Relational Query Languages: Relational Algebra

Juliana Freire

Some slides adapted from J. Ullman, L. Delcambre, R. Ramakrishnan, G. Lindstrom and Silberschatz, Korth and Sudarshan

Relational Query Languages

- ***Query languages:*** Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Simple data structure – sets!
    - Easy to understand, easy to manipulate
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages **!=** programming languages!
  - QLs not expected to be "Turing complete".
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.

Relational Query Languages

- *Query languages:* Allow manipulation and retrieval of data from a database.
- Relational model supports simple,
  - Strong formal foundation based on lo
  - Allows for much optimization.

Some operations cannot be expressed

- Query Languages **!=** programming languages!
  - QLs not expected to be "Turing complete".
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.

Formal Relational Query Languages

- Two mathematical Query Languages form the basis for "real" relational languages (e.g., SQL), and for implementation:
  - *Relational Algebra*: More operational, very useful for representing execution plans.
  - *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative*.)

Describing a Relational Database Mathematically: Relational Algebra

- We can describe tables in a relational database as sets of tuples
- We can describe query operators using set theory
- The query language is called relational algebra
- Normally, not used directly -- foundation for SQL and query processing
  - SQL adds syntactic sugar

What is an “Algebra”

- Mathematical system consisting of:
  - *Operands* --- variables or values from which new values can be constructed
  - *Operators* --- symbols denoting procedures that construct new values from given values
- Expressions can be constructed by applying operators to atomic operands and/or other expressions
  - Operations can be **composed** -- algebra is closed
  - Parentheses are needed to group operators

Basics of Relational Algebra

- Algebra of arithmetic: operands are variables and constants, and operators are the usual arithmetic operators
  - E.g., (x+y)*2 or ((x+7)/(y-3)) + x

- Relational algebra: operands are variables that stand for relations and relations (sets of tuples), and operators are designed to do the most common things we need to do with relations in databases, e.g., *union, intersection, selection, projection, Cartesian product, etc*
  - E.g., ($\pi_{c\text{-}owner}$Checking-account) $\cap$ ($\pi_{s\text{-}owner}$Savings-account)
  - The result is an algebra that can be used as a *query language* for relations.

Basics of Relational Algebra (cont.)

- A query is applied to *relation instances*, and the *result of a query is also a relation instance*
  - *Schemas* of input relations for a query are fixed (but query will run regardless of instance!)
  - The schema for the *result* of a given query is also fixed. Determined by definition of query language constructs.
- Operators refer to relation attributes by position or name:
  - E.g., Account(number, owner, balance, type)

  Positional←Account.$1 = Account.number → Named field

  Positional←Account.$3 = Account.balance → Named field

  - Positional notation easier for formal definitions, named-field notation more readable.
  - Both used in SQL

Relational Algebra: Operations

- The usual set operations: union, intersection, difference
  - Both operands must have the same relation schema
- Operations that remove parts of relations:
  - Selection: pick certain rows
  - Projection: pick certain columns
- Operations that combine tuples from two relations: Cartesian product, join
- Renaming of relations and attributes
- Since each operation returns a relation, operations can be *composed*! (Algebra is “closed”.)

Removing Parts of Relations

σ (sigma)

Selection: Example

σ_c R= select -- produces a new relation with the subset of the tuples in R that match the condition *C*

Sample query: $\sigma_{\text{Type = "savings"}}$ Account

Account

Number	Owner	Balance	Type
~~101~~	~~J. Smith~~	~~1000.00~~	~~checking~~
~~102~~	~~W. Wei~~	~~2000.00~~	~~checking~~
103	J. Smith	5000.00	savings
~~104~~	~~M. Jones~~	~~1000.00~~	~~checking~~
~~105~~	~~H. Martin~~	~~10,000.00~~	~~checking~~

Number	Owner	Balance	Type
103	J. Smith	5000.00	savings

Selection: Another Example

$\sigma_{\textbf{Balance < 4000}}$ Account

Selects rows that satisfy *selection condition*

Account	Number	Owner	Balance	Type
	101	J. Smith	1000.00	checking
	102	W. Wei	2000.00	checking
	~~103~~	~~J. Smith~~	~~5000.00~~	~~savings~~
	104	M. Jones	1000.00	checking
	~~105~~	~~H. Martin~~	~~10,000.00~~	~~checking~~

	Number	Owner	Balance	Type
	101	J. Smith	1000.00	checking
	102	W. Wei	2000.00	checking
	104	M. Jones	1000.00	checking

Schema of result identical to schema of input relation

Projection: Example

π (pi)

$\pi_{\text{AttributeList}}$ R = project -- deletes attributes that are not in *projection list*.

Sample query: $\pi_{\text{Number, Owner, Type}}$ Account

Account	Number	Owner	Balance	Type
	101	J. Smith	1000.00	checking
	102	W. Wei	2000.00	checking
	103	J. Smith	5000.00	savings
	104	M. Jones	1000.00	checking
	105	H. Martin	10,000.00	checking

Projection: Example

π = project

Sample query: $\pi_{\text{Number, Owner, Type}}$ Account

Account	Number	Owner	Balance	Type
	101	J. Smith	1000.00	checking
	102	W. Wei	2000.00	checking
	103	J. Smith	5000.00	savings
	104	M. Jones	1000.00	checking
	105	H. Martin	10,000.00	checking

	Number	Owner	Type
	101	J. Smith	checking
	102	W. Wei	checking
	103	J. Smith	savings
	104	M. Jones	checking
	105	H. Martin	checking

Projection: Example

π = project

Sample query: $\pi_{\text{Number, Owner, Type}}$ Account

Account	Number	Owner	Balance	Type
	101	J. Smith	1000.00	checking
	102	W. Wei	2000.00	checking
	103	J. Smith	5000.00	savings
	104	M. Jones	1000.00	checking
	105	H. Martin	10,000.00	checking

	Number	Owner	Type
	101	J. Smith	checking
	102	W. Wei	checking
	103	J. Smith	savings
	104	M. Jones	checking
	105	H. Martin	checking

Schema of result is a subset of the schema of input relation

Projection: Another Example

π_{Owner} Account

Account	Number	Owner	Balance	Type
	101	J. Smith	1000.00	checking
	102	W. Wei	2000.00	checking
	103	J. Smith	5000.00	savings
	104	M. Jones	1000.00	checking
	105	H. Martin	10,000.00	checking

	Owner
	J. Smith
	W. Wei
	M. Jones
	H. Martin

Note: Projection operator eliminates *duplicates*, *Why???*

In a DBMS products, do you think duplicates should be eliminated for every query? Are they?

Extended (Generalized) Projection

- Allows arithmetic functions and duplicate occurrences of the same attribute to be used in the projection list

$$\prod_{\text{F1, F2, …, Fn}}(E)$$

- E is any relational-algebra expression
- $F_1, F_2, \ldots, F_n$ 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:

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

Can use rename to give a name to the column!

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

Extended Projection: Another Example

R =

A	B
1	2
3	4

$\pi_{A+B\text{->}C,A,A}$ (R) =

C	A1	A2
3	1	1
7	3	3

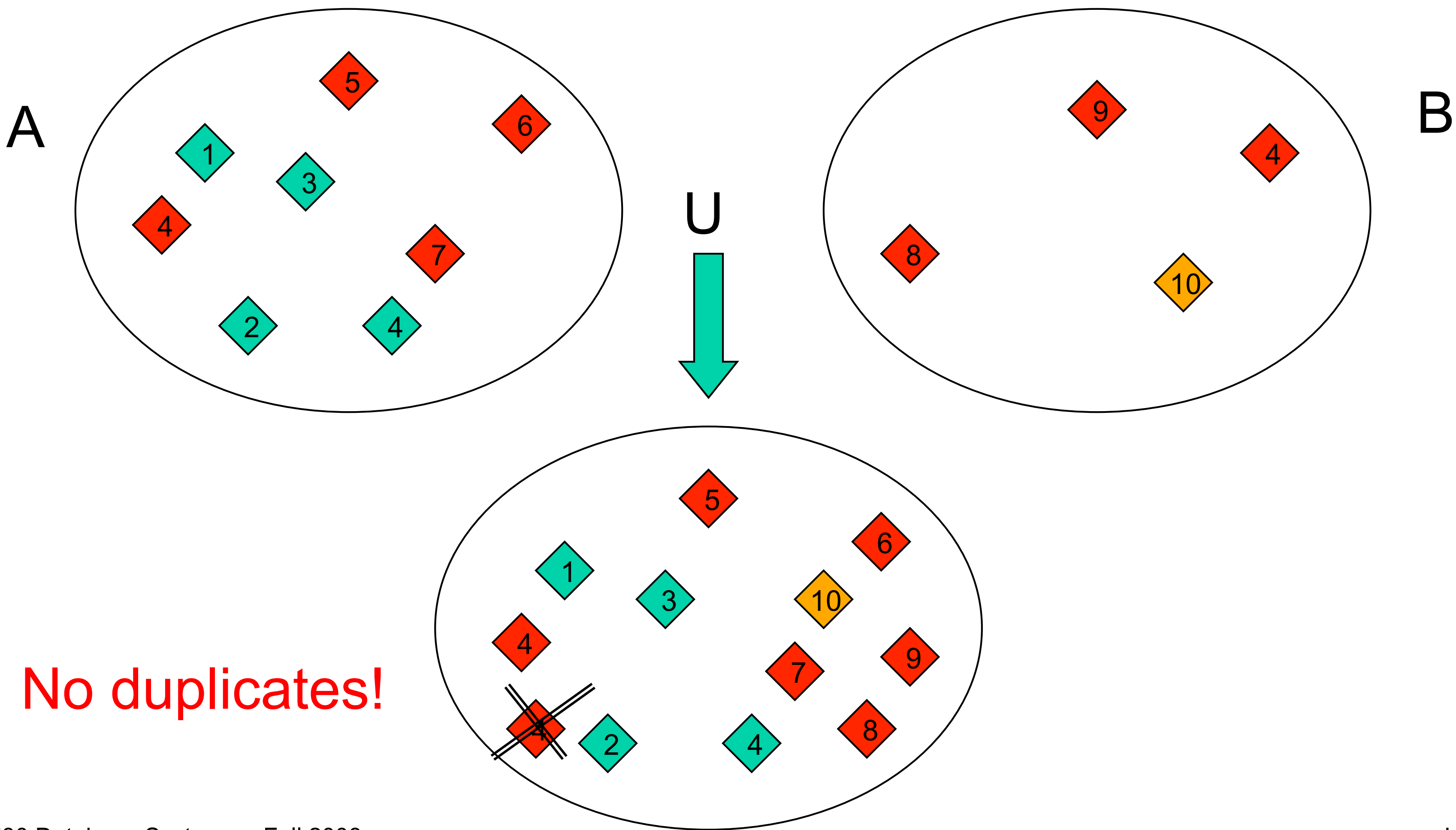
Projection

- Deletes attributes that are not in *projection list*.
- *Schema* of result contains exactly the *fields in the projection list*, with the same names that they had in the input relation.
- Projection operator has to eliminate *duplicates*
  - duplicates are always eliminated in relational algebra: relations are sets!
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

Usual Set Operations

Union of Two Sets

- C = A U B



Union: Example

∪ = union

Checking-account ∪ Savings-account

Checking-account	c-num	c-owner	c-balance
	101	J. Smith	1000.00
	102	W. Wei	2000.00
	104	M. Jones	1000.00
	105	H. Martin	10,000.00

Savings-account	s-num	s-owner	s-balance
	103	J. Smith	5000.00

	c-num	c-owner	c-balance
	101	J. Smith	1000.00
	102	W. Wei	2000.00
	104	M. Jones	1000.00
	105	H. Martin	10,000.00
	103	J. Smith	5000.00

Union Compatible

- Two relations are *union-compatible* if they have the same degree (i.e., the same number of attributes) and the corresponding attributes are defined on the same domains.
- Suppose we have these tables:

Checking-Account (c-num:str, c-owner:str, c-balance:real)

Savings-Account (s-num:str, s-owner:str, s-balance:real)

These are union-compatible tables.

- *Union, intersection, & difference require union-compatible tables*

Intersection

$\cap$ = intersection

Checking-account $\cap$ Savings-account

What's the answer to this query?

Checking-account

c-num	c-owner	c-balance
101	J. Smith	1000.00
102	W. Wei	2000.00
104	M. Jones	1000.00
105	H. Martin	10,000.00

Savings-account

s-num	s-owner	s-balance
103	J. Smith	5000.00

Intersection (cont.)

Checking-account

c-num	c-owner	c-balance
101	J. Smith	1000.00
102	W. Wei	2000.00
104	M. Jones	1000.00
105	H. Martin	10,000.00

Savings-account

s-num	s-owner	s-balance
103	J. Smith	5000.00

Checking-account ∩ Savings-account

What's the answer to this query?

It's empty. There are no tuples that are in both tables.

$(\pi_{\text{c-owner}} \text{Checking-account}) \cap (\pi_{\text{s-owner}} \text{Savings-account})$

What's the answer to this new query?

Intersection (cont.)

Checking-account

c-num	c-owner	c-balance
101	J. Smith	1000.00
102	W. Wei	2000.00
104	M. Jones	1000.00
105	H. Martin	10,000.00

Checking-account ∩ Savings-account

What's the answer to this query?

Savings-account

s-num	s-owner	s-balance
103	J. Smith	5000.00

It's empty. There are no tuples that are in both tables.

$(\pi_{c\text{-}owner}\text{Checking-account}) \cap (\pi_{s\text{-}owner}\text{Savings-account})$

What's the answer to this new query?

	c-owner
	J. Smith

Difference

— = difference

Checking-account

c-num	c-owner	c-balance
101	J. Smith	1000.00
102	W. Wei	2000.00
104	M. Jones	1000.00
105	H. Martin	10,000.00

Savings-account

s-num	s-owner	s-balance
103	J. Smith	5000.00

Find all the customers that own a Checking-account and do not own a Savings-account.

$(\pi_{\text{c-owner}}\text{Checking-account}) - (\pi_{\text{s-owner}}\text{Savings-account})$

- What is the *schema* of result?

c-owner
W. Wei
M. Jones
H. Martin

Challenge Question

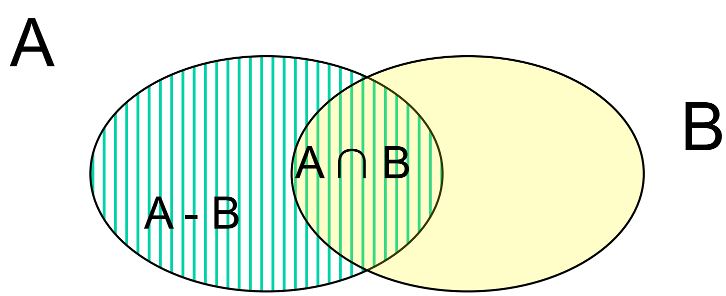
- How could you express the intersection operation if you didn't have an Intersection operator in relational algebra? [Hint: Can you express Intersection using only the Difference operator?]

$A \cap B =$???

Challenge Question

- How could you express the intersection operation if you didn't have an Intersection operator in relational algebra? [Hint: Can you express Intersection using only the Difference operator?]

$A \cap B = A - (A - B)$



Combining Tuples of Two Relations

Cross Product: Example

X cross product

Teacher

t-num	t-name
101	Smith
105	Jones
110	Fong

Teacher X Course

Course

c-num	c-name
514	Intro to DB
513	Intro to OS

t-num	t-name	c-num	c-name
101	Smith	514	Intro to DB
105	Jones	514	Intro to DB
110	Fong	514	Intro to DB
101	Smith	513	Intro to OS
105	Jones	513	Intro to OS
110	Fong	513	Intro to OS

Cross product: combine information from 2 tables

- produces: *every possible combination of* a teacher and a course

Cross Product

- R1 X R2
- Each row of R1 is paired with each row of R2.
- *Result schema* has one field per field of R1 and R2, with field names `inherited' if possible.
- *What about R1 X R1?*

Teacher X Teacher t-num t-name t-num t-name

Conflict!

- *Renaming operator*:

Teacher X ρ_T (Teacher) = Teacher X T

$\boldsymbol{\rho}_{T(t\text{-}num1,\ t\text{-}name1)}$ (Teacher) X Teacher   No conflict!

Renaming

- The ρ operator gives a new schema to a relation.
- $R1 := \rho_{R1(A1,\ldots,An)}(R2)$ makes R1 be a relation with attributes $A1,\ldots,An$ and the same tuples as R2.
- Simplified notation: $R1(A1,\ldots,An) := R2$.

Example: Renaming

Bookstore(

name,	addr
Joe’s	Maple St.
Sue’s	River Rd.

)

R(bs, addr) := Bookstore

R(

bs,	addr
Joe’s	Maple St.
Sue’s	River Rd.

)

Relational Algebra vs. Set Theory
(cross product)

Suppose.. A = {a, b, c} B = {1, 2} C = {x, y} then in

set theory, the cross product is defined as:

A X B = {(a, 1), (b, 1), (c, 1), (a, 2), (b, 2), (c, 2)}

and (A X B) X C =

{((a,1),x), ((b,1),x), ((c,1),x), ((a,2),x), ((b,2),x), ((c,2),x),
((a,1),y), ((b,1),y), ((c,1),y), ((a,2),y), ((b,2),y), ((c,2),y)}

Relational Algebra vs. Set Theory
(cross product) (cont.)

Given A = {a, b, c} B = {1, 2} C = {x, y} with the cross product (A X B) X C in *set theory* =

{((a,1),x), ((b,1),x), ((c,1),x), ((a,2),x), ((b,2),x), ((c,2),x),
((a,1),y), ((b,1),y), ((c,1),y), ((a,2),y), ((b,2),y), ((c,2),y)}

we simplify it in *relational algebra* to:

{(a,1,x), (b,1,x), (c,1,x), (a,2,x), (b,2,x), (c,2,x),
(a,1,y), (b,1,y), (c,1,y), (a,2,y), (b,2,y), (c,2,y)}

by eliminating parentheses….”flattening” the tuples.

Join: Example

⋈ = join

Account ⋈$_{Number=Account}$ Deposit

Account	Number	Owner	Balance	Type
	101	J. Smith	1000.00	checking
	102	W. Wei	2000.00	checking
	103	J. Smith	5000.00	savings
	104	M. Jones	1000.00	checking
	105	H. Martin	10,000.00	checking

Deposit	Account	Transaction-id	Date	Amount
	102	1	10/22/00	500.00
	102	2	10/29/00	200.00
	104	3	10/29/00	1000.00
	105	4	11/2/00	10,000.00

	Number	Owner	Balance	Type	Account	Transaction-id	Date	Amount
	102	W. Wei	2000.00	checking	102	1	10/22/00	500.00
	102	W. Wei	2000.00	checking	102	2	10/29/00	200.00
	104	M. Jones	1000.00	checking	104	3	10/29/00	1000.00
	105	H. Martin	10,000.00	checking	105	4	11/2/00	10000.00

Join: Example

$\bowtie$ join

$$Account \bowtie_{Number=Account} Deposit$$

Note that when the join is based on equality, then we have two identical attributes (columns) in the answer.

Number	Owner	Balance	Type	Account	Trans-id	Date	Amount
102	W. Wei	2000.00	checking	102	1	10/22/00	500.00
102	W. Wei	2000.00	checking	102	2	10/29/00	200.00
104	M. Jones	1000.00	checking	104	3	10/29/00	1000.00
105	H. Martin	10,000.00	checking	105	4	11/2/00	10000.00

Join: Example

$\bowtie$ join

Account $\bowtie_{(Number=Account\ and\ Amount>700)}$ Deposit

Account	Number	Owner	Balance	Type
	101	J. Smith	1000.00	checking
	102	W. Wei	2000.00	checking
	103	J. Smith	5000.00	savings
	104	M. Jones	1000.00	checking
	105	H. Martin	10,000.00	checking

Deposit	Account	T-id	Date	Amount
	102	1	10/22/00	500.00
	102	2	10/29/00	200.00
	104	3	10/29/00	1000.00
	105	4	11/2/00	10,000.00

	Number	Owner	Balance	Type	Account	T-id	Date	Amount
	104	M. Jones	1000.00	checking	104	3	10/29/00	1000.00
	105	H. Martin	10,000.00	checking	105	4	11/2/00	10000.00

Challenge Question

- How could you express the "join" operation if you didn't have a join operator in relational algebra? [Hint: are there other operators that you could use, in combination?]

Joins

- *Condition Join*: $R \bowtie_c S = \sigma_c (R \times S)$
  - Sometimes called a *theta-join*
- *Result schema* same as that of cross-product
- Fewer tuples than cross-product, might be able to compute more efficiently

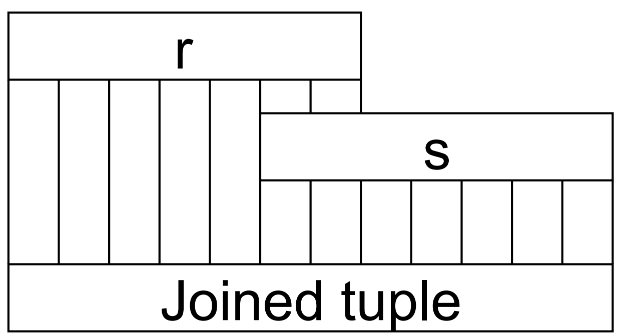
Joins

- *Equi-Join*: A special case of condition join where the condition *c* contains only ***equalities*.**

$$\text{Student} \bowtie_{sid} \text{Takes}$$

  - *Result schema* similar to cross-product, but only one copy of fields for which equality is specified.

- *Natural Join*: Equijoin on *all* common fields.



Challenge Question

- How could you express the natural join operation if you didn't have a natural join operator in relational algebra? Consider you have two relations R(A,B,C) and S(B,C,D).

  ????

Challenge Question

- How could you express the natural join operation if you didn't have a natural join operator in relational algebra? Consider you have two relations R(A,B,C) and S(B,C,D).

$$\pi_{R.A,R.B,R.C,\ S.D}\,(\sigma_{R.B=S.B \text{ and } R.C=S.C}\ (R \times S))$$

The Divide Operator

Suppose we have this extra table, in the Bank database:

Account-types

Type
checking
savings

And that we would like to know which customers have *all* types of accounts…

We can use the Divide operator

$\div$ or / divide $(\pi_{\text{Owner, Type}}$ Account) $\div$ Account-types

Account	~~Number~~	Owner	~~Balance~~	Type
	~~101~~	J. Smith	~~1000.00~~	checking
	~~102~~	W. Wei	~~2000.00~~	checking
	~~103~~	J. Smith	~~5000.00~~	savings
	~~104~~	M. Jones	~~1000.00~~	checking
	~~105~~	H. Martin	~~10,000.00~~	checking

Account-types	Type
	checking
	savings

	Owner
	J. Smith

Find account owners who have ALL types of accounts.

We can use the Divide operator

÷ or / divide $(\pi_{\text{Owner, Type}} \text{Account}) \div \text{Account-types}$

Account	Number	Owner	Balance	Type
	101	J. Smith	1000.00	checking
	102	W. Wei	2000.00	checking
	103	J. Smith	5000.00	savings
	104	M. Jones	1000.00	checking
	105	H. Martin	10,000.00	checking

Account-types	Type
	checking
	savings

	Owner
	J. Smith

Find account owners who have ALL types of accounts.

Divide Operator

For R ÷ S where R (r1, r2, r3, r4) and S(s1, s2)

Since S has two attributes, there must be two attributes in R (say r3 and r4) that are defined on the same domains, respectively, as s1 and s2. We could say that (r3, r4) is *union-compatible* with (s1, s2).

The query answer has the remaining attributes (r1, r2). And the answer has a tuple (r1, r2) in the answer if the (r1, r2) value appears with *every* S tuple in R.

Division

- Not supported as a primitive operator, but useful for expressing queries like:

  Find customers who have all types of accounts.

- Let *A* have 2 fields, *x* and *y*; *B* have only field *y*:
  - *A/B* = $\left\{ \langle x \rangle \mid \exists \langle x, y \rangle \in A \ \forall \langle y \rangle \in B \right\}$
  - i.e., ***A/B* contains all *x* tuples (customers) such that for *every y* tuple (account type) in *B*, there is an *xy* tuple in *A*.**
  - *Or*: If the set of *y* values (account types) associated with an *x* value (customer) in *A* contains all *y* values in *B*, the *x* value is in *A/B*.

- In general, *x* and *y* can be any lists of fields; *y* is the list of fields in *B*, and $x \cup y$ is the list of fields of *A*.

Examples of Division: Suppliers and Parts

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

A

pno
p2

B1

pno
p2
p4

B2

pno
p1
p2
p4

B3

sno
s1
s2
s3
s4

A/B1

sno
s1
s4

A/B2

sno
s1

A/B3

Expressing A/B Using Basic Operators

- Division is not an essential op, but it provides a useful shorthand
  - (Also true of joins, but joins are so common that systems implement joins specially.)
- *Idea*: For *A/B*, compute all *x* values that are not `disqualified' by some *y* value in *B*.
  - *x* value is *disqualified* if by attaching *y* value from *B*, we obtain an *xy* tuple that is not in *A*.

Disqualified x values: $\pi_x((\pi_x(A) \times B) - A)$

A/B: $\pi_x(A) -$ all disqualified tuples

Division: Example

Disqualified x values: $\pi_x((\pi_x(A) \times B) - A)$

A/B: $\pi_x(A)$ – all disqualified tuples

- A = ((s1,p1), (s1,p2),(s2,p1),(s3,p2))
- B = (p1,p2)
- A/B = ???
- π_X (A) = (s1,s2,s3) – duplicates are removed!
- π_X (A) X B = ((s1,p1),(s1,p2),(s2,p1),(s2,p2),(s3,p1),(s3,p2))
- (π_X (A) X B) – A = ((s2,p2),(s3,p1))
- π_X((π_X (A) X B) – A) = (s2,s3) ← disqualified tuples
- A/B = (s1,s2,s3) – (s2,s3) = (s1)

Building Complex Expressions

- Combine operators with parentheses and precedence rules.
- Three notations, just as in arithmetic:
  1. Sequences of assignment statements.
  2. Expressions with several operators.
  3. Expression trees.

Sequences of Assignments

- Create temporary relation names.
- Renaming can be implied by giving relations a list of attributes.

- Example: R3 := R1 $\bowtie_C$ R2 can be written:

  R4 := R1 X R2

  R3 := σ_C (R4)
- Example: Write $r \div s$ as

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

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

$$result := temp1 - temp2$$

Expressions in a Single Assignment

- Example: the theta-join R3 := R1 $\bowtie_C$ R2 can be written: R3 := σ_C (R1 X R2)
- Precedence of relational operators:
    1. [σ, π, ρ] (highest)
    2. [x, $\bowtie$]
    3. $\cap$
    4. [$\cup$, —]

Expression Trees

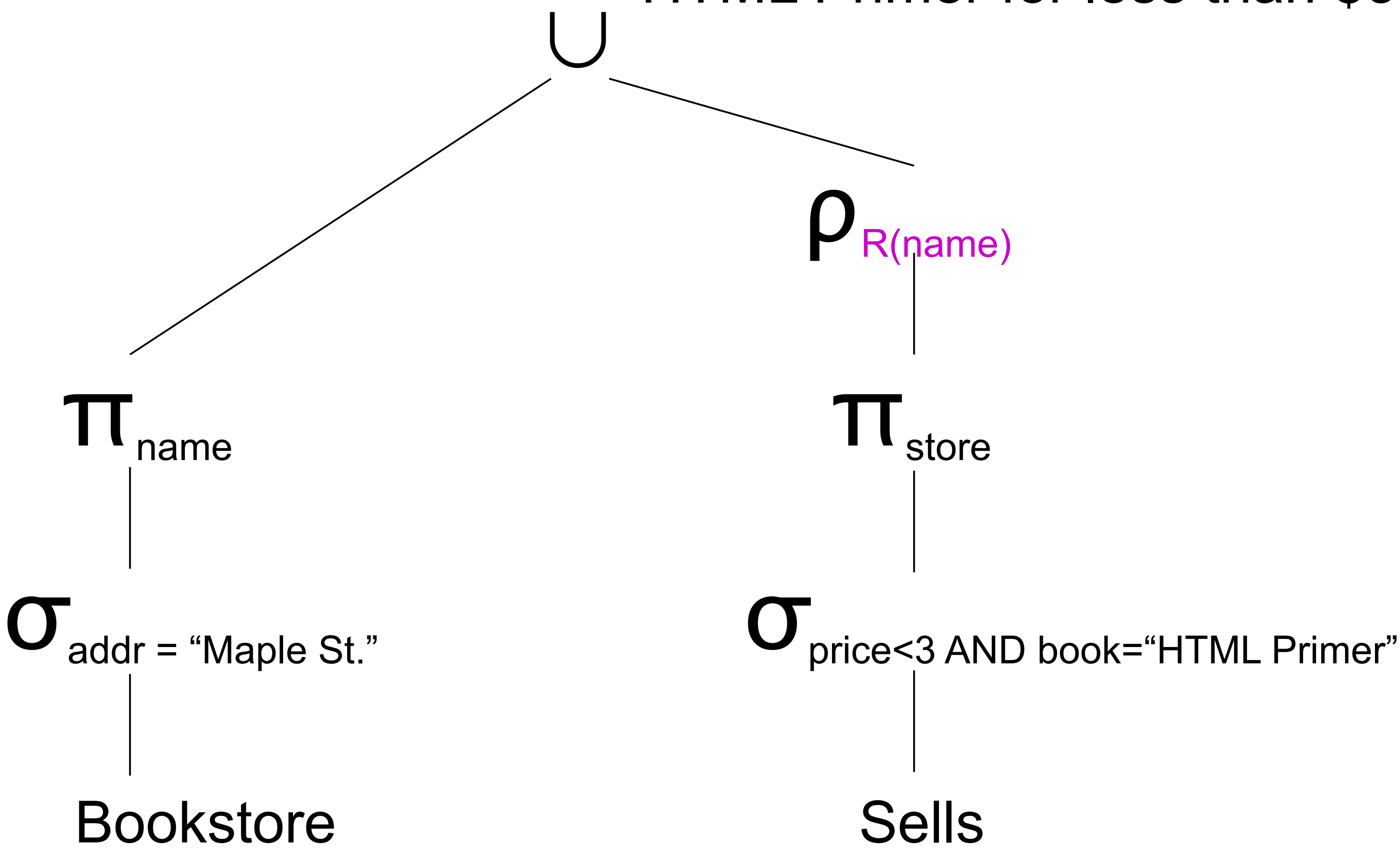
- Leaves are operands --- either variables standing for relations or particular, constant relations.
- Interior nodes are operators, applied to their child or children.

Example: Tree for a Query

- Using the relations Bookstore(name, addr) and Sells(store, book, price), find the names of all the stores that are either on Maple St. or sell HTML Primer for less than $3.

As a Tree:

Bookstore(name, addr) and Sells(store, book, price), find the names of all the stores that are either on Maple St. or sell HTML Primer for less than $3.



Aggregate Functions

- Takes a collection of values and returns a single value as a result.
- Operators:

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

e.g., $\text{sum}_{\text{salary}}$(Pt-works),
Where Pt-works-scheme = (employee-name, branch-name, salary)

- Can control whether duplicates are eliminated

  $\text{count-distinct}_{\text{branch-name}}$ (Pt-works)

Aggregation and Grouping

- Apply aggregation to groups of tuples
  - Example: sum salaries at each branch
- Sample result:

branch-name	sum of salary
Downtown	*5300*
Austin	*3100*
Perryridge	*8100*

- Notation

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

  - E is any relational-algebra expression
  - $G_1, G_2 \ldots, G_n$ 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

$$\mathcal{G}_{\mathbf{sum(c)}}(r)$$

sum-C
27

or

$$\mathrm{sum}_c(r)$$

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

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

branch-name	*balance*
Perryridge	1300
Brighton	1500
Redwood	700

Aggregate Functions (Cont.)

- 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

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

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 (roughly speaking) **false** by definition.
    - Will study precise meaning of comparisons with nulls later

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 ⟕ *Borrower*

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

How would you represent an outer join using the basic relational algebra operations?

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

Modifying the Database

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

- Remove tuples from a relation
- 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 expression.

Deletion Examples

Account(acc_number, branch_name, balance)
Depositor(cust_name, acc_number)
Branch(branch_name, city)
Loan(loan_number,branch_name,amount)

- 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 50

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

- Delete all accounts at branches located in Needham.

$$r_1 \leftarrow \sigma_{\ branch\text{-}city\ =\ \text{"Needham"}}\ (account \bowtie branch)$$

$$r_2 \leftarrow \prod_{branch\text{-}name,\ account\text{-}number,\ balance}\ (r_1)$$

$$account \leftarrow \text{account} - r_2$$

$$r_3 \leftarrow \prod_{\ customer\text{-}name,\ account\text{-}number}\ (r_2 \bowtie \text{depositor})$$

$$depositor \leftarrow \text{depositor} - r_3$$

Is this correct?

Insertion

- Insert tuples (rows) into a relation
  - specify a tuple to be inserted
  - write a query whose result is a set of tuples to be inserted
- Insertion is expressed in relational algebra 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

Account(acc_number, branch_name, balance)
Depositor(cust_name, acc_number)
Borrower(cust_name,loan_number)
Loan(loan_number,branch_name,amount)

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

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

  $$\text{depositor} \leftarrow \textit{depositor} \cup \{(\text{"Smith"}, \text{A-973})\}$$

- 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.

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

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

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

Can you always insert a new tuple into a relation?

Updating

- 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 ith attribute of r, if the ith 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

- Motivation:
  - Protect (hide) information in relations
  - Customize database to better match a user's need

e.g., it is not necessary for the marketing manager to know the loan amount

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

- Any relation that is not part of the logical 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.
  - Static vs. dynamic

View Examples

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

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

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

- We can find all customers of the Perryridge branch by writing:

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

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**

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

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

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

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 (“L-37”, “Perryridge”, *null*) into the *loan* relation
- Others cannot be translated uniquely
  - E.g., suppose the all-customer view contains information about all branches and their customers:

all-customer(branch_name,cust_name)=

$$\prod_{branch\_name,c\_name}(Branch \bowtie Loan) \cup \prod_{branch\_name,c\_name}(Branch \bowtie Account)$$

  - *all-customer* ← *all-customer* ∪ {(“Perryridge”, “John”)}
    - Have to choose loan or account, and create a new loan/account number!

Schema

- Account(acc-number,branch-name,balance)
- Branch(branch-name,branch-city,assets)
- Depositor(cust-name,account-number)
- Customer(cust-name,cust-street,cust-city)
- Borrower(cust-name,loan-number)
- Loan(loan-number,branch-name,amount)

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
- 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
- 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

Practice Exercise

Why do we use Relational Algebra?

Because:

- It is mathematically defined (where relations are sets)
- We can prove that two relational algebra expressions are equivalent. For example:

$$\sigma_{cond1}\,(\sigma_{\,cond2}\; R) \equiv \sigma_{\,cond2}\,(\sigma_{\,cond1}\; R) \equiv \sigma_{\,cond1 \text{ and } cond2}\; R$$

$$R1 \bowtie_{cond} R2 \equiv \sigma_{cond}\,(R1 \text{ X } R2)$$

$$R1 \div R2 \equiv \pi_x(R1) - \pi_x((\pi_x\; R1) \text{ X } R2) - R1)$$

Uses of Relational Algebra Equivalences

- To help query writers - they can write queries in several different ways

- To help query optimizers - they can choose the *most efficient* among different ways to execute the query

and in both cases we know for sure that the two queries (the original and the replacement) are identical…that they will produce the same answer

Find names of stars and the length of the movies they have appeared in 1994

Stars(<u>name</u>, address)

AppearIn(<u>star_name,title, year</u>),

Movies(<u>title, year</u>, length, type, studio_name)

- Information about movie length available in Movies; so need an extra join:

$$\pi_{name,length}\,(\sigma_{year=1994}(Stars \bowtie AppearIn \bowtie Movies))$$

- A more efficient solution:

$$\pi_{name,length}(Stars \bowtie AppearIn \bowtie (\sigma_{year=1994}(\,Movies))$$

- An even more efficient solution:

$$\pi_{name,length}(Stars \bowtie \pi_{name,length}(AppearIn \bowtie (\pi_{title,year,length}\sigma_{year=1994}(Movies)))$$

A query optimizer can find this!

Question

- Relational Algebra is not Turing complete. There are operations that cannot be expressed in relational algebra.
- What is the advantage of using this language to query a database?

Question

- Relational Algebra is not Turing complete. There are operations that cannot be expressed in relational algebra.
- What is the advantage of using this language to query a database?

By limiting the scope of the operations, it is possible to automatically optimize queries

Summary

- The relational model has rigorously defined query languages that are simple and powerful.
- Relational algebra is more operational; useful as internal representation for query evaluation plans.
- Several ways of expressing a given query; a query optimizer should choose the most efficient version.

- More on Views

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

Updates Through Views

- Updates to views must be translated to updates over the base 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**

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

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

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

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

Updates Through Views

- How to translate the view update into an update to the *loan* relation?
- 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
- 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!

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

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

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

Update Through Views: Problem Can Get Much Worse

- Example

  create view branch_city as

  $\Pi_{\text{branch\_name, customer\_city}}$ (borrow ⋈ customer)

- Now, an update

  branch_city ← branch_city ∪ { ("Brighton, "Woodside") }

- Tuples created

branch_name	***loan_number***	***customer_name***	***amount***
Brighton	***null***	***null***	***null***

customer_name	***street***	***customer_city***
null	***null***	***Woodside***

But What Happens When We Access Through This View?

- Suppose we do:

  $\Pi_{\text{branch\_name, customer\_city}}$ (branch_city)

- The result does not include:

  (“Brighton”, “Woodside”)

- Why?
  - Comparisons on null values always yield false
  - As they must -- since they mean “no value”
- Result is anomalous (strange):
  - Insert OK, then missing when queried

MORE ON NULLS LATER!