Chapter 3: Relational Model

- Structure of Relational Databases
- Relational Algebra
- Tuple Relational Calculus
- Domain Relational Calculus
- Extended Relational-Algebra-Operations
- Modification of the Database
- Views

Basic Structure

- Given sets A₁, A₂, ..., A_n a relation r is a subset of A₁ × A₂ × ... × A_n
 Thus a relation is a set of n-tuples (a₁, a₂, ..., a_n) where a_i ∈ A_i
- Example: If

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

Then *r* = {(Jones, Main, Harrison), (Smith, North, Rye), (Curry, North, Rye), (Lindsay, Park, Pittsfield)} is a relation over *customer-name* × *customer-street* × *customer-city*

Relation Schema

- A_1 , A_2 , ..., A_n are attributes
- $R = (A_1, A_2, ..., A_n)$ is a relation schema

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

• r(R) is a *relation* on the relation schema R

customer (Customer-schema)

Relation Instance

- The current values (*relation instance*) of a relation are specified by a table.
- An element *t* of *r* is a *tuple*; represented by a *row* in a table.

customer-name	customer-street	customer-city
Jones	Main	Harrison
Smith	North	Rye
Curry	North	Rye
Lindsay	Park	Pittsfield
customer		

Keys • Let $K \subset R$ • K is a superkey of R if values for K are sufficient to identify a unique tuple of each possible relation r(R). By "possible r" we mean a relation r that could exist in the enterprise we are modeling. Example: {*customer-name, customer-street*} and {*customer-name*} are both superkeys of *Customer*, if no two customers can possibly have the same name. • *K* is a *candidate key* if *K* is minimal Example: {*customer-name*} is a candidate key for *Customer*, since it is a superkey (assuming no two customers can possibly

have the same name), and no subset of it is a superkey.

Determining Keys from E-R Sets

- Strong entity set. The primary key of the entity set becomes the primary key of the relation.
- Weak entity set. The primary key of the relation consists of the union of the primary key of the strong entity set and the discriminator of the weak entity set.
- **Relationship set.** The union of the primary keys of the related entity sets becomes a super key of the relation.

For binary many-to-many relationship sets, above super key is also the primary key.

For binary many-to-one relationship sets, the primary key of the "many" entity set becomes the relation's primary key.

For one-to-one relationship sets, the relation's primary key can be that of either entity set.

Query Languages

- Language in which user requests information from the database.
- Categories of languages:
 - Procedural
 - Non-procedural
- "Pure" languages:
 - Relational Algebra
 - Tuple Relational Calculus
 - Domain Relational Calculus
- Pure languages form underlying basis of query languages that people use.

Relational Algebra

- Procedural language
- Six basic operators
 - select
 - project
 - union
 - set difference
 - Cartesian product
 - rename
- The operators take two or more relations as inputs and give a new relation as a result.

Select Operation

- Notation: $\sigma_P(r)$
- Defined as:

$$\sigma_P(r) = \{t \mid t \in r \text{ and } P(t)\}$$

Where *P* is a formula in propositional calculus, dealing with terms of the form:

≠

>

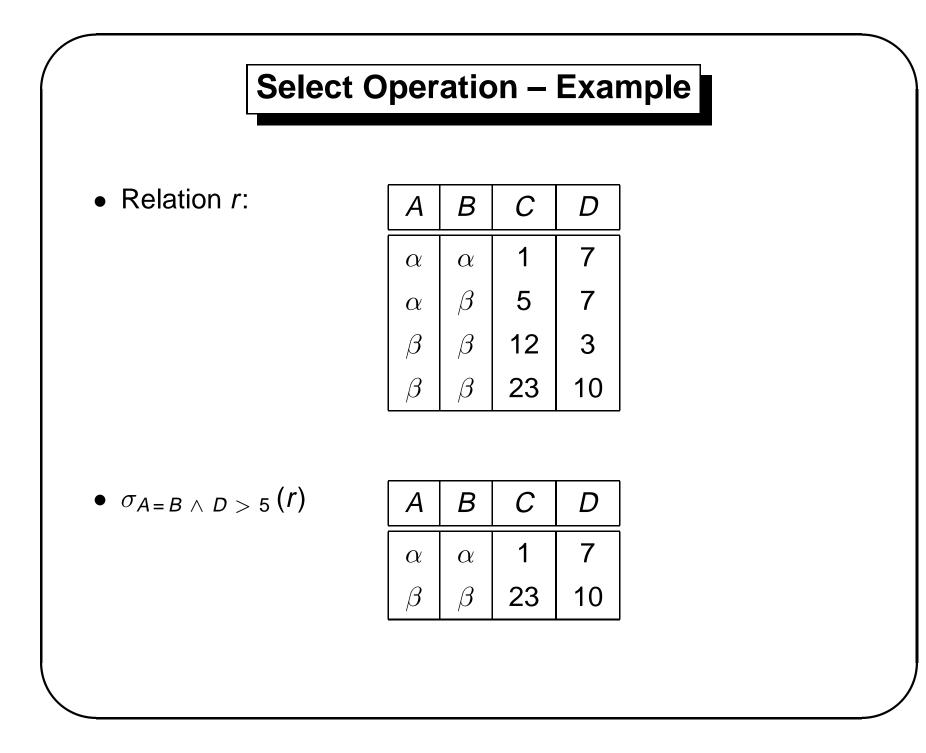
 \geq

<

 \leq

<attribute> = <attribute> or <constant>





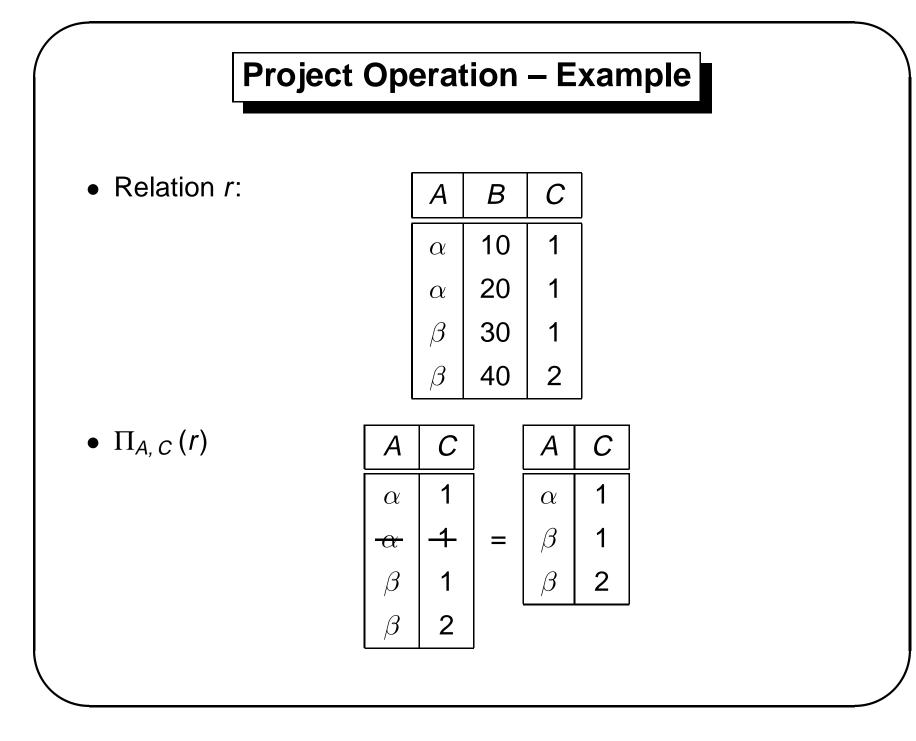
Project Operation

• Notation:

$$\Pi_{A_{1}, A_{2}, ..., A_{k}}(r)$$

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

- The result is defined as the relation of *k* columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets

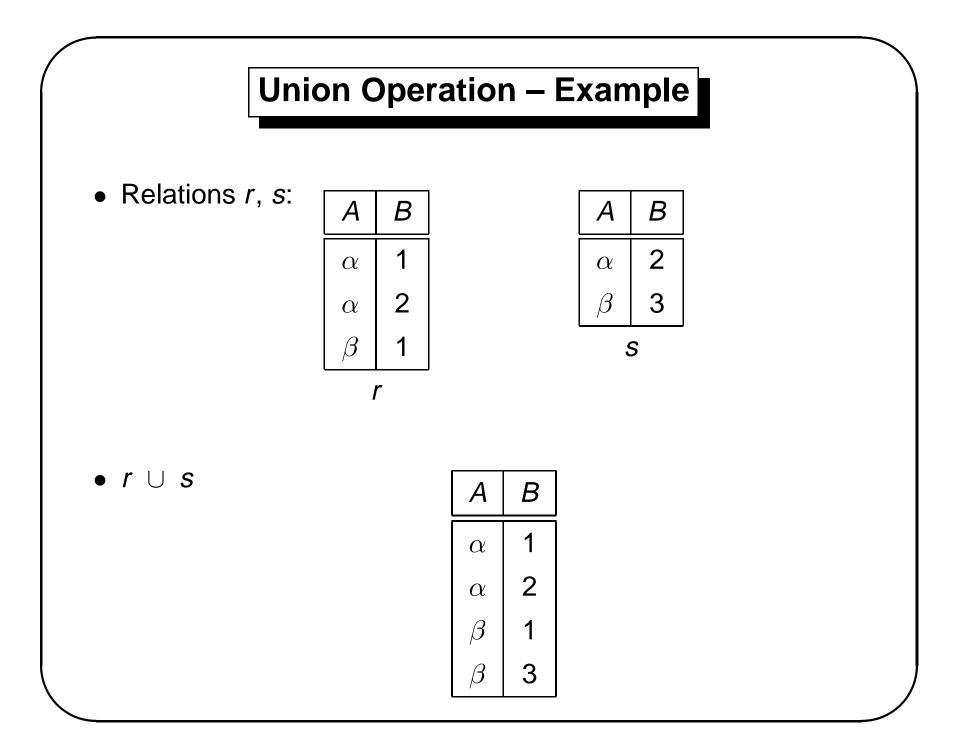


Union Operation

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

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

- For $r \cup s$ to be valid,
 - 1. r, s must have the same arity (same number of attributes)
 - The attribute domains must be *compatible* (e.g., 2nd column of *r* deals with the same type of values as does the 2nd column of *s*)

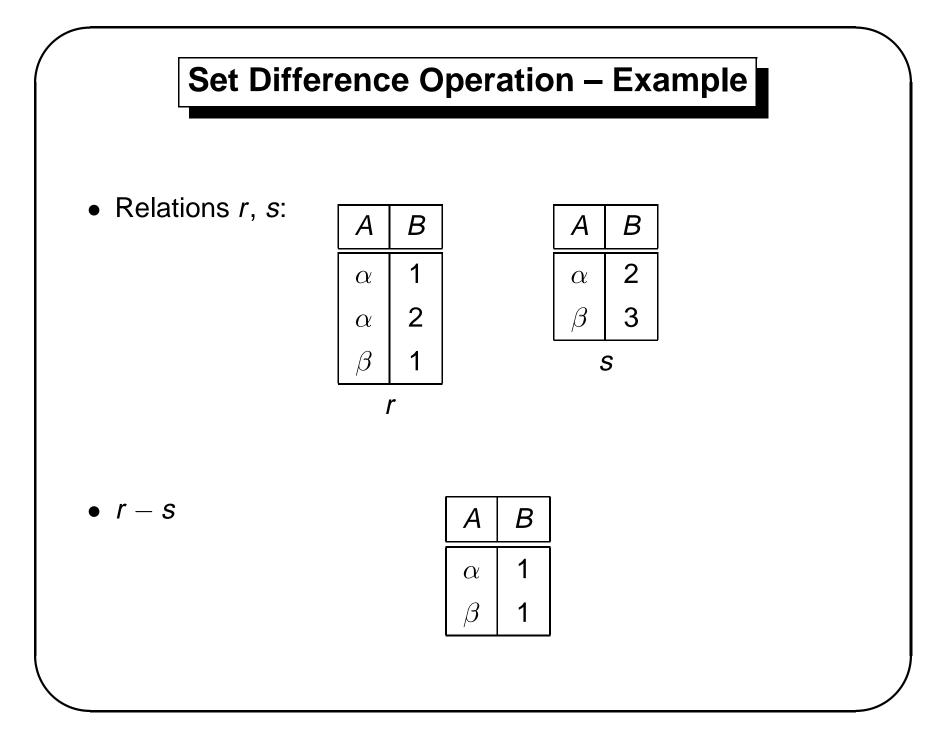


Set Difference Operation

- Notation: r s
- Defined as:

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

- Set differences must be taken between *compatible* relations.
 - *r* and *s* must have the *same arity*
 - attribute domains of *r* and *s* must be compatible

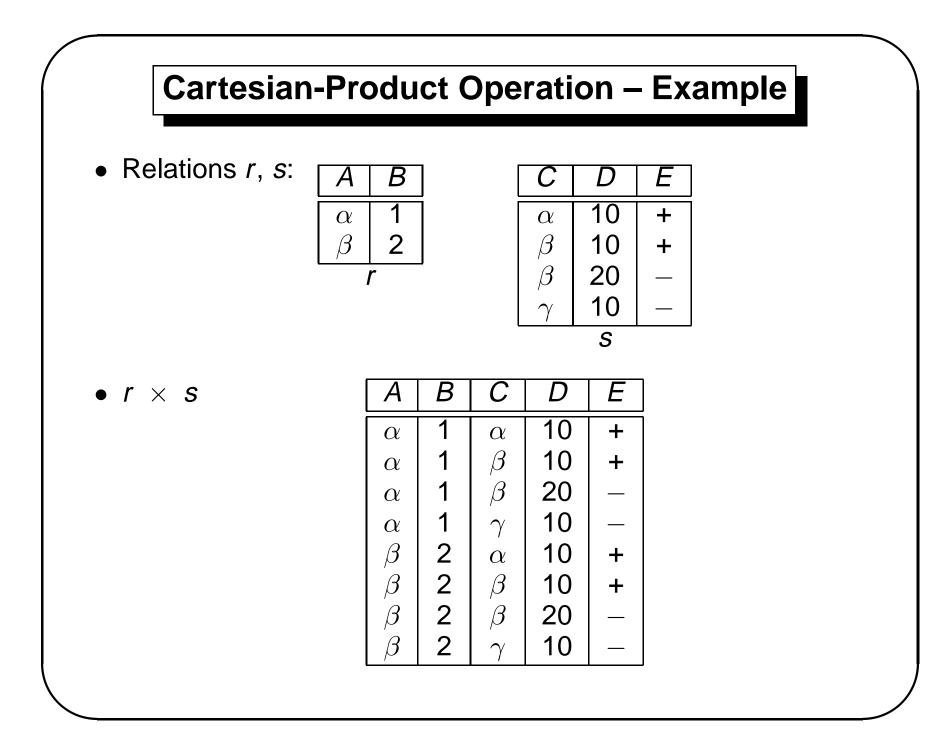


Cartesian-Product Operation

- Notation: $r \times s$
- Defined as:

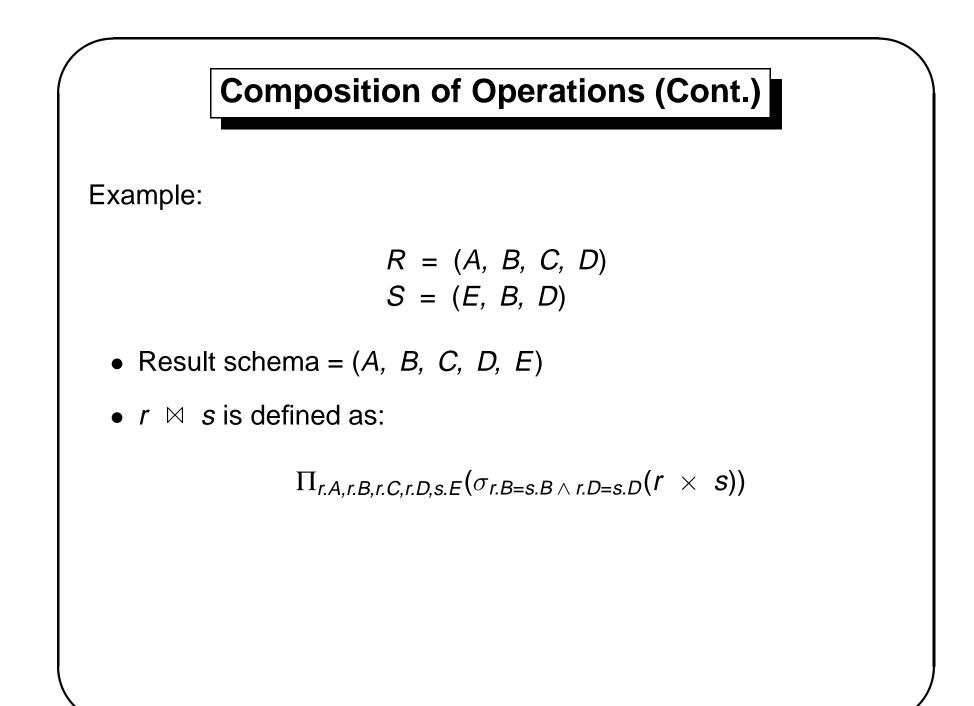
$$r \times s = \{tq \mid t \in r \text{ and } q \in s\}$$

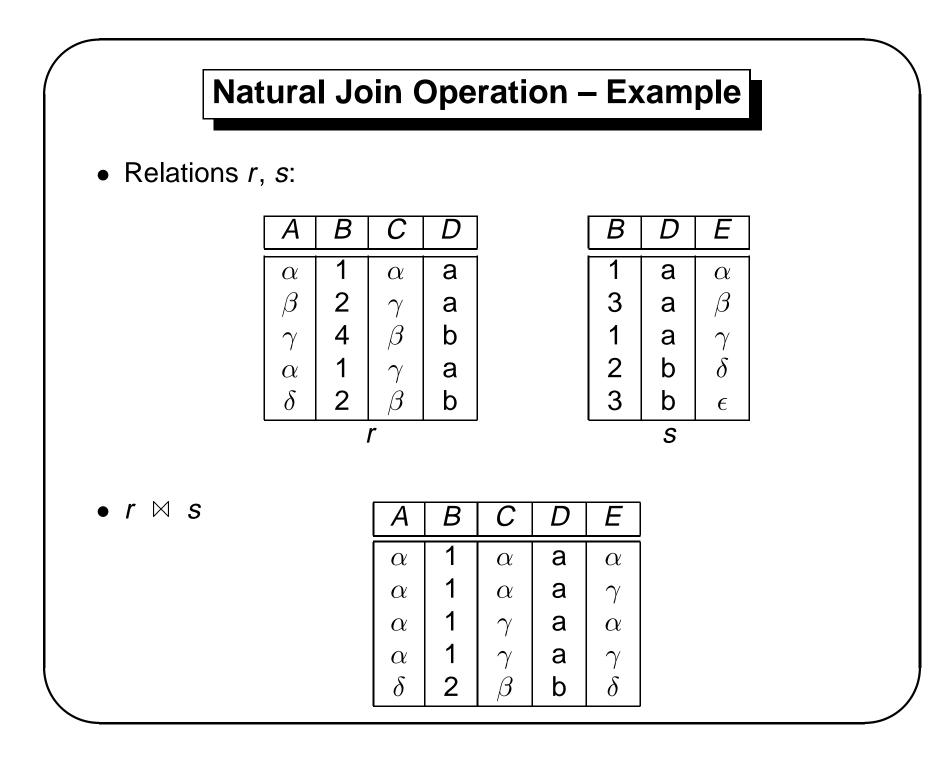
- Assume that attributes of r(R) and s(S) are disjoint. (That is, $R \cap S = \emptyset$).
- If attributes of *r*(*R*) and *s*(*S*) are not disjoint, then renaming must be used.

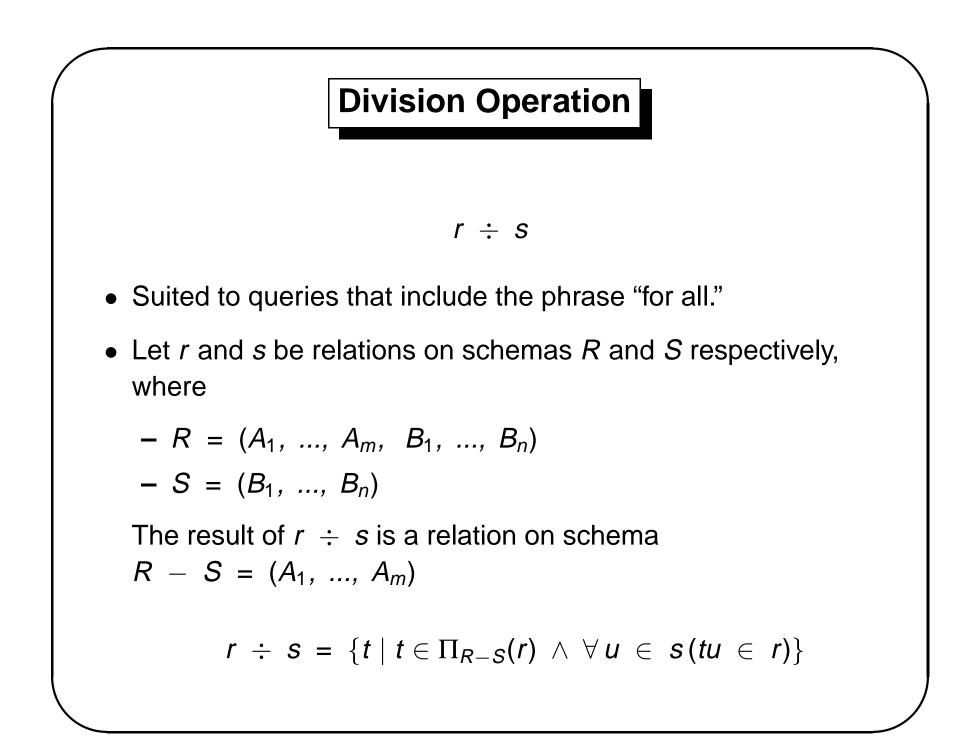


Composition of Operations

- Can build expressions using multiple operations
- Example: $\sigma_{A=C}(r \times s)$
- *r* × s
 - Notation: $r \bowtie s$
 - Let *r* and *s* be relations on schemas *R* and *S* respectively. The result is a relation on schema $R \cup S$ which is obtained by considering each pair of tuples t_r from *r* and t_s from *s*.
 - If t_r and t_s have the same value on each of the attributes in $R \cap S$, a tuple *t* is added to the result, where
 - * *t* has the same value as t_r on *r*
 - * *t* has the same value as t_s on *s*







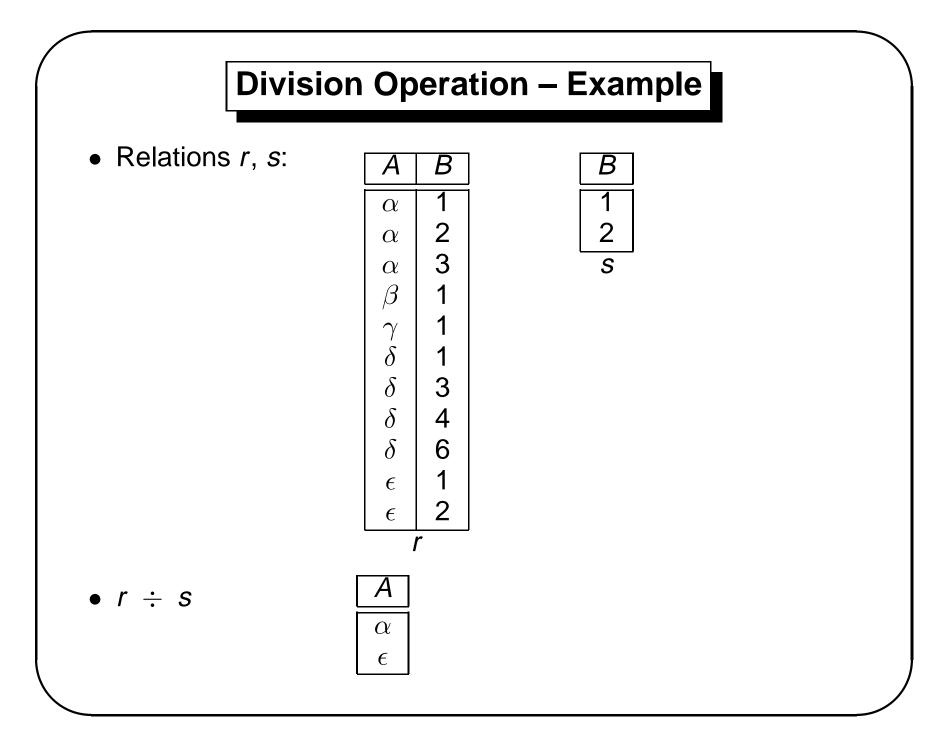
Division Operation (Cont.)

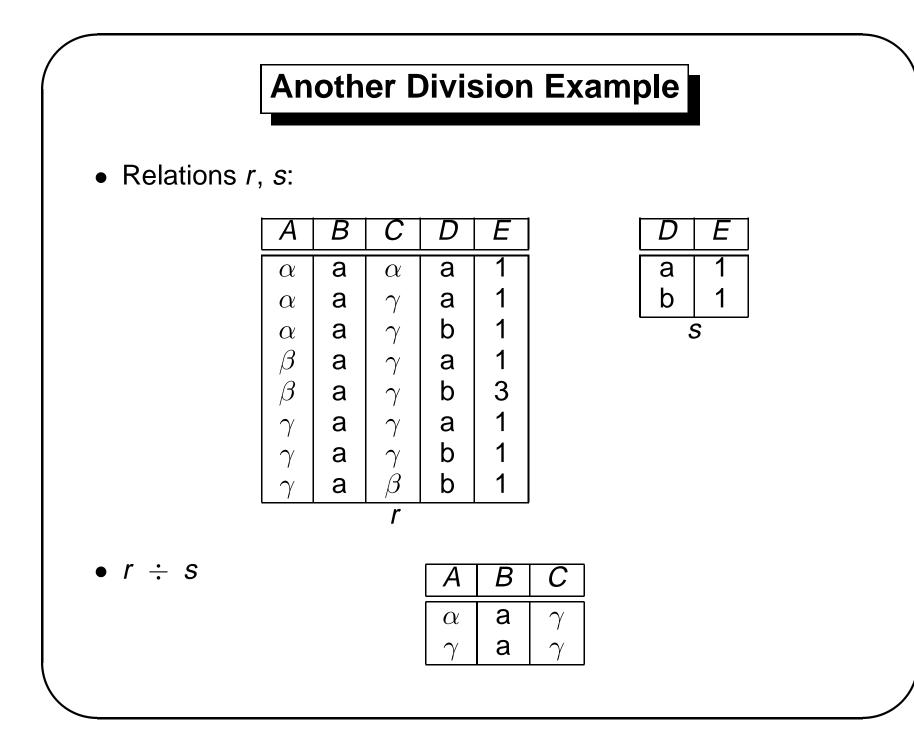
- Property
 - Let $q = r \div s$
 - Then *q* is the largest relation satisfying: $q \times s \subseteq r$
- Definition in terms of the basic algebra operation Let r(R) and s(S) be relations, and let $S \subseteq R$

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

To see why:

- $\Pi_{R-S,S}(r)$ simply reorders attributes of *r*
- $\Pi_{R-S}((\Pi_{R-S}(r) \times s) \Pi_{R-S,S}(r))$ gives those tuples *t* in $\Pi_{R-S}(r)$ such that for some tuple $u \in s$, $tu \notin r$.





Assignment Operation

- The assignment operation (←) provides a convenient way to express complex queries; write query as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.
- Assignment must always be made to a temporary relation variable.
- Example: Write $r \div s$ as

$$temp1 \leftarrow \Pi_{R-S}(r)$$

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

$$result = temp1 - temp2$$

- The result to the right of the \leftarrow is assigned to the relation variable on the left of the \leftarrow .
- May use variable in subsequent expressions.

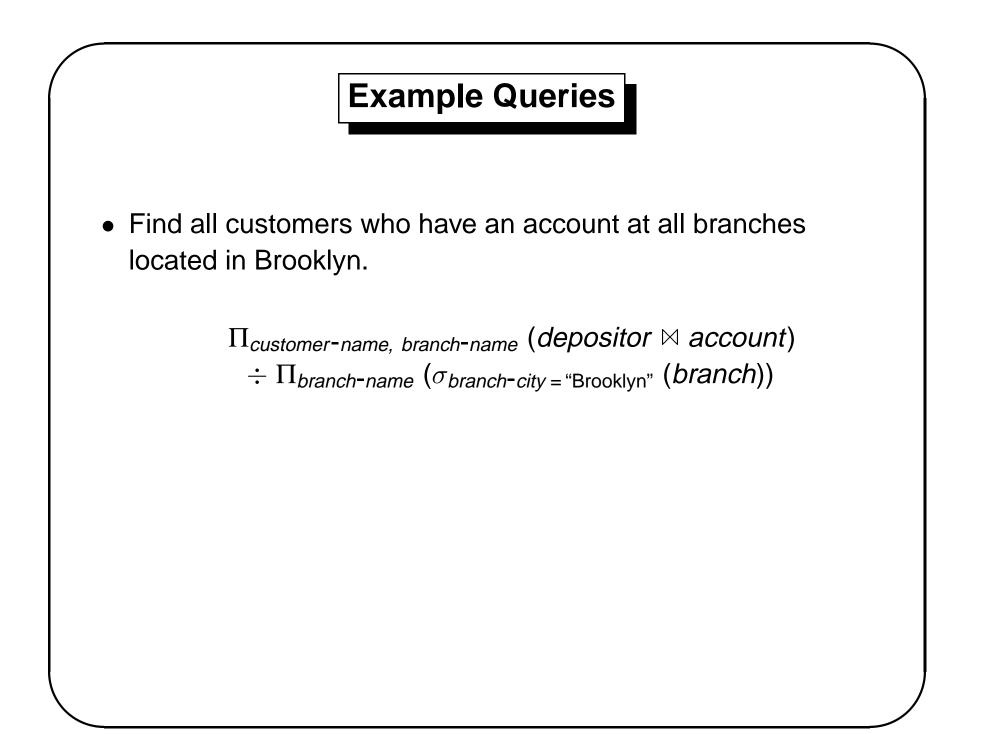
- Find all customers who have an account from at least the "Downtown" and "Uptown" branches.
 - Query 1

 $\Pi_{CN}(\sigma_{BN} = "Downtown" (depositor \bowtie account)) \cap \\\Pi_{CN}(\sigma_{BN} = "Uptown" (depositor \bowtie account))$

where *CN* denotes *customer-name* and *BN* denotes *branch-name*.

– Query 2

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



Tuple Relational Calculus

A nonprocedural query language, where each query is of the form

$\{t \mid P(t)\}$

- It is the set of all tuples *t* such that predicate *P* is true for *t*
- *t* is a *tuple variable*; *t*[*A*] denotes the value of tuple *t* on attribute *A*
- $t \in r$ denotes that tuple *t* is in relation *r*
- *P* is a *formula* similar to that of the predicate calculus

Predicate Calculus Formula

- 1. Set of attributes and constants
- 2. Set of comparison operators: (e.g., <, \leq , =, \neq , >, \geq)
- 3. Set of connectives: and (\land), or (\lor), not (\neg)
- 4. Implication (\Rightarrow): $x \Rightarrow y$, if x if true, then y is true

$$x \Rightarrow y \equiv \neg x \lor y$$

- 5. Set of quantifiers:
 - $\exists t \in r (Q(t)) \equiv$ "there exists" a tuple *t* in relation *r* such that predicate Q(t) is true

•
$$\forall t \in r (Q(t)) \equiv Q$$
 is true "for all" tuples t in relation r

Banking Example

branch (branch-name, branch-city, assets)

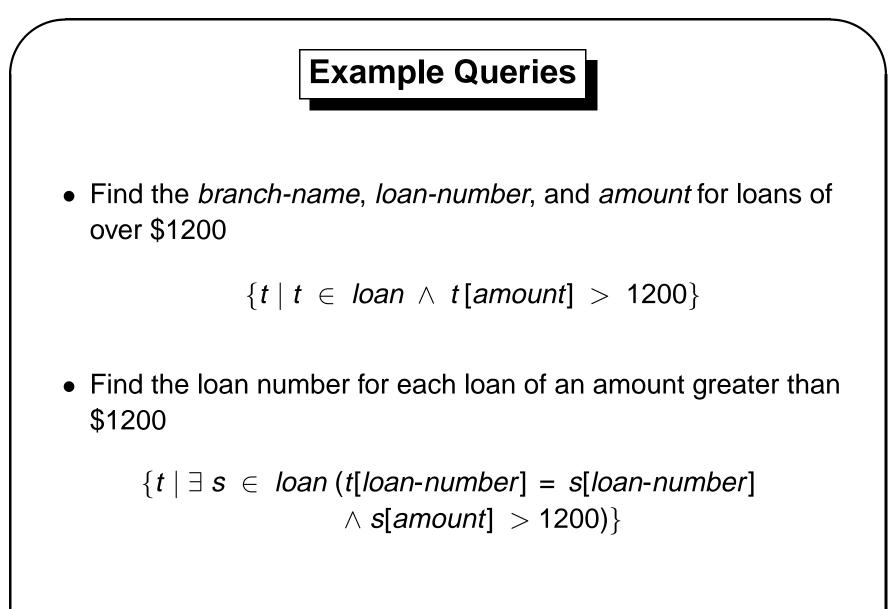
customer (customer-name, customer-street, customer-city)

account (branch-name, account-number, balance)

loan (branch-name, loan-number, amount)

depositor (customer-name, account-number)

borrower (customer-name, loan-number)



Notice that a relation on schema [*customer-name*] is implicitly defined by the query

• Find the names of all customers having a loan, an account, or both at the bank

 $\{t \mid \exists s \in borrower(t[customer-name] = s[customer-name]) \\ \lor \exists u \in depositor(t[customer-name] = u[customer-name]) \}$

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

 $\{t \mid \exists s \in borrower(t[customer-name] = s[customer-name]) \land \exists u \in depositor(t[customer-name] = u[customer-name]) \}$

• Find the names of all customers having a loan at the Perryridge branch

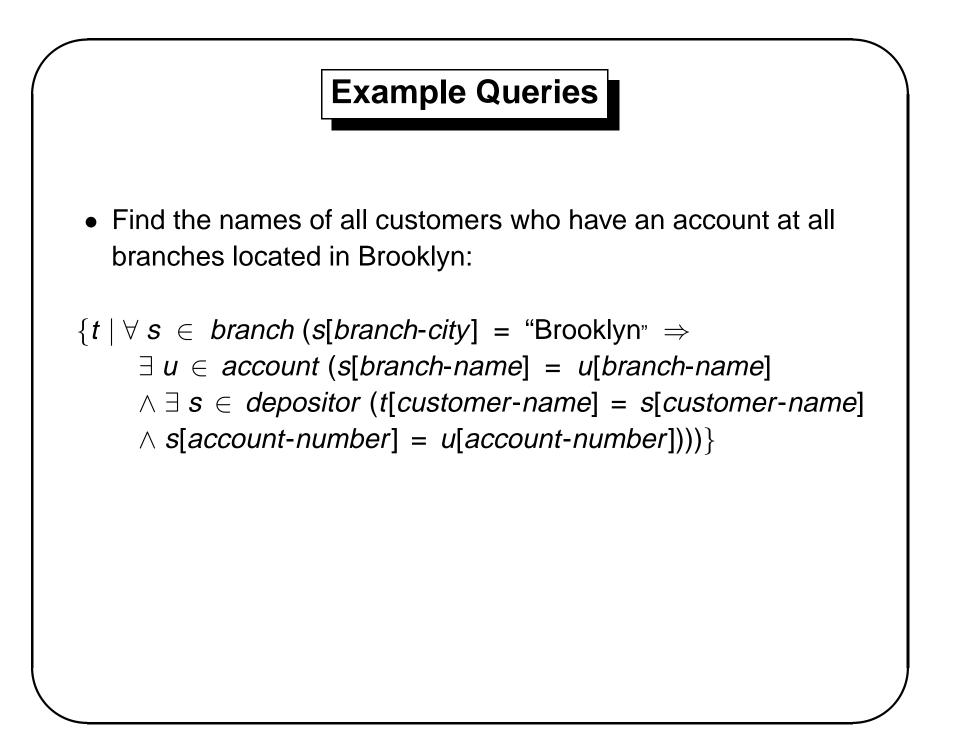
 $\{t \mid \exists s \in borrower(t[customer-name] = s[customer-name] \\ \land \exists u \in loan(u[branch-name] = "Perryridge" \\ \land u[loan-number] = s[loan-number])) \}$

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

{t | ∃s ∈ borrower(t[customer-name] = s[customer-name] ∧ ∃u ∈ loan(u[branch-name] = "Perryridge" ∧ u[loan-number] = s[loan-number]) ∧ ∄v ∈ depositor (v[customer-name] = t[customer-name]}

• Find the names of all customers having a loan from the Perryridge branch and the cities they live in

 $\begin{array}{ll} \{t \mid \exists \ s \ \in \ loan \ (s[branch-name] = "Perryridge" \\ \land \exists \ u \ \in \ borrower \ (u[loan-number] \ = \ s[loan-number] \\ \land t[customer-name] \ = \ u[customer-name] \\ \land \exists \ v \ \in \ customer \ (u[customer-name] \ = \ v[customer-name] \\ \land t[customer-city] \ = \ v[customer-city]))) \end{array}$



Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations.
- For example, $\{t \mid \neg t \in r\}$ results in an infinite relation if the domain of any attribute of relation *r* is infinite
- To guard against the problem, we restrict the set of allowable expressions to *safe* expressions.
- An expression {t | P(t)} in the tuple relational calculus is safe if every component of t appears in one of the relations, tuples, or constants that appear in P

Domain Relational Calculus

- A nonprocedural query language equivalent in power to the tuple relational calculus.
- Each query is an expression of the form:

$$\{ < x_1, x_2, ..., x_n > | P(x_1, x_2, ..., x_n) \}$$

- x_1 , x_2 , ..., x_n represent domain variables
- P represents a formula similar to that of the predicate calculus

Example Queries

• Find the *branch-name*, *loan-number*, and *amount* for loans of over \$1200:

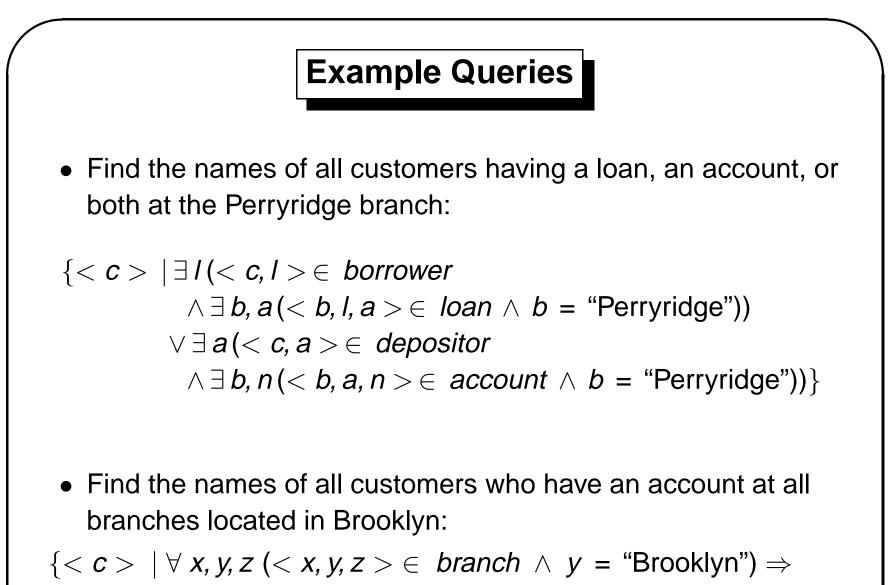
 $\{ < b, l, a > | < b, l, a > \in \text{ loan } \land a > 1200 \}$

 Find the names of all customers who have a loan of over \$1200:

 $\{ < c > | \exists b, l, a (< c, l > \in borrower \land < b, l, a > \in loan \land a > 1200) \}$

• Find the names of all customers who have a loan from the Perryridge branch and the loan amount:

 $\{ < c, a > | \exists I (< c, I > \in borrower \land \exists b (< b, I, a > \in loan \land b = "Perryridge")) \}$



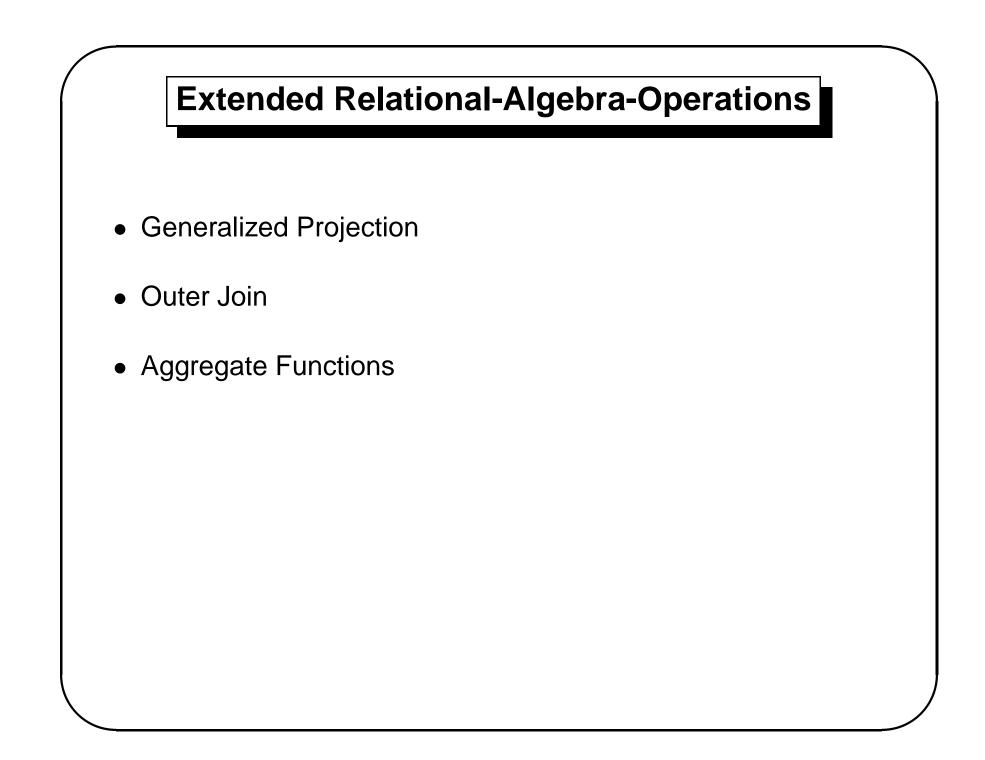
 $\exists a, b (< x, a, b > \in account \land < c, a > \in depositor) \}$

Safety of Expressions

 $\{ < x_1, x_2, ..., x_n > | P(x_1, x_2, ..., x_n) \}$

is safe if all of the following hold:

- All values that appear in tuples of the expression are values from *dom(P)* (that is, the values appear either in *P* or in a tuple of a relation mentioned in *P*).
- 2. For every "there exists" subformula of the form $\exists x (P_1(x))$, the subformula is true if and only if there is a value *x* in *dom*(*P*₁) such that *P*₁(*x*) is true.
- 3. For every "for all" subformula of the form $\forall x \ (P_1(x))$, the subformula is true if and only if $P_1(x)$ is true for all values x from $dom(P_1)$.



Generalized Projection

• Extends the projection operation by allowing arithmetic functions to be used in the projection list.

 $\Pi_{F_1,F_2,\ldots,F_n}(E)$

- *E* is any relational-algebra expression
- Each of F_1, F_2, \ldots, F_n are arithmetic expressions involving constants and attributes in the schema of *E*.
- Given relation *credit-info*(*customer-name*, *limit*, *credit-balance*), find how much more each person can spend:

 $\Pi_{customer-name, limit - credit-balance}$ (credit-info)

Outer Join

- 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 to the result of the join.
- Uses *null* values:
 - null signifies that the value is unknown or does not exist.
 - All comparisons involving *null* are **false** by definition.

Outer Join – Example

• Relation *loan*

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

• Relation *borrower*

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

● Ioan ⋈ Borrower									
	branch-r	name	loan-nu	mber	an	nount	customer-l	name]
	Downtow	vn	L-170		3	000	Jones		ĺ
	Redwoo	d	L-230		4	000	Smith		
• 10	an⊐⊠ bc	orrowe	r						
	an ⊐⊠ bc ch-name		r number	ато	unt	custo	omer-name	loan-	-numbe
brand			number	<i>ато</i> . 300		<i>custo</i> Jone		<i>loan</i> -	-numbe
brand	<i>ch-name</i> ntown	loan-l	number		0	-	S		0

Outer Join – Example ● *loan* ⋈ Borrower branch-name loan-number amount customer-name 3000 L-170 Downtown Jones Redwood L-230 4000 Smith L-155 null null Hayes

● *loan* ⊐⋈⊂ *borrower*

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

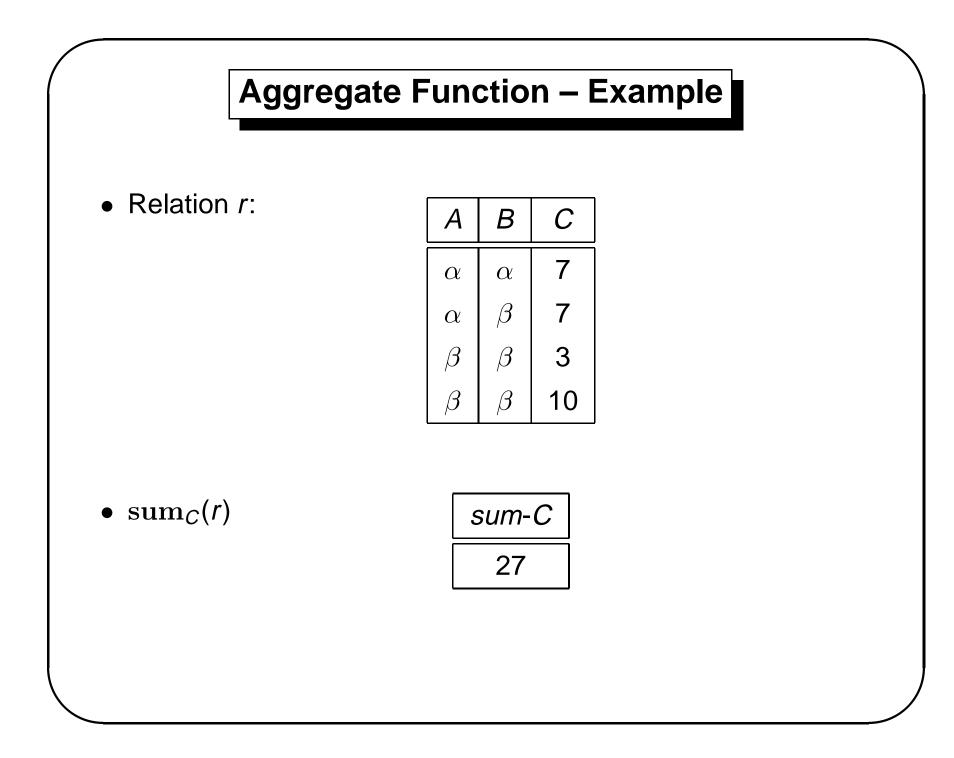
Aggregate Functions

 Aggregation operator G takes a collection of values and returns a single value as a result.

avg: average value
min: minimum value
max:maximum value
sum: sum of values
count: number of values

$$G_{1}, G_{2}, ..., G_{n} \mathcal{G}_{F_{1}} A_{1}, F_{2} A_{2}, ..., F_{m} A_{m}(E)$$

- E is any relational-algebra expression
- G_1, G_2, \ldots, G_n is a list of attributes on which to group
- F_i is an aggregate function
- $-A_i$ is an attribute name



Aggregate Function – Example

• Relation *account* grouped by *branch-name*:

branch-name	account-number	balance
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

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

branch-name	sum-balance
Perryridge	1300
Brighton	750
Redwood	700

Modification of the Database

- The content of the database may be modified using the following operations:
 - Deletion
 - Insertion
 - Updating
- All these operations are expressed using the assignment operator.

Deletion

- A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- Can delete only whole tuples; cannot delete values on only particular attributes.
- A deletion is expressed in relational algebra by:

 $r \leftarrow r - E$

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

Deletion Examples

• Delete all account records in the Perryridge branch.

```
account \leftarrow
```

```
account – \sigma_{branch-name = "Perryridge"} (account)
```

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

loan \leftarrow loan $-\sigma_{amount \geq 0}$ and $amount \leq 50$ (loan)

Delete all accounts at branches located in Needham.

 $r_1 \leftarrow \sigma_{branch-city} = "Needham" (account \bowtie branch)$ $r_2 \leftarrow \Pi_{branch-name, account-number, balance} (r_1)$ $r_3 \leftarrow \Pi_{customer-name, account-number} (r_2 \bowtie depositor)$ $account \leftarrow account - r_2$ $depositor \leftarrow depositor - r_3$

Insertion

- To insert data into a relation, we either:
 - specify a tuple to be inserted
 - write a query whose result is a set of tuples to be inserted
- 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.

Insertion Examples

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

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

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

 $r_1 \leftarrow (\sigma_{branch-name = "Perryridge"} (borrower \bowtie loan))$ account \leftarrow account $\cup \Pi_{branch-name, loan-number, 200} (r_1)$ depositor \leftarrow depositor $\cup \Pi_{customer-name, loan-number} (r_1)$

Updating

- A mechanism to change a value in a tuple without changing *all* values in the tuple
- Use the generalized projection operator to do this task

 $r \leftarrow \Pi_{F_1,F_2,\ldots,F_n}(r)$

- Each F_i is either the *i*th attribute of *r*, if the *i*th attribute is not updated, or, if the attribute is to be updated
- F_i is an expression, involving only constants and the attributes of r, which gives the new value for the attribute

Update Examples

Make interest payments by increasing all balances by 5 percent.

```
account \leftarrow \Pi_{BN,AN,BAL} \leftarrow _{BAL * 1.05} (account)
```

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

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

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

Views

- In some cases, it is not desirable for all users to see the entire logical model (i.e., all the actual relations stored in the database.)
- Consider a person who needs to know a customer's loan number but has no need to see the loan amount. This person should see a relation described, in the relational algebra, by

 $\Pi_{customer-name, loan-number}$ (borrower \bowtie loan)

• Any relation that is not part of the conceptual model but is made visible to a user as a "virtual relation" is called a *view*.

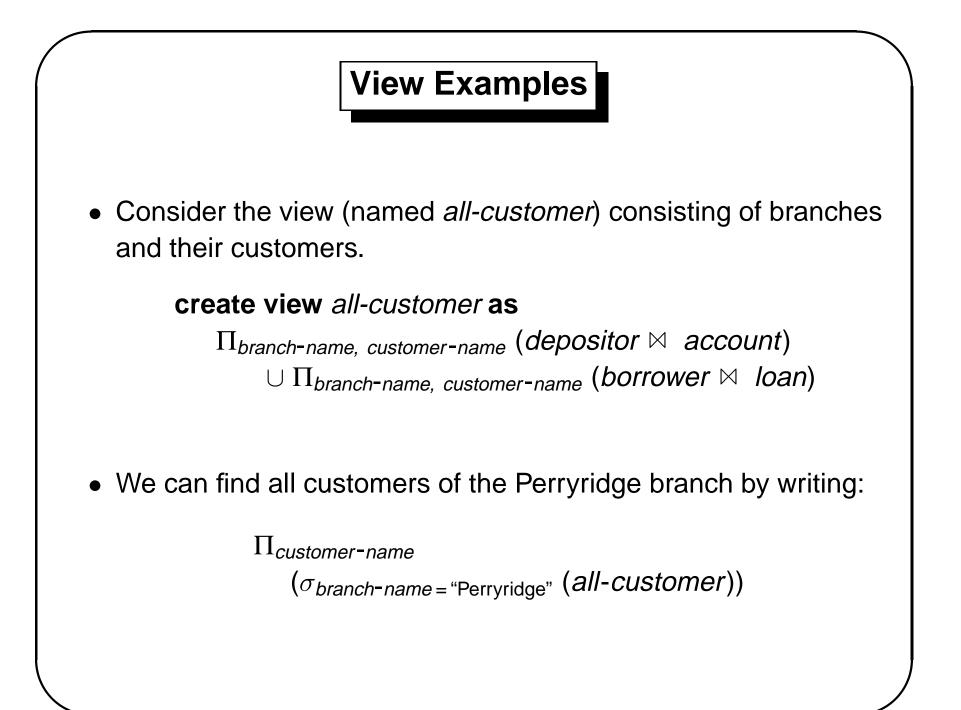
View Definition

 A view is defined using the create view statement which has the form

create view v as <query expression>

where <query expression> is any legal relational algebra query expression. The view name is represented by *v*.

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query epression. Rather, a view definition causes the saving of an expression to be substituted into queries using the view.



Updates Through Views

- Database modifications expressed as views must be translated to modifications of the actual relations in the database.
- Consider the person who needs to see all loan data in the *loan* relation except *amount*. The view given to the person, *branch-loan*, is defined as:

create view branch-loan as

 $\Pi_{branch-name, loan-number}$ (loan)

Since we allow a view name to appear wherever a relation name is allowed, the person may write:

branch-loan \leftarrow *branch-loan* \cup {("Perryridge", L-37)}

Updates Through Views (Cont.)

- The previous insertion must be represented by an insertion into the actual relation *loan* from which the view *branch-loan* is constructed.
- An insertion into *loan* requires a value for *amount*. The insertion can be dealt with by either
 - rejecting the insertion and returning an error message to the user
 - inserting a tuple ("Perryridge", L-37, *null*) into the *loan* relation

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v₁ is said to *depend directly on* a view relation v₂ if v₂ is used in the expression defining v₁
- A view relation v_1 is said to *depend on* view relation v_2 if and only if there is a path in the dependency graph from v_2 to v_1 .
- A view relation v is said to be *recursive* if it depends on itself.

View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view *v*₁ be defined by an expression *e*₁ that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

repeat

Find any view relation v_i in e_1 Replace the view relation v_i by the expression defining v_i **until** no more view relations are present in e_1

 As long as the view definitions are not recursive, this loop will terminate.