some attributes of **R**.
  That is, π^B does not eliminate the duplicates and hence corresponds exactly to the SELECT clause in SQL.

Example

Bag r:

A	B	C
1	2	5
3	4	6
1	2	7
1	2	8

$\pi^B_{A, B}$ (r):

A	B
1	2
3	4
1	2
1	2

Example

Relation Schema: movie(title, year, length, filmType, studioName, producer)

Instance:

title	year	length	filmType	studioName	producer
Star wars	1977	124	color	Fox	12345
Mighty Ducks	1991	104	color	Disney	67890
Wayne's World	1992	95	color	Paramount	99999

$\pi^B_{filmType}$(movie):

filmType
color
color
color

Selection on Bags

- The selection operator σ_C applied to an instance **r** of relation **R**, will return a subset of **r**
  - The tuples in the output relation are those that satisfy condition **C**, which involves attributes of **R**
  - Duplicates are **not** eliminated from the result of a bag-selection

    Note: The selection operation σ in RA is different from the **SELECT** clause in SQL

Example

Bag r:

A	B	C
1	2	5
3	4	6
1	2	7
1	2	7

$\sigma_{C \geq 6}$(r):

A	B	C
3	4	6
1	2	7
1	2	7

Cartesian Product of Bags

- The **Cartesian Product** of bags **r** and **s** is the bag of tuples that can be formed by concatenating pairs of tuples, the first of which comes from **r** and the second from **s**. In symbols, $\mathbf{r} \times \mathbf{s} = \{ t_1.t_2 \mid t_1 \in \mathbf{r} \wedge t_2 \in \mathbf{s} \}$
  - Each tuple of one relation is paired with each tuple of the other, regardless of whether it is a duplicate or not
  - If a tuple t_1 appears ***m*** times in a relation **r**, and a tuple t_2 appears ***n*** times in relation **s**, then tuple $t_1.t_2$ appears **m*n** times in their bag-product, **r × s**

Example

Bag r:

A	B
1	2
1	2

Bag s:

B	C
2	3
4	5
4	5

r × s:

A	R.B	S.B	C
1	2	2	3
1	2	2	3
1	2	4	5
1	2	4	5
1	2	4	5
1	2	4	5

Join of Bags

- The bag join is computed in the same way as the standard join operation
- Duplicates are not eliminated in a bag join operation

Bag r:

A	B
1	2
1	2

Bag s:

B	C
2	3
4	5

r ⋈ s:

A	B	C
1	2	3
1	2	3

Constraints on Relations

- RA offers a convenient way to express a wide variety of constraints, e.g., referential integrity and FD's.
- There are **two ways** to express constraints in RA
  1. If **r** is an expression in RA, then the constraint **r** = ∅ says:
     *"**r** has no tuples, i.e., or **r** is empty"*
  2. If **r** and **s** are RA *expressions*, then the constraint **r** ⊆ **s** says:
     "*every tuple in (the result of)* **r** *is in (the result of)* **s**"

These constraints hold also when **r** and **s** are bags.

Constraints on Relations

- The two types of constraints are not independent. For instance:
  - The constraint **r** ⊆ **s** could be written as **r – s** = ∅

This follows from the definition of "–", because **r** ⊆ **s iff** **r – s** = ∅, meaning that there is no tuple in **r** that is not in **s**

Referential Integrity Constraints

- Referential integrity in relational data model means:
  - if we have a value ***v*** in a tuple *t* in a relation **r**, then we also expect that ***v*** appears in a particular component of some tuple *s* in relation **s**

  E.g., if we have a tuple *(s,c,g)* in ***takes****(sid,cid,grade)*, then there must be a **student** with *sid = s* and a **course** with *cid = c* such that *s* has taken *c*

  *The mentions of values s and c in **takes** "refers" to some values outside this relation, and these values **must** exist*

Example

- Relation schemas:
  Movie (<u>title</u>, <u>year</u>, length, filmType)
  StarsIn (<u>title</u>, <u>year</u>, <u>starName</u>)
- Constraint:
  the **title** and **year** of every movie that appears in relation **starsIn** *must* appear also in **movie**; otherwise there is a violation in referencing in **starsIn**
- Query in RA:
  - $\pi_{\text{title, year}}(\textbf{starsIn}) \subseteq \pi_{\text{title, year}}(\textbf{movie})$

  or equivalently
  - $\pi_{\text{title, year}}(\textbf{starsIn}) - \pi_{\text{title, year}}(\textbf{movie}) = \varnothing$

Functional Dependencies

- **Any** functional dependency $X \longrightarrow Y$ can be expressed as an expression in RA
- Example:
  Consider the relation schema:
  Star (<u>name</u>, address, gender, birthdate)
- How to express the FD: **name → address** in RA?

Functional Dependencies

- Relation schema:
  Star (<u>name</u>, address, birthdate)
- With the FD: **name → address**
- The **idea** is that if we construct all pairs of **star** tuples, we **must not** find a pair that agree on **name** but disagree on **address**
- To "construct" the pairs in RA, we use **Cartesian product**, and to find pairs that violate this FD, we use **selection**
- We are then ready to express this **FD** by equating the result to **ø**, as follows…

Example

Star:

Name	Address	Birthdate
Carrie Fisher	123 Maple	9/9/99
Mark Hamill	456 Oak rd.	8/8/88
Harrison Ford	789 Palm rd.	7/7/77

$\rho_{S1(\text{name, address, birthdate})}(\textbf{star})$

Name	Address	Birthdate
Carrie Fisher	123 Maple	9/9/99
Mark Hamill	456 Oak rd.	8/8/88
Harrison Ford	789 Palm rd.	7/7/77

$\rho_{S2(\text{name, address,birthdate})}(\textbf{star})$

Name	Address	Birthdate
Carrie Fisher	123 Maple	9/9/99
Mark Hamill	456 Oak rd.	8/8/88
Harrison Ford	789 Palm rd.	7/7/77

Example

s1 × s2:

S1.Name	S1.Address	S1.Birthdate	S2.Name	S2.Address	S2.Birthda
Carrie Fisher	123 Maple	9/9/99	Carrie Fisher	123 Maple	9/9/99
Carrie Fisher	123 Maple	9/9/99	Mark Hamill	456 Oak rd.	8/8/88
Carrie Fisher	123 Maple	9/9/99	Harrison Ford	789 Palm rd.	7/7/77
Mark Hamill	456 Oak rd.	8/8/88	Carrie Fisher	123 Maple	9/9/99
Mark Hamill	456 Oak rd.	8/8/88	Mark Hamill	456 Oak rd.	8/8/88
Mark Hamill	456 Oak rd.	8/8/88	Harrison Ford	789 Palm rd.	7/7/77
Harrison Ford	789 Palm rd.	7/7/77	Carrie Fisher	123 Maple	9/9/99
Harrison Ford	789 Palm rd.	7/7/77	Mark Hamill	456 Oak rd.	8/8/88
Harrison Ford	789 Palm rd.	7/7/77	Harrison Ford	789 Palm rd.	7/7/77

$\sigma_{\text{S1.name=S2.name AND S1.address} \neq \text{S2.address}}(\text{s1} \times \text{s2}) = \varnothing$

Functional Dependencies

- Relation schema:
  Star (<u>name</u>, address, birthdate)
- With the FD: **name → address**
- In RA:

$\sigma_{\text{S1.name=S2.name AND S1.address} \neq \text{S2.address}}(\rho_{S1}(\textbf{star}) \times \rho_{S2}(\textbf{star})) = \varnothing$

Domain Constraints

- Relation schema:
  Star (<u>name</u>, address, gender, birthdate)
- How to express the following constraint?
  Valid values for **gender** are **'F'** and **'M'**
- In RA:
  - $\sigma_{gender \neq 'F' \text{ AND } gender \neq 'M'}$ **(star)** = $\varnothing$
  - This is an example of ***domain constraints***

Domain Constraints

- Relation schema:
  Employee (<u>eid</u>**,** name, address, salary)
- How to express the constraint:
  Maximum employee salaries is $150,000
- In RA:
  - $\sigma_{salary > 150000}$ **(employee)** = $\varnothing$

"For All" Queries (1)

- Given the database schema:
  Student(Sid, Sname, Addr)
  Course(Cid, Cname, Credits)
  Enrolled (<u>Sid</u>, <u>Cid</u>)
- Consider the query:
  "*Find students enrolled in **all** the courses.*"
- A first attempt (below) fails!
  π_{Sid} (Enrolled)
- This RA query returns students enrolled in some courses.
- So, how to correctly express "For All" types of queries?

"For All" Queries (2)

- A *solution strategy* would be to:
  - start with the list of all students (**all guys**), from which we then subtract those who have not taken some courses (**bad guys**)
- That is, to find all the "good guys", we need to find "all guys" from which we then remove the "bad guys", i.e.,

  Answer (**Good guys**) = **All guys** – **Bad guys**

"For All" Queries (3)

- Set of all students which we need to consider:

  All Courses $\Leftarrow \pi_{Cid}$ **(Course)**
  All Students $\Leftarrow \pi_{Sid}$ **(Student)**
- Steps to find students not enrolled in all the courses
  1. Create all possible "student-course" pairs:
     SC-Pairs $\leftarrow \pi_{Sid}$ **(Student)** $\times\ \pi_{Cid}$ **(Course)**
  2. Get all "actual" student-course pairs -- take them from **Enrolled**
  3. Students who are not enrolled in all the courses:
     B $\leftarrow \pi_{Sid}(\pi_{Sid}$**(Student)** $\times\ \pi_{Cid}$**(Course)** – **Enrolled**)
- **Answer** : **All Students** – **"Bad"**

The Division Operation (÷)

- The previous query can be conveniently and expressed in RA using the ***division*** operator ÷
  - Divide Enrolled by π_{Cid} (Course)
    that is, Enrolled ÷ π_{Cid} (Course)
  - Schema of the result is {Sid, Cid} – {Cid}
- R ÷ S requires that the attributes of S to be a subset of R.
  - The schema of the output would be R – S

Example: Enrolled (student, sport)

Find students enrolled in all sports {Hockey, Football}.

Enrolled (Student, sport)

Jim	Hockey
Joe	Football
Jim	Football
Sue	Hockey

Example: Enrolled(student, sport)

$\pi_{student}$(Enrolled) $\times$ π_{sport}(Enrolled) – Enrolled =

Jim	Hockey
Jim	Football
Joe	Hockey
Joe	Football
Sue	Hockey
Sue	Football

–

Jim	Hockey
Joe	Football
Jim	Football
Sue	Hockey

=

Joe	Hockey
Sue	Football

$\pi_{student}$(Enrolled) – $\pi_{student}$($\pi_{student}$(Enrolled) $\times$ π_{sport}(Enrolled) – Enrolled)

All

Jim
Joe
Sue

–

Bad

Joe
Sue

= ? ➔ Jim is the only student enrolled in all sports

Another Example

- r ÷ s = π_{R-S}(r – (π_{R-S}(r)×s - r))
- Given the DB schema:
  - Customer(cid, name)
  - Branch(bid, district)
  - Account(cid, bid)
- Query: "*Find the names of all customers who have an account in every branch located in the Westmount area*"
- Solution?
  - π_{name} (Customer $\bowtie$ Account $\bowtie$ ($\sigma_{district = \text{"Westmount"}}$(Branch)))?
- No, this returns the names of all customers who have an account at some branch in Westmount, but not necessarily at every such branch.

Database:
Customer(cid, name), Branch(bid, district), Account(cid, bid)

- We can apply the division operator ÷
  - Find all customer-branch pairs **(cid, bid)** for which customer (cid) has an account at branch (bid):
    $\pi_{cid,bid}$ (customer $\bowtie$ account)
  - Divide the above by all bid's of branches in Westmount
    π_{bid} ($\sigma_{district = \text{"Westmount"}}$(branch))
- π_{name} ((customer $\bowtie$ account) ÷ π_{bid} ($\sigma_{district = \text{"Westmount"}}$(branch)))
- So, the division operation r ÷ s is defined as:
  r ÷ s = π_{R-S}(r – (π_{R-S}(r)×s - r))