Relational Model

Chapter 3

Relational Model

- A query language is a language in which user requests to retrieve some information from the database.
- The query languages are considered as higher level languages than programming languages.
- Query languages are of two types,
  - Procedural Language
  - Non-Procedural Language

Relational Model

- In procedural language, the user has to describe the specific procedure to retrieve the information from the database.
  - *Example: The Relational Algebra*
- In non-procedural language, the user retrieves the information from the database without describing the specific procedure to retrieve it.
  - *Example: 1) The Tuple Relational Calculus*

    2) The Domain Relational Calculus

Relational Algebra

- The relational algebra is a procedural query language.
- It consists of a set of operations that take one or two relations (tables) as input and produce a new relation, on the request of the user to retrieve the specific information, as the output.
- The relational algebra contains the following operations,
  1) Selection
  2) Projection
  3) Rename

$\left.\begin{array}{l}\text{1) Selection}\\ \text{2) Projection}\end{array}\right\}$ Unary Operations

Relational Algebra

$$\left.\begin{array}{l}\text{4) Union}\\ \text{5) Set-Difference}\\ \text{6) Cartesian product}\\ \text{7) Intersection}\\ \text{8) Join}\\ \text{9) Divide}\\ \text{10) Assignment}\end{array}\right\}\ \text{Binary Operations}$$

Basic Structure

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

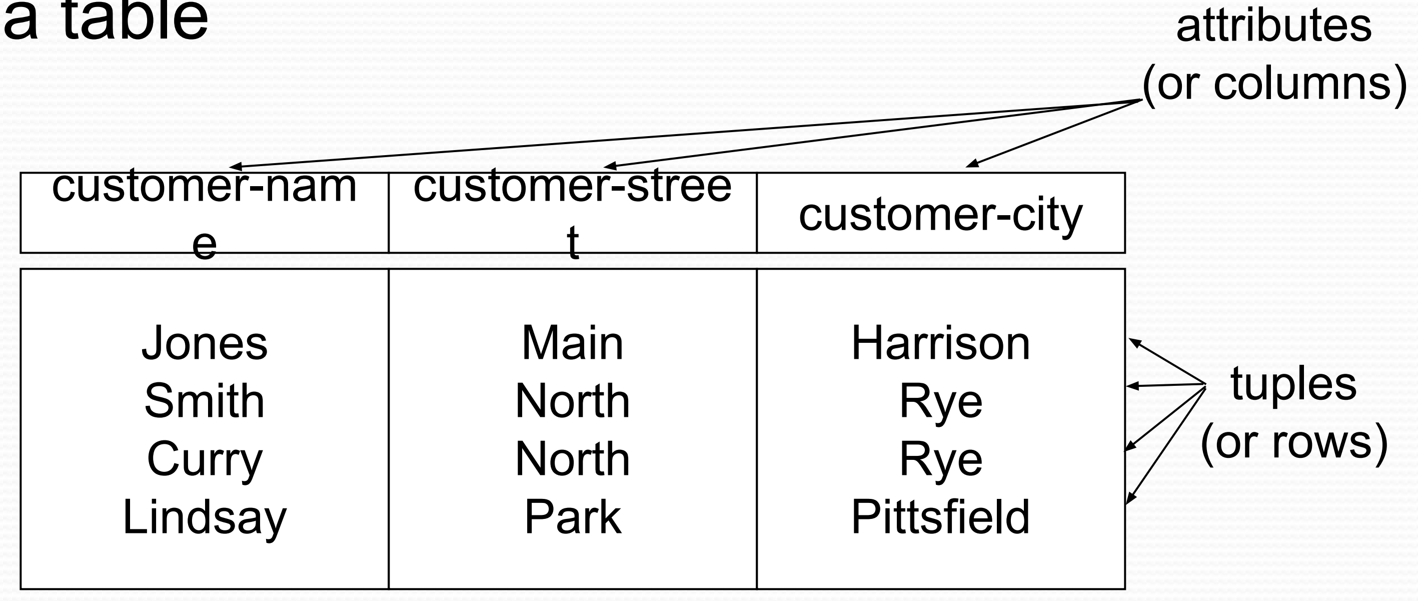
  customer-name = {Jones, Smith, Curry, Lindsay}
  customer-street = {Main, North, Park}
  customer-city = {Harrison, Rye, Pittsfield}

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

  is a relation over *customer-name* x *customer-street* x *customer-city*

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

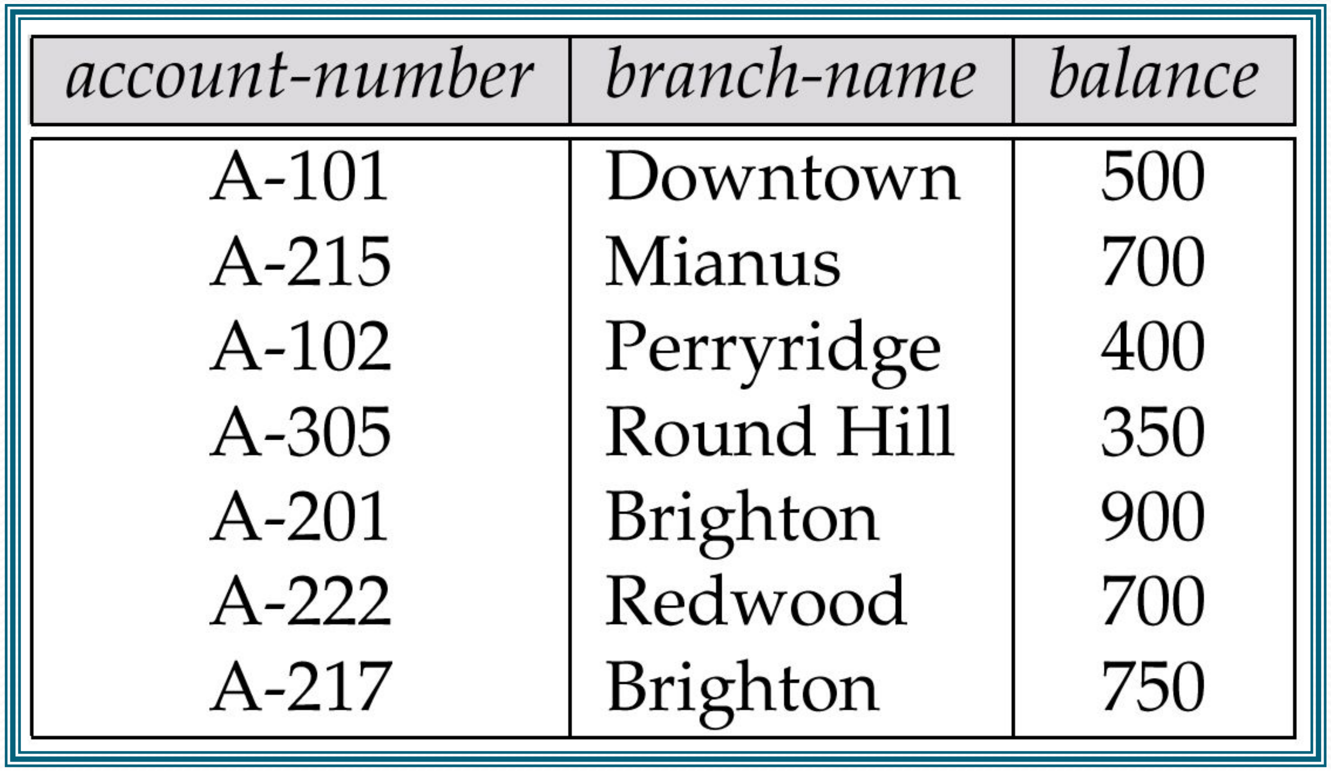


customer-name	customer-street	customer-city
Jones	Main	Harrison
Smith	North	Rye
Curry	North	Rye
Lindsay	Park	Pittsfield

customer

Relations are Unordered

- Order of tuples is irrelevant (tuples may be stored in an arbitrary order)
- E.g. account relation with unordered tuples



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

Select Operation – Example

- The Selection is a relational algebra operation that uses a condition to select rows from a relation.

  - Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

  - $\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
α	α	1	7
β	β	23	10

Select Operation

- Notation: $\sigma_p(r)$
- p is called the selection predicate
- Defined as:

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

  Where p is a formula in propositional calculus consisting of terms connected by : ∧ (**and**), ∨ (**or**), ¬ (**not**)

  Each term is one of:

  <attribute> *op* <attribute> or <constant>

  where *op* is one of: =, ≠, >, ≥. <. ≤
- Example of selection:

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

Select Operation

Book

Isbn_no	Sub	Price	Year
A1010	DBMS	450	2019
A1020	DBMS	380	2010
A1030	CJT	700	2011

- Select tuples from book where subject is DBMS
  - $\sigma_{Sub=\text{“DBMS”}}(\text{Book})$
- Selects tuples from book where subject is DBMS and 'price' is 450.
  - $\sigma_{Sub=\text{“DBMS” and } Price=\text{“450”}}(\text{Book})$
- Selects tuples from book where subject is 'database' and 'price' is 450 or those books published after 2010.
  - $\sigma_{Sub=\text{“DBMS” and } Price=\text{“450” or } Year>2010}(\text{Book})$

Project Operation – Example

- The projection is a relational algebra operation that extracts specified columns from a table.
- Relation *r*:

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

- $\Pi_{A,C}(r)$

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

Project Operation

- Notation:

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

  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
- E.g. To eliminate the *branch-name* attribute of *account*

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

Project Operation

Book

Isbn_no	Sub	Price	Year
A1010	DBMS	450	2019
A1020	DBMS	380	2010
A1030	CJT	700	2011

- Select and project Isbn no. and subject from book
  - $\Pi_{Isbn\_no,\ Sub}\ (Book)$

Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.

  Example:

  $$\rho_{X}(E)$$

  returns the expression E under the name X
- If a relational-algebra expression E has arity n, then

  $$\rho_{X\,(A1,\,A2,\,\ldots,\,An)}(E)$$

  returns the result of expression E under the name X, and with the attributes renamed to $A_1, A2, \ldots, An$.

Rename Operation

Book

Isbn_no	Sub	Price	Year
A1010	DBMS	450	2019
A1020	DBMS	380	2010
A1030	CJT	700	2011

- Rename relation book to book1
  - $\rho\ (Book1, Book)$
- Create a relation book_name with Isbn_no and Subject name
  - $\rho(Book\_name, \Pi_{(Isbn\_no,\ Sub)}(Book))$

Union Operation – Example

- Relations *r, s:*

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

r ∪ s:

A	B
α	1
α	2
β	1
β	3

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)

  2. 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)

- E.g. to find all customers with either an account or a loan

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

Union Operation

- Book(isbnno,booknm,author)
- Article(ano,title,author)
- Display name of the author who have either written book or article
  - $\Pi_{\text{author}}(\text{Book}) \cup \Pi_{\text{author}}(\text{Article})$

Set Difference Operation – Example

- Relations *r, s:*

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

r – s:

A	B
α	1
β	1

Set Difference Operation

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

  $r - s = \{t \mid t \in r$ **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

Set Difference Operation

- Book(isbnno,booknm,author)
- Article(ano,title,author)
- Display the authors who have written book but not article
  - $\prod_{\text{author}}(\text{Book}) - \prod_{\text{author}}(\text{Article})$

Cartesian-Product Operation-Example

Relations r, s:

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

r x s:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Cartesian-Product Operation

- Notation *r* x *s*
- Defined as:

  $$r \times s = \{t\,q \mid t \in r \textbf{ and } q \in s\}$$

- 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.

Cartesian-Product Operation

Book

Isbn_no	Sub	Author
A1010	DBMS	Korth
A1020	DBMS	Bayross
A1030	CJT	Schildt

Article

A_no	Title	Author
A1	Er model	Korth
A2	plsql	Bayross

- Book x Article

Isbn_no	Sub	Author	A_no	Title	Author
A1010	DBMS	Korth	A1	Er model	Korth
A1010	DBMS	Korth	A2	plsql	Bayross
A1020	DBMS	Bayross	A1	Er model	Korth
A1020	DBMS	Bayross	A2	plsql	Bayross
A1030	CJT	Schildt	A1	Er model	Korth
A1030	CJT	Schildt	A2	plsql	Bayross

Cartesian-Product Operation

- Display all the books and articles written by korth
  - $\sigma_{\text{author="Korth"}}$(Book x Article)

Composition of Operations

- Can build expressions using multiple operations
- Example: $\sigma_{A=C}(r \times s)$
- $r \times s$

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

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

A	B	C	D	E
α	1	α	10	a
β	2	β	20	a
β	2	β	20	b

Banking Example

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

customer (customer-name, customer-street, customer-only)

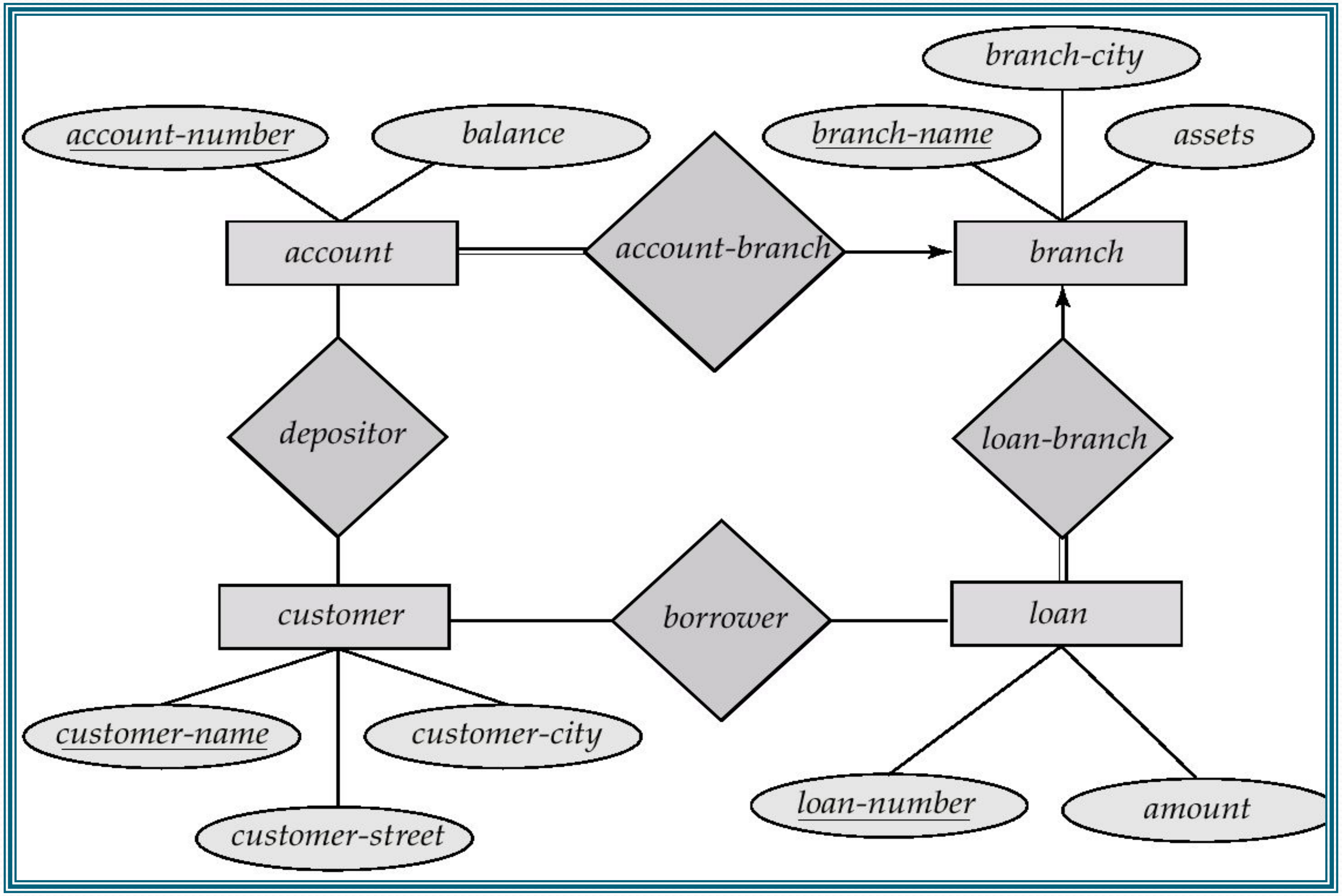
account (account-number, branch-name, balance)

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

depositor (customer-name, account-number)

borrower (customer-name, loan-number)

Sample ER-model



Example Queries

- Find all loans of over $1200

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

•Find the loan number for each loan of an amount greater than $1200

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

Example Queries

- Find the names of all customers who have a loan, an account, or both, from the bank

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

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

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

- Find the names of all customers who have a loan but do not have an account at bank.

$$\Pi_{customer\text{-}name}(borrower) - \Pi_{customer\text{-}name}(depositor)$$

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.
- Q1.

$$\Pi_{customer\text{-}name}\,(\sigma_{branch\text{-}name=\text{“}Perryridge\text{”}}\,(\sigma_{borrower.loan\text{-}number\,=\,loan.loan\text{-}number}(borrower \times loan)))$$

- Q2.

$$\Pi_{\text{customer-name}}(\sigma_{\text{loan.loan-number = borrower.loan-number}}((\sigma_{\text{branch-name = “Perryridge”}}(\text{loan})) \times \text{borrower}))$$

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

$$\Pi_{customer\text{-}name}\,(\sigma_{branch\text{-}name\,=\,\text{“}Perryridge\text{”}}\,(\sigma_{borrower.loan\text{-}number\,=\,loan.loan\text{-}number}(\text{borrower x loan}))) \;-\; \Pi_{customer\text{-}name}(\text{depositor})$$

Example Queries

Find the largest account balance

- Rename *account* relation as *d*
- The query is:

$$\Pi_{balance}(account) - \Pi_{account.balance}$$

$$(\sigma_{account.balance < d.balance}(account \; x \; \rho_{d}(account)))$$

Formal Definition

- A basic expression in the relational algebra consists of either one of the following:
  - A relation in the database
  - A constant relation
- Let E_1 and E_2 be relational-algebra expressions; the following are all relational-algebra expressions:
  - $E_1 \cup E_2$
  - $E_1 - E_2$
  - $E_1 \times E_2$
  - $\sigma_p (E_1)$, P is a predicate on attributes in E_1
  - $\prod_s(E_1)$, S is a list consisting of some of the attributes in E_1
  - $\rho_x (E_1)$, x is the new name for the result of E_1

Additional Operations

- We define additional operations that do not add any power to the relational algebra, but that simplify common queries.
- Set intersection
- Natural join
- Division
- Assignment

Set-Intersection Operation

- Notation: $r \cap s$
- Defined as:
- $r \cap s = \{ t \mid t \in r$ **and** $t \in s \}$
- Assume:
  - *r*, *s* have the *same arity*
  - attributes of r and s are compatible
- Note: $r \cap s = r - (r - s)$

Set-Intersection Operation - Example

- Relation r, s:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- r ∩ s

A	B
α	2

Natural-Join Operation

- Notation: r $\bowtie$ s
- Let *r* and *s* be relations on schemas *R* and *S* respectively.
  Then, r $\bowtie$ s is a relation on schema *R* ∪ *S* obtained as follows:
  - Consider 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* ∩ *S*, add a tuple *t* to the result, where
    - *t* has the same value as t_r on *r*
    - *t* has the same value as t_s on *s*

Natural-Join Operation

- Example:

  $R = (A, B, C, D)$

  $S = (E, B, D)$

  - Result schema = (A, B, C, D, E)
  - $r \bowtie s$ is defined as:

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

Natural Join Operation – Example

- Relations r, s:

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	$\in$

s

r $\bowtie$ s

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Natural Join

CID	Course	Dept
CS01	DBMS	IT
ME01	MOS	ME
CL01	SE	CL

Cours
e

Dept	Head
IT	VD
ME	GDB
CL	MKS

HOD

Course $\bowtie$ HOD

CID	Course	Dept	Head
CS01	DBMS	IT	VD
ME01	MOS	ME	GDB
CL01	SE	CL	MKS

Division Operation

r □ s

- Suited to queries that include the phrase "for all".
- Let *r* and *s* be relations on schemas R and S respectively where
  - $R = (A_1, \ldots, A_m, B_1, \ldots, B_n)$
  - $S = (B_1, \ldots, B_n)$

  The result of r ÷ s is a relation on schema

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

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

Division Operation – Example

Relations r, s:

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
$\in$	6
$\in$	1
β	2

r

B
1
2

s

r ÷ s:

A
α
β

Another Division Example

Relations r, s:

A	B	C	D	E
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

r

D	E
a	1
b	1

s

r ÷ s:

A	B	C
α	a	γ
γ	a	γ

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 $S \subseteq R$

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

  To see why
  - $\prod_{R-S,S}(r)$ simply reorders attributes of r
  - $\prod_{R-S}(\prod_{R-S} (r) \times s) – \prod_{R-S,S}(r))$ gives those tuples t in

    $\prod_{R-S} (r)$ 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 a result of the query.
  - Assignment must always be made to a temporary relation variable.
- Example: Write $r \div s$ as

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

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

$$result = temp1 - temp2$$

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

Branch Table

branch-name	*branch-city*	*assets*
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

Customer Table

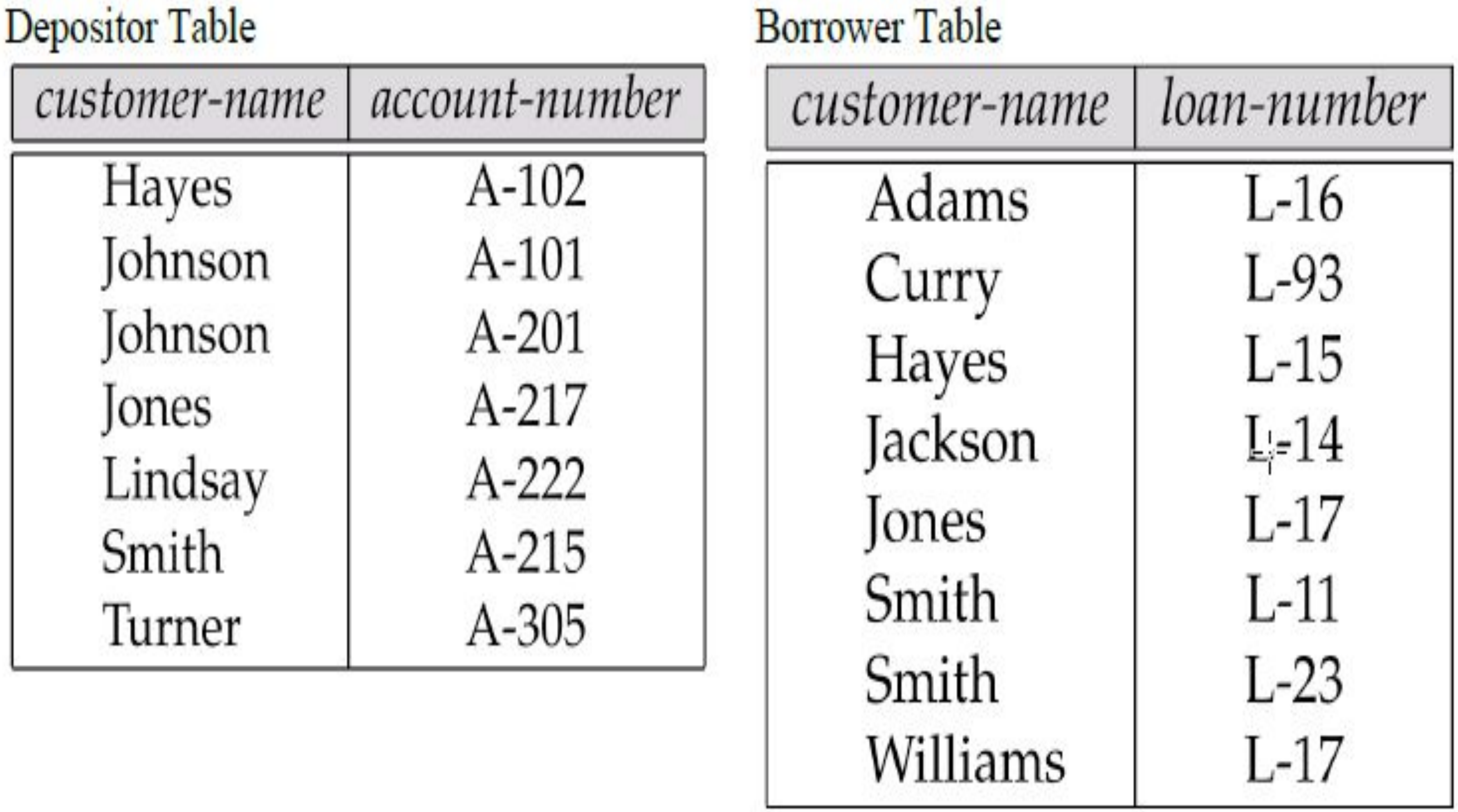
customer-name	*customer-street*	*customer-city*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

Account Table

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

Loan Table

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500



Depositor Table

customer-name	*account-number*
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Borrower Table

customer-name	*loan-number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Example Queries

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

Query 1

$$\Pi_{CN}(\sigma_{BN=\text{“Downtown”}}(depositor \bowtie account)) \cap$$

$$\Pi_{CN}(\sigma_{BN=\text{“Uptown”}}(depositor \bowtie account))$$

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

Query 2

$$\Pi_{customer\text{-}name,\ branch\text{-}name}(depositor \bowtie account)$$

$$\div \rho_{temp(branch\text{-}name)}(\{(\text{“Downtown”}), (\text{“Uptown”})\})$$

Example Queries

- Find all customers who have an account at all branches located in Brooklyn city.
- $\Pi_{customer\text{-}name,\ branch\text{-}name}(depositor \bowtie account) \div \Pi_{branch\text{-}name}(\sigma_{branch\text{-}city = \text{“Brooklyn”}}(branch))$

Extended Relational-Algebra-Operations

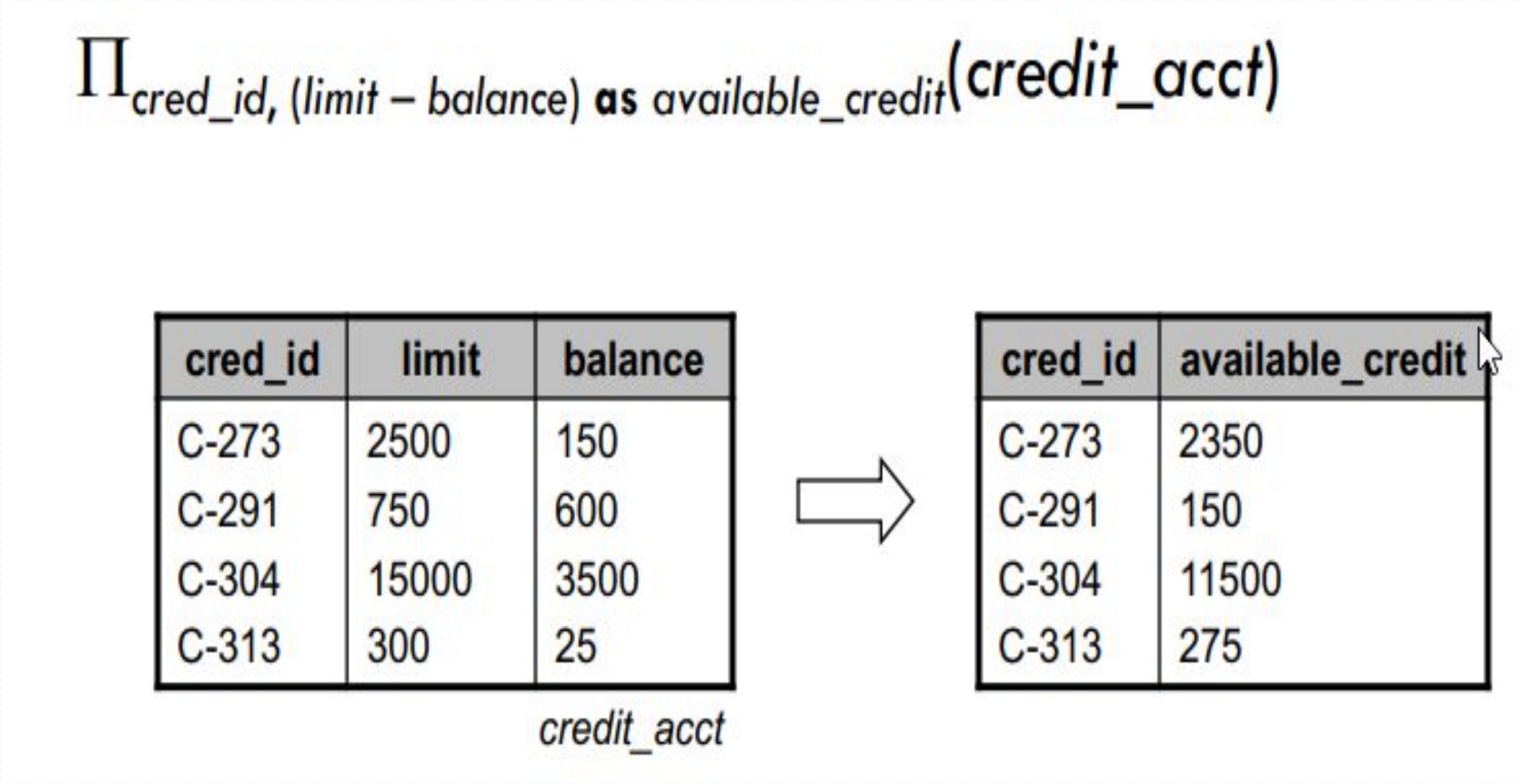
- Generalized Projection
- Outer Join
- Aggregate Functions

Generalized Projection

- Extends the projection operation by allowing arithmetic functions to be used in the projection list. $\prod_{F1, F2, \ldots, Fn}(E)$
- *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:

$$\prod_{\textit{customer-name, limit – credit-balance}} (\textit{credit-info})$$

Generalized Projection



$$\Pi_{\mathit{cred\_id},\ (\mathit{limit} - \mathit{balance})\ \textbf{as}\ \mathit{available\_credit}}(\mathit{credit\_acct})$$

cred_id	limit	balance
C-273	2500	150
C-291	750	600
C-304	15000	3500
C-313	300	25

credit_acct

$\Rightarrow$

cred_id	available_credit
C-273	2350
C-291	150
C-304	11500
C-313	275

Aggregate Functions and Operations

- **Aggregation function** 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

- **Aggregate operation** in relational algebra

$$_{G1,\ G2,\ \ldots,\ Gn}\ g\ _{F1(\ A1),\ F2(\ A2),\ldots,\ Fn(\ An)}\ (E)$$

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

Aggregate Operation – Example

- Relation *r*:

A	B	C
α	α	7
α	β	7
β	β	3
β	β	10

$g_{\mathbf{sum(c)}}(r)$

sum-C
27

Aggregate Operation – 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

$${}_{\text{branch-name}}\,g\,{}_{\text{sum(balance)}}\,(\text{account})$$

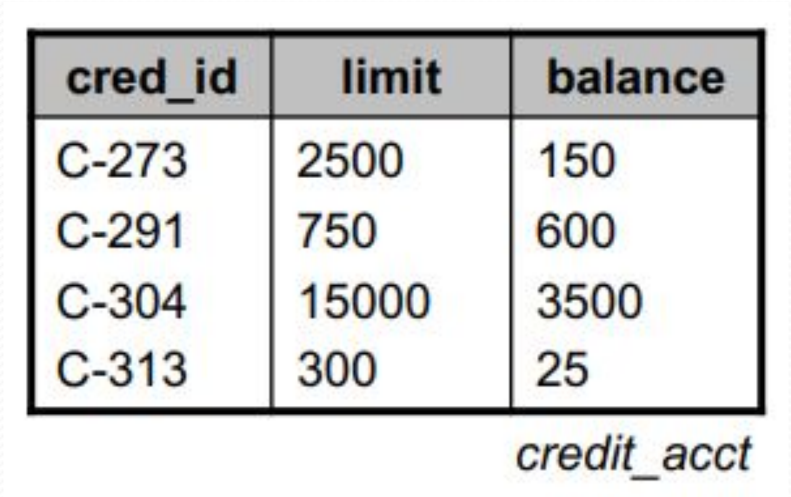
branch-name	balance
Perryridge	1300
Brighton	1500
Redwood	700

Aggregate Functions

- Result of aggregation does not have a name
  - Can use rename operation to give it a name
  - For convenience, we permit renaming as part of aggregate operation

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

Aggregate Functions



cred_id	limit	balance
C-273	2500	150
C-291	750	600
C-304	15000	3500
C-313	300	25

credit_acct

- **Find total amount owed by credit company**

$$g_{\textbf{sum}(\text{balance})}\,(\text{Credit\_acct})$$

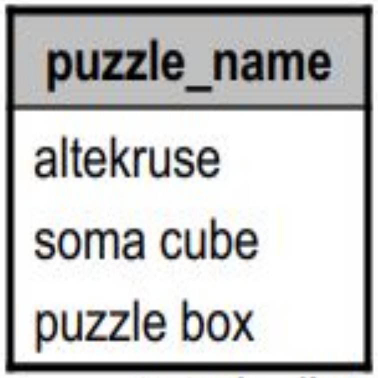
427
5

- **Find the maximum available credit of any account**

$$g_{\textbf{max}(\text{available\_credit})}\,(\,\Pi_{\textit{(limit - balance)}\ \textbf{\textit{as}}\ \textit{available\_credit}}\,(\textit{credit\_acct}))$$

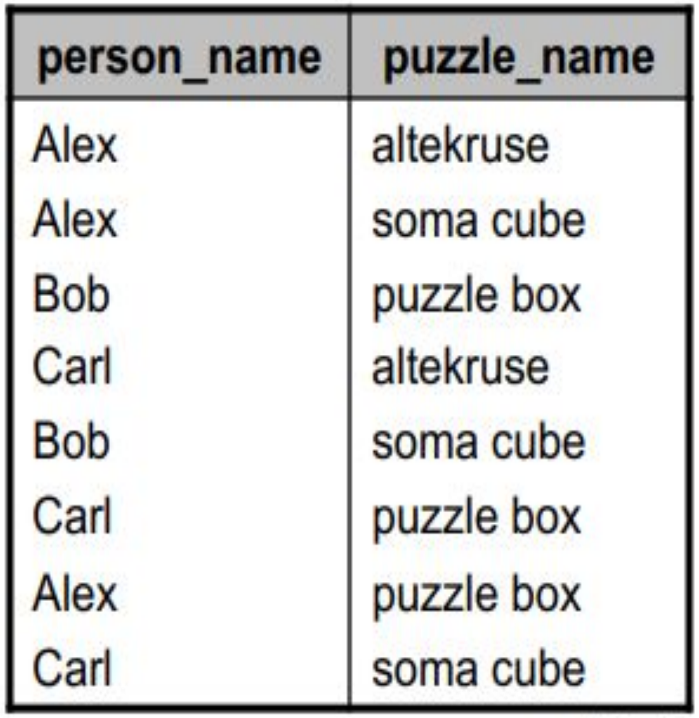
1150
0

Aggregate Functions



puzzle_name
altekruse
soma cube
puzzle box

puzzle_list

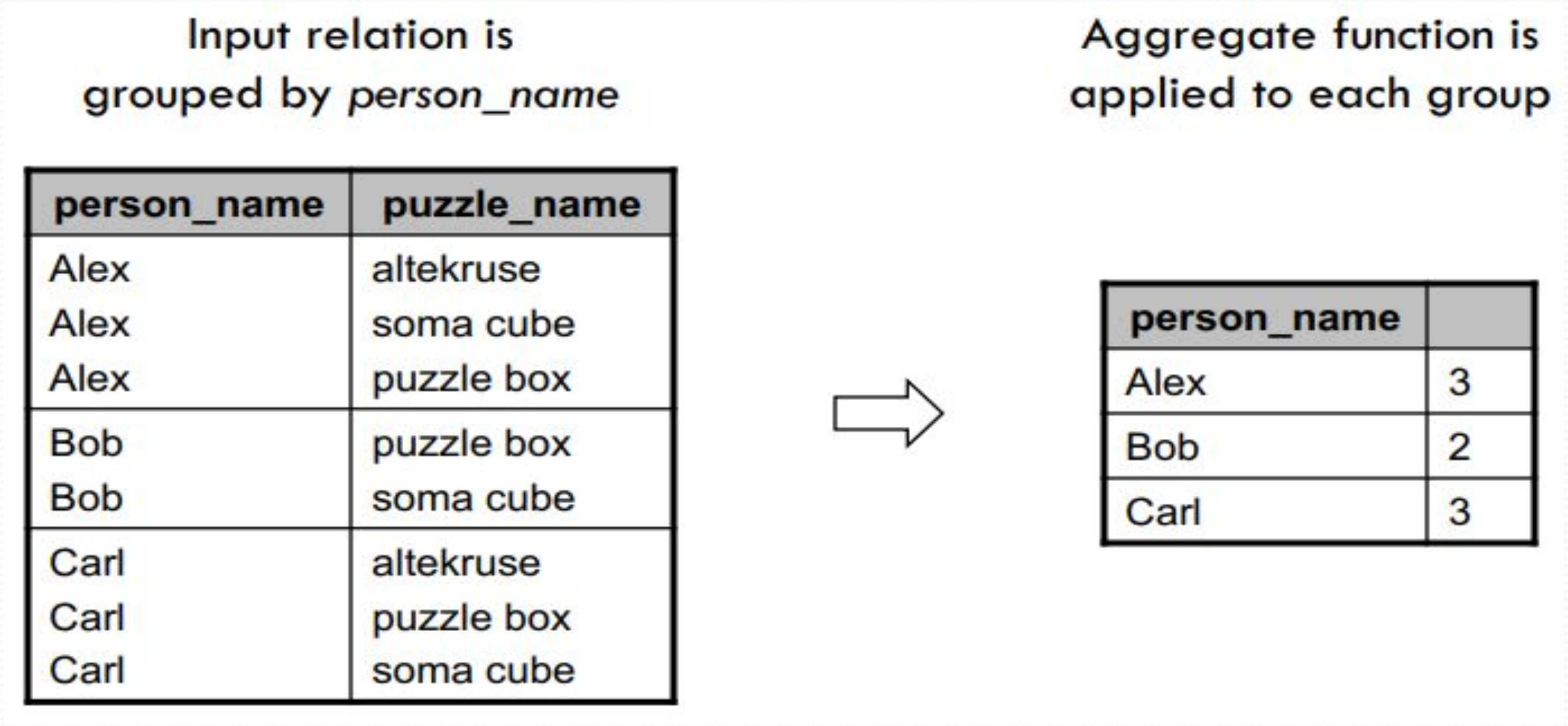


person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Bob	puzzle box
Carl	altekruse
Bob	soma cube
Carl	puzzle box
Alex	puzzle box
Carl	soma cube

completed

- **How many puzzles has each person completed**

Aggregate Functions



Input relation is grouped by *person_name*

person_name	puzzle_name
Alex	altekruse
Alex	soma cube
Alex	puzzle box
Bob	puzzle box
Bob	soma cube
Carl	altekruse
Carl	puzzle box
Carl	soma cube

Aggregate function is applied to each group

person_name	
Alex	3
Bob	2
Carl	3

$${}_{\text{person-name}}\,g\,{}_{\textbf{count}(\text{puzzle\_name})}\,(\text{completed})$$

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form 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 (roughly speaking) **false** by definition.

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 L-230	Downtown Redwood	3000 4000	Jones Smith

- **Left Outer Join**

  loan ⟕ Borrower

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

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null.*
- Aggregate functions simply ignore null values
  - Is an arbitrary decision. Could have returned null as result instead.
  - We follow the semantics of SQL in its handling of null values
- For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same
  - Alternative: assume each null is different from each other
  - Both are arbitrary decisions, so we simply follow SQL

Null Values

- Comparisons with null values return the special truth value *unknown*
  - If *false* was used instead of *unknown*, then *not (A < 5)* would not be equivalent to *A >= 5*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: *(true* **and** *unknown)* = *unknown,*
    (false **and** *unknown)* = *false,*
    (unknown **and** *unknown)* = *unknown*
  - NOT*:* (**not** *unknown) = unknown*
  - In SQL "*P* **is unknown"** evaluates to true if predicate *P* evaluates to *unknown*
- Result of select predicate is treated as *false* if it evaluates to *unknown*

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

Book

Isbn_no	**Sub**	**Author**
A1010	DBMS	Korth
A1020	DBMS	Bayross
A1030	CJT	Schildt

- Delete all book records with isbn no. A1020
- $Book \leftarrow Book - \sigma_{Isbn\_no\ =\ \text{“}A1020\text{”}}(Book)$

Isbn_no	**Sub**	**Author**
A1010	DBMS	Korth
A1030	CJT	Schildt

Branch Table

branch-name	*branch-city*	*assets*
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

Customer Table

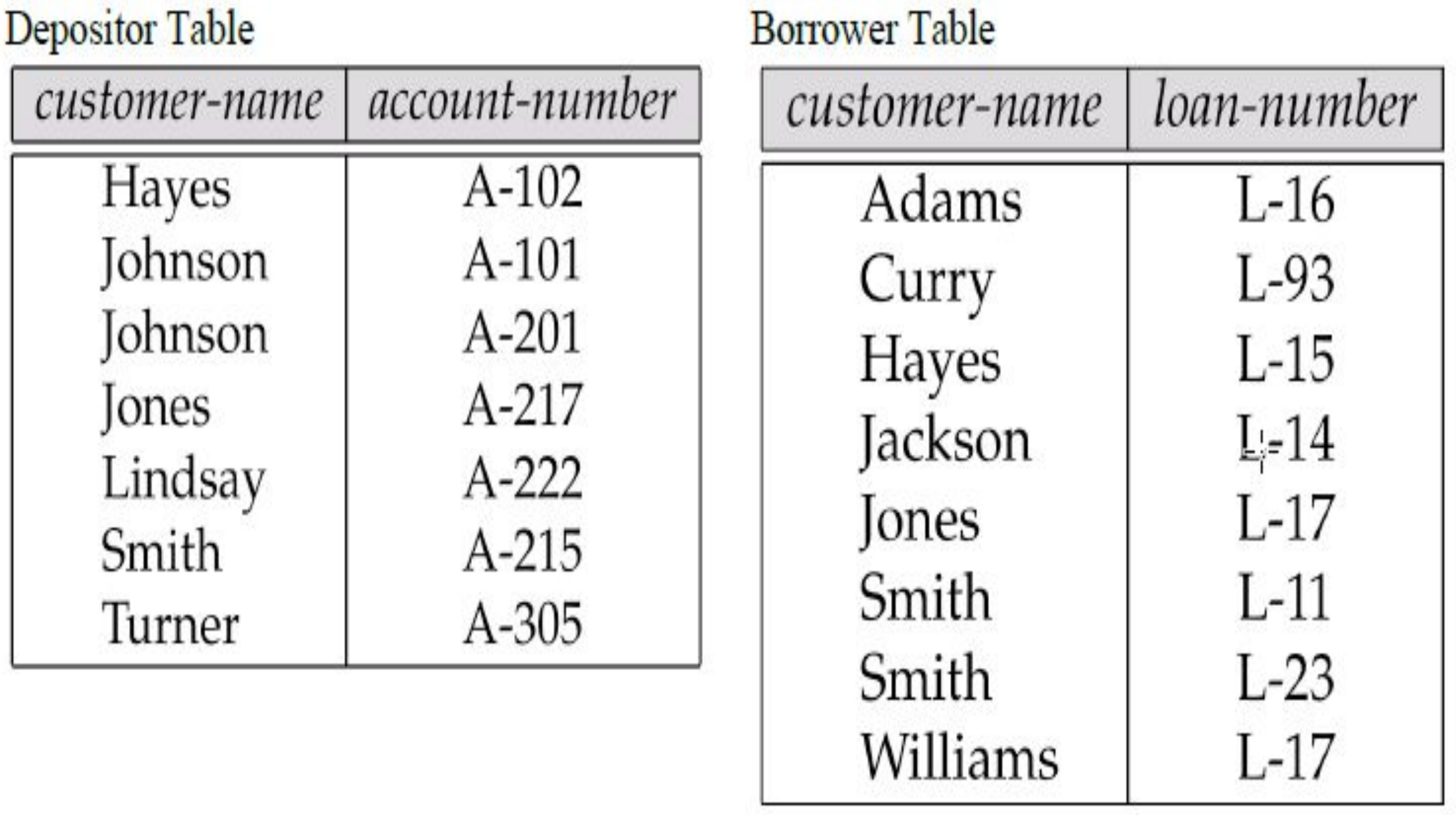
customer-name	*customer-street*	*customer-city*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

Account Table

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

Loan Table

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500



Depositor Table

customer-name	*account-number*
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Borrower Table

customer-name	*loan-number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Deletion Examples

- Delete all account records in the Perryridge branch.

$$account \leftarrow account - \sigma_{branch\text{-}name = \text{“Perryridge”}}(account)$$

- Delete all loan records with amount in the range of 0 to 1000

$$loan \leftarrow loan - \sigma_{amount \geq 0 \text{ and } amount \leq 1000}(loan)$$

- Delete all accounts at branches located in Brooklyn.

Deletion Examples

- $\sigma_{\textit{branch-city} = \text{"Brooklyn"}} (\textit{account} \bowtie \textit{branch})$

Acount_number	Branch_name	balance	Branch-city	assets
A-201	Brighton	900	Brooklyn	7100000
A-217	Brighton	750	Brooklyn	7100000
A-101	Downtown	500	Brooklyn	9000000

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

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.

$$account \leftarrow account \cup \{(\text{“Perryridge”}, \text{A-973}, 1200)\}$$

$$\text{depositor} \leftarrow depositor \cup \{(\text{“Smith”}, \text{A-973})\}$$

- 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.

$$r_1 \leftarrow (\sigma_{branch\text{-}name\ =\ \text{“Perryridge”}}(borrower \bowtie loan))$$

$$account \leftarrow account \cup \prod\nolimits_{branch\text{-}name,\ account\text{-}number, 200}(r_1)$$

$$\text{depositor} \leftarrow depositor \cup \prod\nolimits_{customer\text{-}name,\ loan\text{-}number}(r_1)$$

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

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

- 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.

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

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

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

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

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

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

- Any relation that is not 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 expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

View Definition

Table

Student

Stu_id	Name	Sem	Mo_no	Grade
IT1010	ABC	5	123456789	A
IT1020	DEF	5	345678912	A
IT030	XYZ	5	678912345	B

View

Select * from student

Student View

Stu_id	Name	Sem	Mo_no	Grade
IT1010	ABC	5	123456789	A
IT1020	DEF	5	345678912	A
IT030	XYZ	5	678912345	B

View Examples

- Consider the view (named *all-customer*) consisting of account branches, loan branches and their customers.

create view *all-customer* **as**

$$\Pi_{branch\text{-}name,\ customer\text{-}name}\ (depositor \bowtie account)$$

$$\cup\ \Pi_{branch\text{-}name,\ customer\text{-}name}\ (borrower \bowtie loan)$$

●Find all customers of the Perryridge branch by writing:

$$\Pi_{customer\text{-}name}$$

$$(\sigma_{branch\text{-}name = \text{“Perryridge”}}\ (all\text{-}customer))$$

Updates Through View

- 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**

  $\Pi_{\textit{branch-name, loan-number}}\ (\textit{loan})$
- Since we allow a view name to appear wherever a relation name is allowed, the person may write:

  $\textit{branch-loan} \leftarrow \textit{branch-loan} \cup \{(\text{“Perryridge”}, \text{L-37})\}$

Updates Through Views

- 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 ("L-37", "Perryridge", *null*) into the *loan* relation
- Some updates through views are impossible to translate into database relation updates
  - create view v as $\sigma_{\textit{branch-name} = \text{"Perryridge"}}$ (*account*))

    $v \leftarrow v \cup$ (L-99, Downtown, 23)
- Others cannot be translated uniquely
  - *all-customer* $\leftarrow$ *all-customer* $\cup$ {("Perryridge", "John")}
    - Have to choose loan or account, and create a new loan/account number!

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1

  create view **v2** as $\prod_{account\_number, branch\text{-}name, balance}(account)$

  create view v1 as $\sigma_{account\_number, branch\text{-}name = \text{“Perryridge”}}(\boldsymbol{v2}))$
- A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2

  create view v1 as $\sigma_{account\_number, branch\text{-}name = \text{“Perryridge”}}(\boldsymbol{v2 \times loan})$
- 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

Tuple Relational Calculus & Domain Relational Calculus

Tuple Relational Calculus

- A formal query language in which we specify a declarative expression to describe a retrieval request.
- We don’t specify how the query is to be evaluated.
- We only specify what is to be retrieved.
- Tuple Relational Calculus is a *nonprocedural* language.
- Relational Algebra is a procedural way of writing the query.

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 ($\wedge$), or ($\vee$), not ($\neg$)
4. Implication ($\Rightarrow$): $x \Rightarrow y$, if x is true, then y is true

   $$x \Rightarrow y \equiv \neg x \vee y$$

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*

Tuple Relational Calculus

Example: **Find all employees whose salary is more than 50K**

Answer:

{ t | EMPLOYEE(t) and t.SALARY > 50K }

or

{ t | EMPLOYEE(t) and t[SALARY] > 50K }

Or

{ t | t ϵ EMPLOYEE and t[SALARY] > 50K }

Or

{ t | ∃ s ϵ EMPLOYEE (t[EMP_NAME]=s[EMP_NAME] and

t[SALARY]=s[SALARY] and s[SALARY] > 50K) }

Banking Example

- *branch (branch-name, branch-city, assets)*
- *customer (customer-name, customer-street, customer-city)*
- *account (account-number, branch-name, balance)*
- *loan (loan-number, branch-name, amount)*
- *depositor (customer-name, account-number)*
- *borrower (customer-name, loan-number)*

Example Queries

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

$$\{t \mid t \in loan \land t\,[amount] > 1200\}$$

- Find the loan number for each loan of an amount greater than $1200

$$\{t \mid \exists\, s \in \text{loan}\ (t[loan\text{-}number] = s[loan\text{-}number] \land s\,[amount] > 1200)\}$$

Notice that a relation on schema [loan-number] is implicitly defined by the query

Example Queries

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

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

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

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

Example Queries

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

$$
\begin{aligned}
\{t \mid \exists s \in & \ \textit{borrower}(t[\textit{customer-name}] = s[\textit{customer-name}] \\
& \land \exists u \in \textit{loan}(u[\textit{branch-name}] = \text{“Perryridge”} \\
& \qquad \land \ u[\textit{loan-number}] = s[\textit{loan-number}]))\}
\end{aligned}
$$

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

$$
\begin{aligned}
\{t \mid \exists s \in & \ \textit{borrower}(\ t[\textit{customer-name}] = s[\text{customer-name}] \\
& \land \exists u \in \textit{loan}(u[\textit{branch-name}] = \text{“Perryridge”} \\
& \qquad \land \ u[\textit{loan-number}] = s[\text{loan-}\textit{number}])) \\
& \land \ \textbf{not} \ \exists v \in \textit{depositor}\ (v[\textit{customer-name}] = t[\text{customer-name}])\ \}
\end{aligned}
$$

Example Queries

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

$$\{t \mid \exists s \in loan(s[branch\text{-}name] = \text{“Perryridge”}$$
$$\wedge\ \exists u \in borrower\ (u[loan\text{-}number] = s[loan\text{-}number]$$
$$\wedge\ t\ [customer\text{-}name] = u[customer\text{-}name])$$
$$\wedge\ \exists\ v \in customer\ (u[customer\text{-}name] = v[customer\text{-}name]$$
$$\wedge\ t[customer\text{-}city] = v[customer\text{-}city])))\}$$

Example Queries

- Find the names of all customers who have an account at all branches located in Brooklyn:

$$\{t \mid \exists\ c \in \text{customer}\ (t[\text{customer-name}] = c[\text{customer-name}])\ \wedge$$

$$\forall\ s \in \mathit{branch}(s[\mathit{branch\text{-}city}] = \text{“Brooklyn”} \Rightarrow$$

$$\exists\ u \in \mathit{account}\ (\ s[\mathit{branch\text{-}name}] = u[\text{branch-name}]$$

$$\wedge\ \exists\ s \in \mathit{depositor}\ (\ \ t[\mathit{customer\text{-}name}] = s[\text{customer-name}]$$

$$\wedge\ \ s[\mathit{account\text{-}number}] = u[\text{account-number}]\ ))\ )\}$$

Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations.
- For example, {t | ¬ *t* ∈ *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*
  - NOTE: this is more than just a syntax condition.
    - E.g. { *t* | *t*[*A*]=5 ∨ **true** } is not safe --- it defines an infinite set with attribute values that do not appear in any relation or tuples or constants 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:

$$\{ < x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

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

Example Queries

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

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

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

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

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

$$\{< c, a > \mid \exists\ l\ (< c, l > \in borrower \land \exists b(< l, b, a > \in loan \land b = \text{“Perryridge”}))\}$$

or $\{< c, a > \mid \exists\ l\ (< c, l > \in borrower \land < l, \text{“Perryridge”}, a > \in loan)\}$

Example Queries

- Find the names of all customers having a loan, an account, or both at the Perryridge branch:

$$
\begin{aligned}
\{< c > \mid \exists\, l\, (\{< c, l > \in \mathit{borrower} \\
\quad \wedge\ \exists\, b,a(< l, b, a > \in \mathit{loan} \wedge b = \text{“Perryridge”})) \\
\vee\ \exists\, a(< c, a > \in \mathit{depositor} \\
\quad \wedge\ \exists\, b,n(< a, b, n > \in \mathit{account} \wedge b = \text{“Perryridge”}))\}
\end{aligned}
$$

- Find the names of all customers who have an account at all branches located in Brooklyn:

$$
\begin{aligned}
\{< c > \mid \exists\ \mathrm{s}, n\ (< c, s, n > \in \text{customer}) \wedge \\
\quad \forall\ x,y,z(< x, y, z > \in \mathit{branch} \wedge y = \text{“Brooklyn”}) \Rightarrow \\
\quad \exists\ a,b(< x, y, z > \in \mathit{account} \wedge < c,a > \in \mathit{depositor})\}
\end{aligned}
$$

Safety of Expressions

$$\{ < x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

is safe if all of the following hold:

1. 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 of 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).

End of Chapter 3