

### The Relational Model

- Structure of Relational Databases
- Relational Algebra

Reading:

- => Chapter 2
- => Chapter 6, sections 1 & 2 (3 is optional).



- Formally, given sets  $D_1$ ,  $D_2$ , ...,  $D_n$  a <u>relation</u> r is a subset of  $D_1 \times D_2 \times \ldots \times D_n$
- Thus, a relation is a <u>set</u> of <u>tuples</u>  $(a_1, a_2, ..., a_n)$  where each  $a_i \in D_i$
- Example:

cust-name	= {Jones, Smith, Curry, Lindsay}
cust-street	= {Main, North, Park}
cust-city	= {Harrison, Rye, Pittsfield}

r = {(Jones, Main, Harrison), (Smith, North, Rye), (Curry, North, Rye), (Lindsay, Park, Pittsfield)}



- Since a relation is a set, the order of tuples is irrelevant and may be thought of as arbitrary.
- In a real DBMS, tuple order is typically very important and not arbitrary.
- Historically, this was/is a point of contention for the theorists.



# Table vs. Relation

- In a DBMS, a relation is represented or stored as a table.
- The Relation:

{ (A-101,Downtown,500), (A-102,Perryridge,400), (A-201,Brighton,900), : (A-305,Round Hill,350) }

The Table:

account-number	branch-name	balance
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350



- Each attribute of a relation has a name.
- The set of allowed values for each attribute is called the <u>domain</u> of the attribute.
- Attribute values are required to be *atomic*, that is, indivisible.
- This will differ from ER modeling, which will have:
  - Multi-valued attributes
  - Composite attributes



### The Evil Value "Null"

- The special value *null* is an implicit member of every domain.
- Thus, tuples can have a *null* value for some of their attributes.
- A null value can be interpreted in several ways:
  - value is unknown
  - value does not exist
  - > value is known and exists, but just hasn't been entered yet
- The null value causes complications in the definition of many operations.
- We shall consider their effect later.



Let  $A_1, A_2, ..., A_n$  be attributes. Then  $R = (A_1, A_2, ..., A_n)$  is a <u>relation schema</u>.

*Customer-schema* = (*customer-name*, *customer-street*, *customer-city*)

Sometimes referred to as a *relational schema* or *relational scheme*.



### Database

#### A database consists of multiple relations: (example)

account	-	account information
depositor	-	depositor information, i.e., who deposits into which accounts
customer	-	customer information

Storing all information as a single relation is possible:

bank(account-number, balance, customer-name, ..)

- This results in:
  - Repetition of information (e.g. two customers own an account)
  - The need for null values (e.g. represent a customer without an account).



#### Banking enterprise: (keys underlined)

customer (<u>customer-name</u>, customer-street, customer-city) branch (<u>branch-name</u>, branch-city, assets) account (<u>account-number</u>, branch-name, balance) loan (<u>loan-number</u>, branch-name, amount) depositor (<u>customer-name</u>, <u>account-number</u>) borrower (<u>customer-name</u>, <u>loan-number</u>)



### **Relational Schemes**

#### University enterprise:

classroom (<u>building</u>, <u>room-number</u>, capacity) department (<u>dept-name</u>, building, budget) course (<u>course-id</u>, title, dept-name, credits) instructor (<u>ID</u>, name, depart-name, salary) section (<u>course-id</u>, <u>sec-id</u>, <u>semester</u>, <u>year</u>, building, room-number, time-slot-id) teaches (<u>ID</u>, <u>course-id</u>, <u>sec-id</u>, <u>semester</u>, <u>year</u>) student (<u>ID</u>, name, dept-name, tot-cred) takes (<u>ID</u>, <u>course-id</u>, <u>sec-id</u>, <u>semester</u>, <u>year</u>, grade) advisor (<u>s-ID</u>, <u>i-ID</u>) time-slot (<u>time-slot-id</u>, <u>day</u>, <u>start-time</u>, end-time) prereq (<u>course-id</u>, <u>prereq-id</u>)



### **Relational Schemes**

#### Employee enterprise:

employee(<u>person-name</u>, street, city) works(<u>person-name</u>, company-name, salary) company(<u>company-name</u>, city) manages(<u>person-name</u>, manager-name)



Language in which user requests information from the database.

- Recall there are two categories of languages
  - procedural
  - > non-procedural

#### "Pure" languages:

- Relational Algebra (procedural, according to the current version of the book)
- Tuple Relational Calculus (non-procedural)
- Domain Relational Calculus (non-procedural)
- Pure languages form underlying basis of "real" query languages.



### Relational Algebra

- Procedural language (according to the book), at least in terms of style.
- Six basic operators:
  - > select
  - > project
  - > union
  - set difference
  - cartesian product
  - rename



- Each operator takes one or more relations as input and results in a new relation.
- Each operation defines:
  - Requirements or constraints on its' parameters.
  - > Attributes in the resulting relation, including their types and names.
  - > Which tuples will be included in the result.

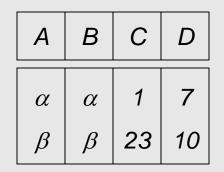


# Select Operation – Example

Relation r

A	В	С	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

•  $\sigma_{A=B^{A}D>5}(r)$ 





Notation:

 $\sigma_p(r)$ 

where *p* is a <u>selection predicate</u> and *r* is a relation (or more generally, a relational algebra expression).

Defined as:

 $\sigma_p(\mathbf{r}) = \{t \mid t \in r \text{ and } p(t)\}$ 

where *p* is a formula in propositional logic consisting of <u>terms</u> connected by:  $\land$  (**and**),  $\lor$  (**or**),  $\neg$  (**not**), and where each term can involve the comparison operators: =,  $\neq$ , >, ≥, <, ≤

\* Note that, in the books notation, the predicate p cannot contain a subquery.



Example:

 $\sigma_{\textit{branch-name="Perryridge"}}$ (account)

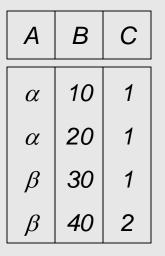
σ customer-name="Smith" ^ customer-street = "main" (Customer)

Logically, one can think of selection as performing a table scan, but technically this may or may not be the case, i.e., an index may be used; that's why relational algebra is most frequently referred to as non-procedural.



## Project Operation – Example

Relation r.



■ ∏<sub>A,C</sub> (*r*)

$$\begin{array}{c|cccc}
A & C \\
\hline \alpha & 1 \\
\alpha & 1 \\
\beta & 1 \\
\hline \beta & 2 \\
\hline \end{array} \begin{array}{c}
A & C \\
\hline \alpha & 1 \\
\beta & 1 \\
\hline \beta & 2 \\
\hline \end{array}$$



### **Project Operation**

Notation:

### $\prod_{A1, A2, ..., Ak} (r)$

where  $A_1$ ,  $A_2$  are attribute names and *r* is a relation.

- The result is defined as the relation of k columns obtained by erasing the columns that are not listed.
- Duplicate rows are <u>removed</u> from result, since relations are <u>sets</u>.
- Example:

 $\prod_{\text{account-number, balance}} (\text{account})$ 

Note, however, that account is not actually modified.



The projection operation can also be used to reorder attributes.

 $\prod_{branch-name, \ balance, \ account-number}$  (account)

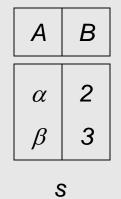
As before, however, note that account is not actually modified; the order of the attributes is modified only in the result of the expression.



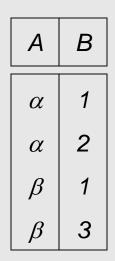
## Union Operation – Example

Relations r, s:

A	В		
α	1		
α	2		
β	1		
r			



 $\mathbf{r} \cup \mathbf{s}$ 





### Union Operation

- **Notation:**  $r \cup s$
- Defined as:

 $r \cup s = \{t \mid t \in r \text{ or } t \in s\}$ 

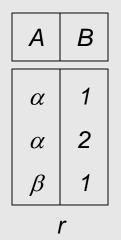
- Union can only be taken between *compatible* relations.
  - *r* and *s* must have the same arity (same number of attributes)
  - > attribute domains of *r* and *s* must be compatible (e.g., 2nd attribute of *r* deals with "the same type of values" as does the 2nd attribute of *s*)
- Example: find all customers with either an account or a loan

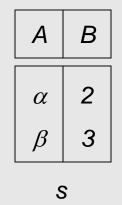
 $\prod_{customer-name} (depositor) \cup \prod_{customer-name} (borrower)$ 



### Set Difference Operation

Relations *r*, *s*:





r-s





## Set Difference Operation, Cont.

- Notation r s
- Defined as:

#### $r-s = \{t \mid t \in r \text{ and } t \notin s\}$

Set difference can only be taken between *compatible* relations.

- r and s must have the same arity
- > attribute domains of *r* and *s* must be compatible

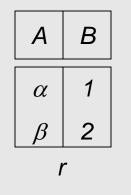
Note that there is no requirement that the attribute names be the same.

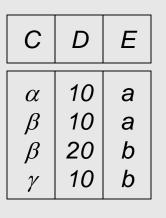
- So what about attributes names in the result?
- Similarly for union.



# **Cartesian-Product Operation**

Relations r, s:





rxs:

S

$$A$$
 $B$ 
 $C$ 
 $D$ 
 $E$ 
 $\alpha$ 
 1
  $\alpha$ 
 10
  $a$ 
 $\alpha$ 
 1
  $\beta$ 
 10
  $a$ 
 $\alpha$ 
 1
  $\beta$ 
 20
  $b$ 
 $\alpha$ 
 1
  $\gamma$ 
 10
  $a$ 
 $\alpha$ 
 1
  $\beta$ 
 20
  $b$ 
 $\alpha$ 
 1
  $\gamma$ 
 10
  $a$ 
 $\beta$ 
 2
  $\alpha$ 
 10
  $a$ 
 $\beta$ 
 2
  $\beta$ 
 10
  $a$ 
 $\beta$ 
 2
  $\beta$ 
 20
  $b$ 
 $\beta$ 
 2
  $\beta$ 
 10
  $a$ 
 $\beta$ 
 2
  $\beta$ 
 20
  $b$ 
 $\beta$ 
 2
  $\beta$ 
 20
  $b$ 
 $\beta$ 
 2
  $\gamma$ 
 10
  $a$ 
 $\beta$ 
 2
  $\gamma$ 
 10
  $b$ 



- Notation r x s
- Defined as:

$$r \times s = \{tq \mid t \in r \text{ and } q \in s\}$$

- In some cases the attributes of *r* and *s* are disjoint, i.e., that  $R \cap S = \emptyset$ .
- If the attributes of *r* and *s* are not disjoint:
  - Each attributes' name has its originating relations name as a prefix.
  - > If r and s are the same relation, then the rename operation can be used.



- The rename operator allows the results of an expression to be renamed.
- The operator appears in two forms:
  - $\rho_{x}(E)$   $\rho_{x(A1, A2, ..., An)}(E)$
- returns the expression *E* under the name *X*returns the expression *E* under name *X*, with
  - attributes renamed to A1, A2,..., An
- Typically used to resolve a name class or ambiguity.



#### Expressions can be built using multiple operations

rxs

A	В	С	D	Е
α	1	α	10	а
α	1	β	10	а
α	1	β	20	b
α	1	γ	10	b
$\beta$	2	α	10	а
$\beta$	2	β	10	а
$\beta$	2	$\beta$	20	b
β	2	γ	10	b

$$\sigma_{A=C}(r x s)$$

A	В	С	D	E
α	1	α	10	а
$\beta$	2	$\beta$	20	а
β	2	$\beta$	20	b



Formal (recursive) Definition of a Relational Algebraic Expression

- A basic expression in relational algebra consists of one of the following:
  - A relation in the database
  - A constant relation
- Let  $E_1$  and  $E_2$  be relational-algebra expressions. Then the following are all also relational-algebra expressions:
  - $\succ E_1 \cup E_2$
  - $\succ$   $E_1 E_2$
  - $\succ$   $E_1 \times E_2$
  - >  $\sigma_p(E_1)$ , *P* is a predicate on attributes in  $E_1$
  - >  $\prod_{s}(E_{1})$ , S is a list consisting of attributes in  $E_{1}$
  - $\succ \rho_x(E_1)$ , x is the new name for the result of  $E_1$



Recall the relational schemes from the banking enterprise:

branch (<u>branch-name</u>, branch-city, assets) customer (<u>customer-name</u>, customer-street, customer-city) account (<u>account-number</u>, branch-name, balance) loan (<u>loan-number</u>, branch-name, amount) depositor (<u>customer-name</u>, <u>account-number</u>) borrower (<u>customer-name</u>, <u>loan-number</u>)



Find all loans of over \$1200 (a bit ambiguous).

 $\sigma_{amount > 1200}$  (loan)

Find the loan number for each loan with an amount greater than \$1200.

 $\prod_{loan-number} (\sigma_{amount > 1200} (loan))$ 



Find the names of all customers who have a loan, an account, or both.

$$\prod_{customer-name}$$
 (borrower)  $\cup \prod_{customer-name}$  (depositor)

Find the names of all customers who have a loan and an account.

 $\prod_{customer-name}$  (borrower)  $\cap \prod_{customer-name}$  (depositor)



### Find the names of all customers who have a loan at the Perryridge branch.

 $\prod_{customer-name} (\sigma_{branch-name="Perryridge"} (\sigma_{borrower.loan-number=loan.loan-number} (borrower x loan)))$ 

#### Notes:

- > There is no "looping" construct in relational algebra, hence the Cartesian product.
- > The two selections could have been combined into one.
- > The selection on *branch-name* could have been applied to *loan* first, as shown next...



 Alternative - Find the names of all customers who have a loan at the Perryridge branch.

 $\prod_{customer-name}(\sigma_{loan.loan-number = borrower.loan-number}(borrower \times \sigma_{branch-name = "Perryridge"}(loan)))$ 

- Notes:
  - > What are the implications of doing the selection first?
  - How does a non-Perryridge borrower tuple get eliminated?
  - > Couldn't the amount and branch-name be eliminated from loan early on?
  - > What would be the implications?



Find the names of all customers who have a loan at the Perryridge branch but no account at any branch of the bank.

 $\Pi_{customer-name} (\sigma_{branch-name = "Perryridge"} (\sigma_{borrower.loan-number = loan.loan-number} (borrower x loan))) - \Pi_{customer-name} (depositor)$ 

A general query writing strategy – start with something simpler, and then enhance.



#### Find the largest account balance:

- Requires comparing each account balance to every other account balance.
- > Accomplished by performing a Cartesian product between account and itself.
- > Unfortunately, this results in ambiguity of attribute names.
- Resolved by renaming one instance of the *account* relation as *d*.

 $\prod_{balance} (account) - \prod_{account.balance} (\sigma_{account.balance < d.balance} (account \times \rho_d (account)))$ 



- The following operations do not add any "power," or rather, capability to relational algebra queries, but simplify common queries.
  - Set intersection
  - Natural join
  - Theta join
  - Outer join
  - Division
  - > Assignment
- All of the above can be defined in terms of the six basic operators.



## Set-Intersection Operation

• Notation:  $r \cap s$ 

Defined as:

 $r \cap s = \{ t \mid t \in r \text{ and } t \in s \}$ 

#### Assume:

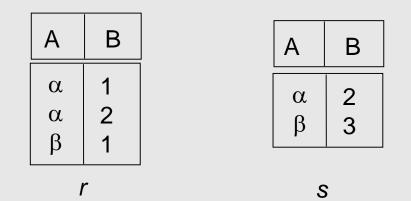
- *r*, *s* have the same arity
- > attributes of *r* and *s* are compatible
- In terms of the 6 basic operators:

$$r \cap s = r - (r - s)$$

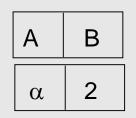


# Set-Intersection Operation, Cont.

Relation r, s:



I ∩ S





## Natural-Join Operation

### ■ Notation: r 🖂 s

Let r and s be relations on schemas R and S respectively.

#### • $r \bowtie s$ is a relation that:

- > Has all attributes in  $R \cup S$
- For each pair of tuples  $t_r$  and  $t_s$  from *r* and *s*, respectively, if  $t_r$  and  $t_s$  have the same value on <u>all</u> attributes in  $R \cap S$ , add a "joined" tuple *t* to the result.
- Joining two tuples  $t_r$  and  $t_s$  creates a third tuple t such that:
  - $\succ$  t has the same value as  $t_r$  on attributes in R
  - $\succ$  t has the same value as  $t_s$  on attributes in S



Relational schemes for relations *r* and *s*, respectively:

R = (A, B, C, D)S = (E, B, D) -- Note the common attributes, which is typical.

Resulting schema for  $r \bowtie s$ :

(A, B, C, D, E)

In terms of the 6 basic operators  $r \bowtie s$  is defined as:

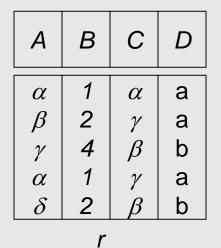
 $\prod_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B=s.B} \wedge_{r.D=s.D} (r \times s))$ 

More generally, computing the natural join equates to a Cartesian product, followed by a selection, followed by a projection.



### Natural Join Example

#### Relations *r*, *s*:



В	D	Е
1	а	α
3	а	$\beta$
1	а	$\gamma \\ \delta$
2 3	b	-
3	b	E

S

• Contents of  $r \bowtie s$ :

A	В	С	D	Ε
α	1	α	а	α
α	1	α	а	γ
α	1	γ	а	α
α	1	γ	а	γ
$\delta$	2	$\beta$	b	$\delta$



Find the names of all customers who have a loan at the Perryridge branch.

Original Expression:

 $\prod_{customer-name} (\sigma_{branch-name="Perryridge"} (\sigma_{borrower.loan-number=loan.loan-number} (borrower x loan)))$ 

Using the Natural Join Operator:

 $\prod_{customer-name} (\sigma_{branch-name = "Perryridge"} (borrower \bowtie loan))$ 

Specifying the join explicitly makes it look nicer, plus it helps the query optimizer.



Find the instructor ID's for those who teach in the Crawford building.

 $\prod_{ID}(\sigma_{building = "Crawford"} (teaches \bowtie section))$ 

In this case the natural join is on four attributes – course\_id, section\_id, semester, and year.



## Theta-Join Operation

- Notation:  $r \Join_{\theta} s$
- Let *r* and *s* be relations on schemas *R* and *S* respectively, and let  $\theta$  be a predicate.
- Then,  $r \Join_{\theta} s$  is a relation that:
  - > Has all attributes in  $R \cup S$  *including duplicate attributes*.
  - For each pair of tuples  $t_r$  and  $t_s$  from r and s, respectively, if  $\theta$  evaluates to true for  $t_r$  and  $t_s$ , then add a "joined" tuple t to the result.
- In terms of the 6 basic operators  $r \Join_{\theta} s$  is defined as:

 $\sigma_{\theta} (r \times s)$ 



### Theta-Join Example #1

### Example:

- R = (A, B, C, D)S = (E, B, D)
- Resulting schema:

(*r*.*A*, *r*.*B*, *r*.*C*, *r*.*D*, *s*.*E*, *s*.*B*, *s*.*D*)



Consider the following relational schemes:

Score = (<u>ID#</u>, <u>Exam#</u>, Grade) Exam = (<u>Exam#</u>, Average)

Consider the following query:

"Find the ID#s for those students who scored less than average on some exam."

 $\prod_{\text{Score.ID#}} (\text{Score} \bowtie_{\text{Score.Exam} = \text{Exam.Exam} \land \text{Score.Grade} < \text{Exam.Average} \text{Exam})$ 

Note the above could also be done with a natural join, followed by a selection.



Consider the following relational schemes: (Orlando temperatures)

Temp-Avgs = (<u>Year</u>, Avg-Temp) Daily-Temps-2010 = (<u>Date</u>, High-Temp)

Consider the following query:

*"Find the days during 2010 where the high temperature for the day was higher than the average for some prior year."* 

 $\prod_{Date} (Daily-Temps-2010 \bowtie_{Daily-Temps-2010.High-Temp > Temp-Avgs.Avg-Temp \land Temp-Avgs.Year < 2010} Temp-Avgs)$ 

Looks ugly, perhaps, but phrasing the query this way does have benefits for query optimization.



- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that do not match tuples in the other relation.
- Typically introduces *null* values.



### *Outer Join – Example*

### Relation *loan:*

loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

#### Relation *borrower*:

customer-name	loan-number
Jones	L-170
Smith	L-230
Hayes	L-155



### Outer Join – Example

### Inner Join

#### *loan* $\bowtie$ *Borrower*

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

#### Left Outer Join

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	null



### *Outer Join – Example*

### Right Outer Join

*loan* ⋈\_ *borrower* 

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	null	null	Hayes

### Full Outer Join

*loan* \_>\_*borrower* 

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	null
L-155	null	null	Hayes



Consider the following relational schemes:

Student = (<u>SS#</u>, Address, Date-of-Birth)

Grade-Point-Average = (<u>SS#</u>, GPA)

Consider the following query:

*"Create a list of all student SS#'s and their GPAs. Be sure to include <u>all</u> <i>students, including first semester freshman, who do not have a GPA."* 

Solution:

 $\prod_{SS\#,GPA}$  (Student  $\square \lor$  Grade-Point-Average)



In terms of the 6 basic operators (plus natural join <sup>©</sup>), let *r(R)* and *s(S)* be relations:

 $r \sqsupseteq s = (r - \prod_{R} (r \bowtie s)) \times \{(null, null, \dots, null)\} \cup (r \bowtie s)$ 

where  $\{(null, null, ..., null)\}$  is on the schema S - R



## **Division Operation**

- Notation:  $r \div s$
- Suited to queries that require "universal quantification," e.g., include the phrase "for all."



• Let *r* and *s* be relations on schemas *R* and *S* respectively where  $S \subseteq R$ .

Assume without loss of generality that the attributes of *R* and *S* are:

$$R = (A_1, ..., A_m, B_1, ..., B_n)$$
  
S = (B<sub>1</sub>, ..., B<sub>n</sub>)

The  $A_i$  attributes will be referred to as *prefix* attributes, and the  $B_i$  attributes will be referred to as *suffix* attributes.

The result of  $r \div s$  is a relation on schema

$$R-S=(A_1,\ldots,A_m)$$

where:

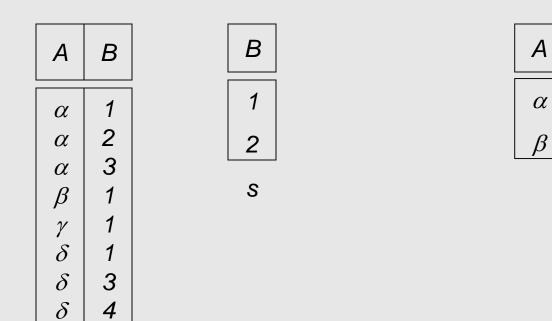
 $r \div s = \{ t \mid t \in \prod_{R-S}(r) \land \forall u \in s (tu \in r) \}$ 



# Division – Example #1

### Relations *r*, *s*:

*r* ÷ s:



Database System Concepts

 $\in$ 

 $\in$ 

β

r

6

1

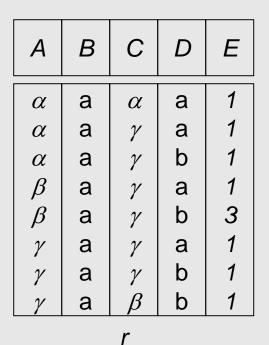
2



# Division – Example #2

### Relations *r*, *s*:

*r* ÷ s:





S

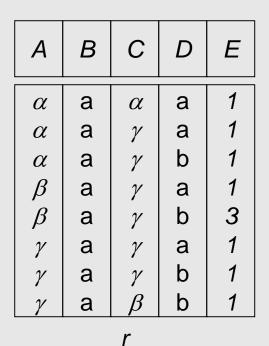
A	В	С
α	а	γ
γ	а	γ

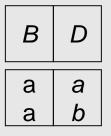


# Division – Example #3

### Relations *r, s*:







S

A	С	Е
$\left  \begin{array}{c} \alpha \\ \gamma \end{array} \right $	Y Y	1 1



In terms of the 6 basic operators, let r(R) and s(S) be relations, and let  $S \subseteq R$ :

$$r \div s = \prod_{R-S} (r) - \prod_{R-S} ((\prod_{R-S} (r) \times s) - \prod_{R-S,S} (r))$$

To see why:

- >  $\prod_{R-S,S}(r)$  simply reorders attributes of r
- >  $\prod_{R-S}(\prod_{R-S} (r) \ge s) \prod_{R-S,S}(r))$  gives those tuples t in  $\prod_{R-S} (r)$  such that for some tuple *u* ∈ *s*, *tu* ∉ *r*.
- Property:
  - $\blacktriangleright$  Let  $q = r \div s$
  - > Then q is the largest relation satisfying  $q \ge r$



• Consider the following query:

"Find the names of all customers who have an account at both the 'Downtown' and the 'Uptown' branches."

Query 1:

 $\prod_{CN}(\sigma_{BN="Downtown"}(\textit{depositor} \boxtimes \textit{account})) \cap \prod_{CN}(\sigma_{BN="Uptown"}(\textit{depositor} \boxtimes \textit{account}))$ 

Query 2:

 $\prod_{customer-name, branch-name}$  (depositor  $\bowtie$  account)  $\div \rho_{temp(branch-name)}$  ({("Downtown"), ("Uptown")})



Consider the following (more general) query:

"Find all customers who have an account at all branches located in the city of Brooklyn."

- How could Query 1 be modified for this scenario?
- How about Query 2?

 $\prod_{customer-name, branch-name} (depositor \bowtie account) \div \prod_{branch-name} (\sigma_{branch-city = "Brooklyn"} (branch))$ 

By the way, what would (should) be the result of the query if there are no Brooklyn branches?



- The assignment operator ( $\leftarrow$ ) provides an easy way to express complex queries.
- Example (for  $r \div s$ ):

 $temp1 \leftarrow \prod_{R-S} (r)$  $temp2 \leftarrow \prod_{R-S} ((temp1 \times s) - \prod_{R-S,S} (r))$  $result \leftarrow temp1 - temp2$ 

\*Do the exercises on the employee/works/company/manages DB!

\*And also the exercises on the university DB!



- Generalized Projection
- Aggregate Operator



Extends projection by allowing arithmetic functions in the projection list.

```
\prod_{F1, F2, ..., Fn} (E)
```

- *E* is any relational-algebra expression
- Each of  $F_1$ ,  $F_2$ , ...,  $F_n$  are arithmetic expressions involving constants and attributes in the schema of *E*.



Consider the following relational scheme:

credit-info=(customer-name, limit, credit-balance)

Give a relational algebraic expression for the following query:

"Determine how much credit is left on each persons' line of credit; Also determine the percentage of their credit line that they have already used."

 $\prod_{customer-name, limit - credit-balance, (credit-balance/limit)*100}$  (credit-info)



#### An *aggregation function* takes a collection of values and returns a single value:

avg min max sum

count

- average value
- minimum value
- maximum value
- sum of values
- number of values
- Other aggregate functions are provided by most DBMS vendors.
- Not all aggregate operators are numeric, e.g., some apply to strings.



Aggregation functions are used in the <u>aggregate operator</u>.

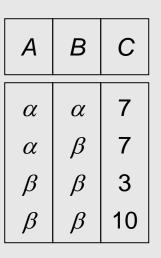
G1, G2, ..., Gn 
$$\mathcal{G}$$
 F1(A1), F2(A2),..., Fn(An) (E)

- $\succ$  E is any relational-algebra expression.
- $\succ$   $G_1, G_2, \dots, G_n$  is a list of attributes on which to group (can be empty).
- $\succ$  Each  $F_i$  is an aggregate function.
- Each  $A_i$  is an attribute name.

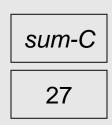


### Aggregate Function – Example

Relation *r*.



 $\boldsymbol{g}_{sum(c)}(r)$ 



Could also add *min*, *max*, and other aggregates to the above expression.

$\boldsymbol{\mathcal{G}}_{sum(c), min(c), max(c)}(r)$	sum-C	min-C	max-C	
	27	3	10	



- Grouping is somewhat like sorting, although not identical.
- Relation *account* grouped by *branch-name*:

account-number	branch-name	balance
A-102	Perryridge	400
A-374	Perryridge	900
A-224	Brighton	175
A-161	Brighton	850
A-435	Brighton	400
A-201	Brighton	625
A-217	Redwood	750
A-215	Redwood	750
A-222	Redwood	700



- Grouping and aggregate functions frequently occur together.
- A list of branch names and the sum of all their account balances:

branch-name g sum(balance) (account)

branch-name	balance
Perryridge	1300
Brighton	2050
Redwood	2200



### Aggregate Operation Grouping on Multiple Attributes

#### Consider the following relational scheme:

2

*History* = (<u>Student-Name</u>, <u>Department</u>, <u>Course-Number</u>, Grade)

#### Sample data:

Student-Name	Department	Course-Number	Grade
Smith	CSE	1001	90
Jones	MTH	2030	82
Smith	MTH	1002	73
Brown	PSY	4210	86
Jones	CSE	2010	65



## Aggregate Operation Grouping on Multiple Attributes

#### Consider the following query:

"Construct a list of student names and, for each name, list the average course grade for each department in which the student has taken classes."

Smith	CSE	87
Smith	MTH	93
Jones	CHM	88
Jones	CSE	75
Brown	PSY	97

Recalling the schema:

*History* = (<u>Student-Name</u>, <u>Department</u>, <u>Course-Number</u>, Grade)

Answer:

student-name, department  $\boldsymbol{g}_{avg(grade)}(History)$ 



## Aggregate Operation Grouping on Multiple Attributes

Adding count(Course-Number) would tell how many courses the student had in each department. Similarly min and max could be added.

student-name, department  $\boldsymbol{g}$  avg(grade), count(Course-Number), min(Grade), max(Grade)(History)



## Aggregate Operation Grouping on Multiple Attributes

Would the following two expressions give the same result?

student-name, department  $\boldsymbol{g}_{avg(grade), count(Course-Number), min(Grade), max(Grade)}(History)$ 

department, student-name g avg(grade), count(Course-Number), min(Grade), max(Grade)(History)



Note that the aggregated attributes do not have names?

 $g_{sum(c), min(c), max(c)}(r)$ 

sum-C	min-C	max-C
27	3	10



Note that the aggregated attributes do not have names?

 $g_{sum(c), min(c), max(c)}(r)$ 

?	?	?
27	3	10

Aggregated attributes can be renamed in the aggregate operator:

branch-name  $\mathcal{G}_{sum(balance)as sum-balance}(account)$ 



# Aggregate Functions and Null Values

- Null values are controversial.
- Various proposals exist in the research literature on whether null values should be allowed and, if so, how they should affect operations.
- Null values can frequently be eliminated through normalization and decomposition.



# Aggregate Functions and Null Values

How nulls are treated by relational operators:

- For duplicate elimination and grouping, null is treated like any other value, i.e., two nulls are assumed to be the same.
- > Aggregate functions (except for *count*) simply ignore null values.
- The above rules are consistent with SQL.
- Note how the second rule can be misleading:
  - Is avg(grade) actually a class average?



# Null Values and Expression Evaluation

#### Null values also affect how selection predicates are evaluated:

- > The result of any arithmetic expression involving *null* is *null*.
- Comparisons with *null* returns the special truth value *unknown*.
- > Value of a predicate is treated as *false* if it evaluates to *unknown*.

 $\sigma_{balance*100 > 500}$  (account)

For more complex predicates, the following <u>three-valued logic</u> is used:

	OR:	(unknown <b>or</b> true) (unknown <b>or</b> false) (unknown <b>or</b> unknown)	= true = unknown = unknown
>	AND:	(true <b>and</b> unknown) (false <b>and</b> unknown) (unknown <b>and</b> unknown)	= unknown = false = unknown
≻	NOT:	(not unknown)	= unknown

 $\sigma_{(balance*100 > 500)}$  and (branch-name = "Perryridge")(account)



# Null Values and Expression Evaluation, Cont.

- Why doesn't a comparison with *null* simply result in *false*?
- If false was used instead of unknown, then:

not (A < 5)

would not be equivalent to:

#### A >= 5

Why would this be a problem?

How does a comparison with *null* resulting in *unknown* help?



- The database contents can be modified with operations:
  - Deletion
  - Insertion
  - Updating
- These operations can all be expressed using the assignment operator.
  - Some can be expressed other ways too.



Deletion

A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

where r is a relation and E is a relational algebra query.

- The deletion of a single tuple is expressed by letting *E* be a constant relation containing one tuple.
- Only whole tuples can be deleted, not specific attribute values.



Forget referential integrity for the moment...

"Delete all account records with a branch name equal to Perryridge."

account  $\leftarrow$  account –  $\sigma$  <sub>branch-name = "Perryridge"</sub> (account)

"Delete all loan records with amount in the range of 0 to 50."

 $loan \leftarrow loan - \sigma_{amount \ge 0 and amount \le 50}$  (loan)



Now suppose we want to maintain proper referential integrity...

"Delete all accounts at branches located in Needham" (Version #1):

 $r_{1} \leftarrow \sigma_{branch-city} = "Needham" (account \bowtie branch)$   $r_{2} \leftarrow \prod_{account-number, branch-name, balance} (r_{1})$   $r_{3} \leftarrow \prod_{customer-name, account-number} (depositor \bowtie r_{2})$   $account \leftarrow account - r_{2}$  $depositor \leftarrow depositor - r_{3}$ 



## Alternative Versions

#### Version #2:

 $\begin{aligned} r_{1} \leftarrow \prod_{branch-name} (\sigma_{branch-city = "Needham"} (branch)) \\ r_{2} \leftarrow \prod_{account-number} (\prod_{account-number, branch-name} (account) \bowtie r_{1}) \\ account \leftarrow account - (account \bowtie r_{2}) \\ depositor \leftarrow depositor - (depositor \bowtie r_{2}) \end{aligned}$ 

Version #3:

 $r_{1} \leftarrow (\sigma_{branch-city} <> "Needham" (depositor <math>\bowtie$  account  $\bowtie$  branch)) account  $\leftarrow \prod_{account-number, branch-name, balance}(r_{1})$ depositor  $\leftarrow \prod_{customer-name, account-number}(r_{1})$ 



## Alternative Versions

#### Version #4:

 $r_{1} \leftarrow account \boxtimes \sigma_{branch-city} <> "Needham" (branch)$ account  $\leftarrow \prod_{account-number, branch-name, balance}(r_{1})$ depositor  $\leftarrow \prod_{customer-name, account-number}(depositor \boxtimes r_{1})$ 

- Which version is preferable?
- Note that the last two do not fit the authors pattern for deletion, i.e., as a set-difference.



In relational algebra, an insertion is expressed by:

$$r \leftarrow r \cup E$$

where r is a relation and E is a relational algebra expression.

The insertion of a single tuple is expressed by letting *E* be a constant relation containing one tuple.



Insert information in the database specifying that Smith has \$1200 in account A-973 at the Perryridge branch.

```
account \leftarrow account \cup {(A-973, "Perryridge", 1200)}
depositor \leftarrow depositor \cup {("Smith", A-973)}
```

Provide, as a gift, a \$200 savings account for all loan customers at the Perryridge branch. Let the loan number serve as the account number for the new savings account.

$$r_{1} \leftarrow (\sigma_{branch-name = "Perryridge"} (borrower \Join loan))$$
  
account  $\leftarrow$  account  $\cup \prod_{loan-number, branch-name, 200} (r_{1})$   
depositor  $\leftarrow$  depositor  $\cup \prod_{customer-name, loan-number} (r_{1})$ 



Generalized projection is used to change one or more values in a tuple.

$$r \leftarrow \prod_{F1, F2, \dots, Fl} (r)$$

- Each  $F_i$  is either:
  - > The *i*th attribute of r, if the *i*th attribute is not updated, or,
  - An expression, involving only constants and attributes of r, which gives a new value for an attribute, when that attribute is to be updated.



Make interest payments by increasing all balances by 5 percent.

account  $\leftarrow \prod_{AN, BN, BAL^* 1.05}$  (account)

where AN, BN and BAL stand for account-number, branch-name and balance, respectively.

Pay 6 percent interest to all accounts with balances over \$10,000 and pay 5 percent interest to all others.

 $account \leftarrow \prod_{AN, BN, BAL * 1.06} (\sigma_{BAL > 10000} (account)) \\ \cup \prod_{AN, BN, BAL * 1.05} (\sigma_{BAL \le 10000} (account))$ 



Views are very important, but we will not consider them until chapter 3.