Lesson 10: Relational Data Model & SQL

Contents

- Structure of Relational Databases
- Relational Algebra
- Basic Relational-Algebra Operations
- Additional Relational-Algebra Operations
- Extended Relational-Algebra Operations
- Null Values and Three-valued Logics
- Database Modification by Relational-Algebra Operations

- Brief Introduction to SQL
- SQL and Relations
- Fundamental SQL statements
- *null* values in SQL
- Database modifications in SQL

Why Relations?

account_number	*branch_name*	*balance*
A-101	Downtown	500
A-102	Bridges	400
A-201	Brighton	900
A-215	Berkeley	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Palo Alto	350

- We have seen tables
- Why do we need another view of data?
- There is a number of reasons:
  - Need to create a rigorous mathematical model
  - This model enables for formalizing database operations
  - The exact model is needed to formulate declarative queries and optimize their processing
- The central idea is to describe a database as a collection of predicates over a finite set of predicate variables, defining constraints on the possible values and combinations of values.

What is a *Relation*?

- Mathematically, given sets D_1, D_2, …, D_n a **relation** R is a ***subset*** of the Cartesian product $D_1 \times D_2 \times \ldots \times D_n$
  Thus, a relation is a set of n-tuples (a_1, a_2, …, a_n) where each $a_i \in D_i$
- Example:
  - *customer_name* = {Jones, Smith, Curry, Lindsay, …} /* Set of all customer names */
  - *customer_street* = {Main, North, Park, …} /* Set of all street names*/
  - *customer_city* = {Harrison, Rye, Pittsfield, …} /* Set of all city names */

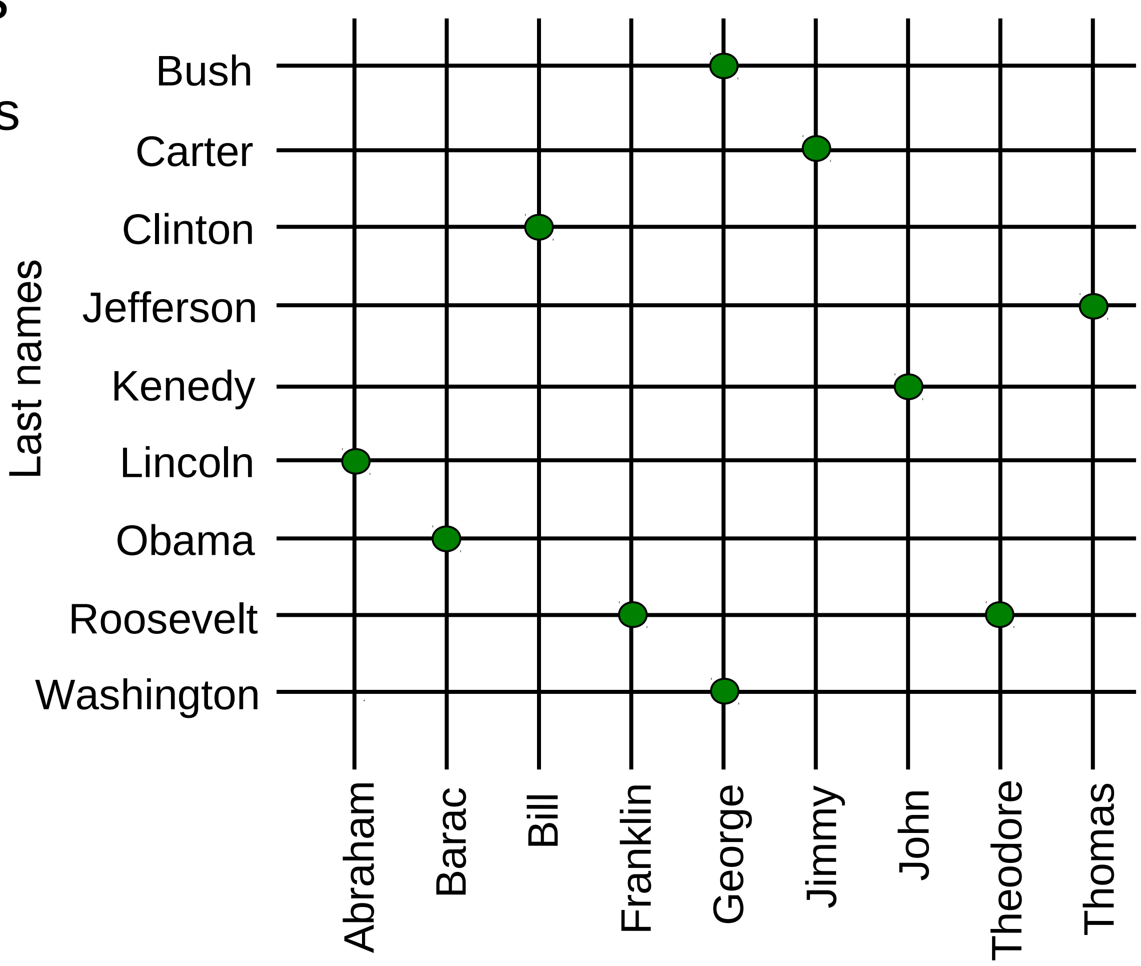
  Then *r* = { (Jones, Main, Harrison),
  (Smith, North, Rye),
  (Curry, North, Rye),
  (Lindsay, Park, Pittsfield) }

  is a relation, i.e. subset of

  customer_name x *customer_street* x *customer_city*
- As we are concerned with finite sets, such sets can be expressed by enumeration, i.e. tables

Relation is a Subset of a Cartesian Product

- No duplicates in sets
  - Very important for database applications
- Component set members can be in any order
  - Sorted or unsorted



Selected U.S. Presidents

Attribute Types

- Each attribute of a relation has a name
- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**; that is, indivisible
  - E.g. the value of an attribute can be an account number, but cannot be a set of account numbers
- Domain is said to be atomic if all its members are atomic
- The special value *null* is a member of every domain
- The null value causes complications in the definition of many operations
  - We shall ignore the effect of null values in our main presentation and consider their effect later

Relation Schema & Relation Instance

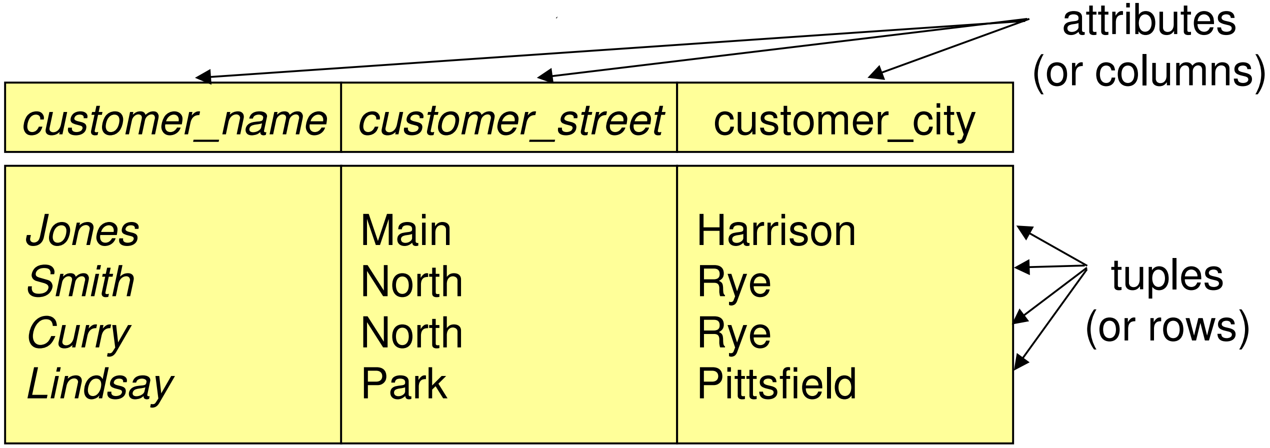
- Relation Schema
  - $A_1, A_2, \ldots, A_n$ are *attributes*
  - $R = (A_1, A_2, \ldots, A_n)$ is a *relation schema*

    Example:

    Customer_schema = (*customer_name, customer_street, customer_city*)
  - *r*(*R*) denotes a *relation r* on the *relation schema R*

    Example:

    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

Database

- A database consists of multiple relations
- Information about an enterprise is broken up into parts, with each relation storing one part of the information

account :	stores information about accounts
depositor :	stores information about which customer owns which account
customer :	stores information about customers

- Storing all information as a single relation such as

  bank(*account_number, balance, customer_name*, ..)

  results in
  - repetition of information
    - e.g., if two customers own an account (What gets repeated?)
  - the need for null values
    - e.g., to represent a customer without an account

- Normalization theory deals with how to design relational schemas

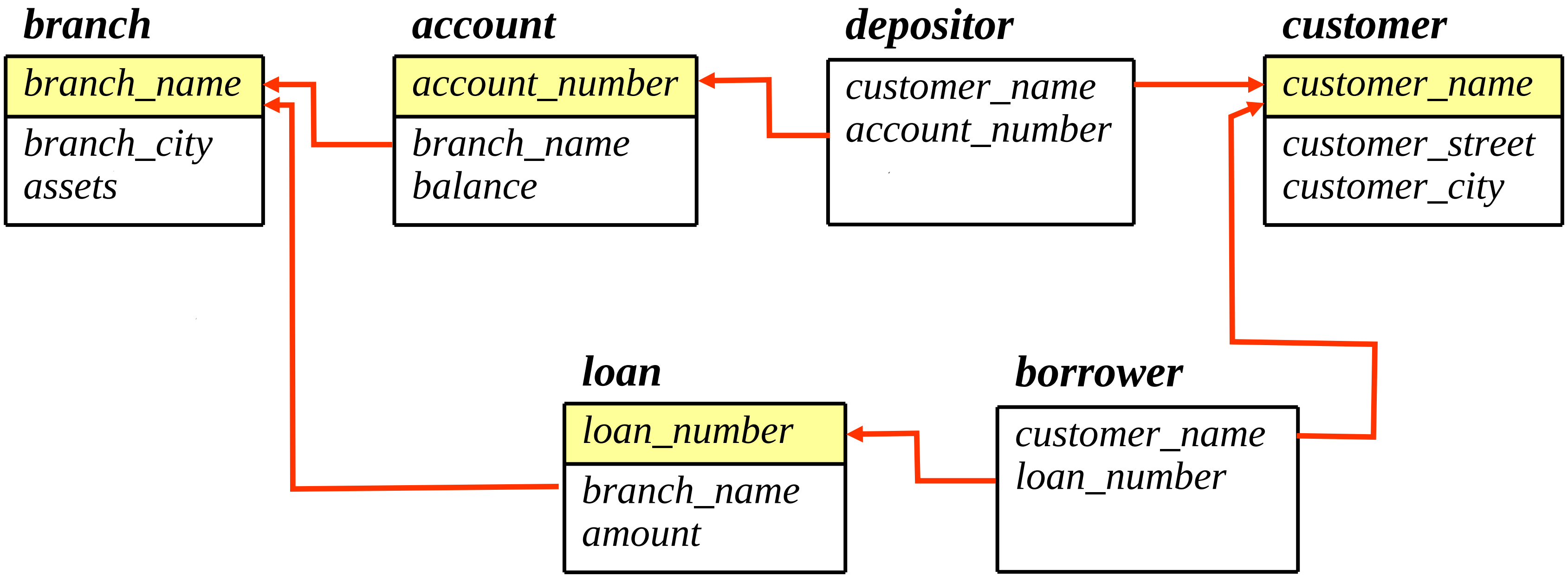
Keys (revisited)

- Let K ⊆ 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
    - In real life, an attribute such as *customer_id* would be used instead of *customer_name* to uniquely identify customers, but we omit it to keep our examples small, and instead assume customer names are unique.
- *K* is a **candidate key** if *K* is minimal
  - Example: {*customer_name*} is a candidate key for *Customer*, since it is a superkey and no subset of it is a superkey.
- **Primary key:** a candidate key chosen as the principal means of identifying tuples within a relation
  - Should choose an attribute whose value never, or very rarely, changes.
    - E.g. email address is unique, but may change

Foreign Keys

- A relation schema may have an attribute that corresponds to the primary key of another relation. The attribute is called a **foreign key**.
  - E.g. *customer_name* and *account_number* attributes of *depositor* are foreign keys to *customer* and *account* respectively.
  - Only values occurring in the primary key attribute of the **referenced relation** may occur in the foreign key attribute of the **referencing relation**.



branch
branch_name
branch_city
assets

account
account_number
branch_name
balance

depositor
customer_name
account_number

customer
customer_name
customer_street
customer_city

loan
loan_number
branch_name
amount

borrower
customer_name
loan_number

Relational Algebra

- Procedural language
- Six basic operators
  - select: σ
  - project: $\prod$
  - union: $\cup$
  - set difference: –
  - Cartesian product: x
  - rename: ρ

- The operators take one or two relations as inputs and produce a new relation as a result.

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 : $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
Each **term** is one of:

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

where *op* is one of: $=, \neq, >, \geq, <, \leq$

- Example of selection: $\sigma_{branch\_name=\text{“Redwood”}}(account)$

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

Project Operation

- Notation: $\prod_{A_1, A_2, \ldots, A_k}(r)$

  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 are removed from result, since relations are sets
- Example: To eliminate the *branch_name* attribute of *account*

$$\prod_{account\_number,\ balance}(account)$$

r

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

$\prod_{A,C}(r)$

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

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** (example: 2nd column of r deals with the same type of values as does the 2nd column of s)
- Example: to find all customers with either an account or a loan

$$\Pi_{customer\_name}(depositor) \cup \Pi_{customer\_name}(borrower)$$

Relations r, s:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

$r \cup s$:

A	B
α	1
α	2
β	1
β	3

Set Difference Operation

- Notation $r - s$
- Defined as:

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

- 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

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

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, $R \cap S = \emptyset$.
- Can build expressions using multiple operations
- If attributes of *r*(*R*) and *s*(*S*) are not disjoint, then *renaming* must be used.

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

Caution: May generate HUGE tables

Composition of Operations

- Building operations by composing several together

$r \times s$:

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	10	a
β	2	β	20	b

Rename Operation

- Not a "true relational algebra" 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

$$\rho_{X(A_1, A_2, \ldots, A_n)}(E)$$

returns the result of expression E under the name X, and with the attributes renamed to A_1, A_2, …., A_n.

Banking Example

- Relations
  - *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 all loans of over $1200

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

  - Find the loan number for each loan amounting over $1200

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

  - Find the names of all customers who have an account at the Redwood branch

    $$\Pi_{customer_{name}}(\sigma_{branch_{name} = Redwood}$$
    $$(\sigma_{depositor.account\_number = account.account\_number}(depositor \times loan)))$$

Banking Example (cont.)

- Example Queries (cont.)
  - Find the names of all customers who have a loan at the Redwood branch but do not have an account at any branch of the bank

$$\Pi_{customer\_name}(\sigma_{branch\_name = Redwood}(\sigma_{borrower.loan\_number = loan.loan\_number}(borrower \times loan))) - \Pi_{customer\_name}(depositor)$$

  - Find the names of all customers who have a loan at the Redwood branch
    - Possibility No. 1

$$\Pi_{customer_{name}}(\sigma_{branch_{name} = Redwood}(\sigma_{borrower.loan\_number = loan.loan\_number}(borrower \times loan)))$$

    - Possibility No. 2

$$\Pi_{customer_{name}}(\sigma_{borrower.loan\_number = loan.loan\_number}((\sigma_{branch_{name} = Redwood}(borrower)) \times loan))$$

Banking Example (cont.)

- Example Queries (use of *rename*)
  - Find the largest account balance
  - Strategy:
    - Find those balances that are *not* the largest
    - Rename *account* relation as *temp* so that we can compare each account balance with all others
    - Use set difference to find those account balances that were *not* found in the earlier step.
  - The query is:

$$\Pi_{balance}(account) - \Pi_{account.balance}\left(\sigma_{account.balance < temp.balance}\left(account \times \rho_{temp}(account)\right)\right)$$

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:

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

- Assume:
  - *r*, *s* have the *same arity*
  - attributes of *r* and *s* are compatible
- Note: $r \cap s = r - (r - s)$

Relations *r*, *s*:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

$r \cap s$:

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 $R \cup S$ 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 $R \cap S$, 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
- The result of the natural join is the set of all combinations of tuples in R and S that are equal on their common attribute names
- Example:

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

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

Natural Join Operation – Example

- Relations *r*, *s*:

A	*B*	*C*	*D*
α	1	μ	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	∈

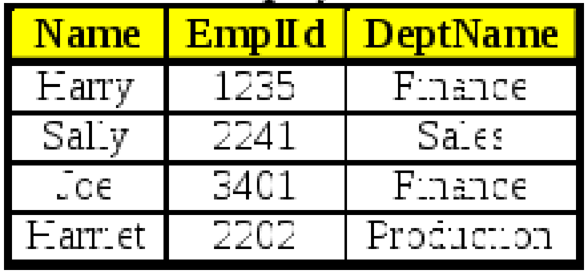
s

r ⋈ *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

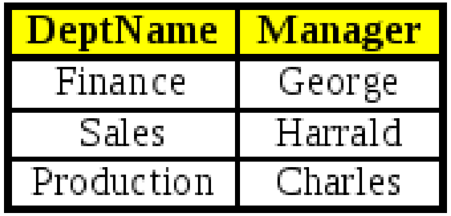
- Practical example

Employee



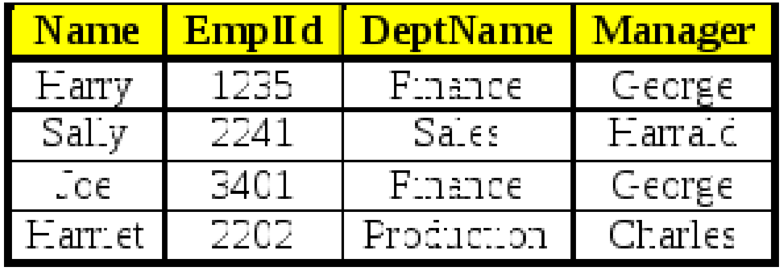
Name	EmplId	DeptName
Harry	1235	Finance
Sally	2241	Sales
Joe	3401	Finance
Harriet	2202	Production

Dept



DeptName	Manager
Finance	George
Sales	Harrald
Production	Charles

Employee ⋈ Dept



Name	EmplId	DeptName	Manager
Harry	1235	Finance	George
Sally	2241	Sales	Harrald
Joe	3401	Finance	George
Harriet	2202	Production	Charles

Division Operation

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

  The result of $r \div s$ is a relation on schema $R - S = (A_1, \ldots, A_m)$

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

  where *tu* means the concatenation of tuples *t* and *u* to produce a single tuple
- 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 $S \subseteq R$

    $$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}\left(\left(\Pi_{R-S}(r) \times s \right) - \Pi_{R-S,S}(r) \right)$$
  - 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 $u \in s$, $tu \notin r$

Division Operation – Example

- Relations *r*, *s*:

A	*B*
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
∈	6
∈	1
β	2

r

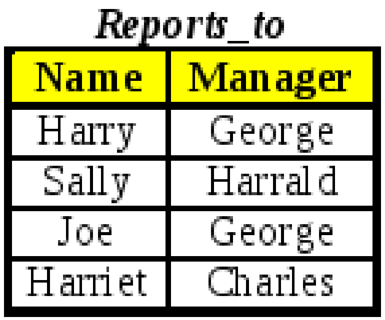
B
1
2

s

$r \div s$:

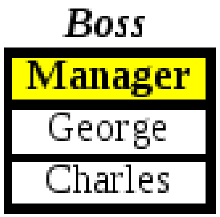
A
α
β

- Practical example



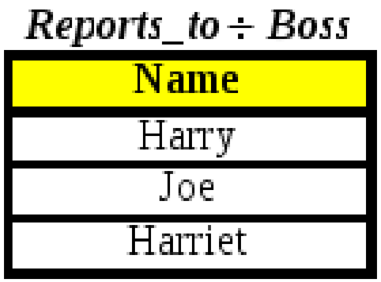
Reports_to

Name	Manager
Harry	George
Sally	Harrald
Joe	George
Harriet	Charles



Boss

Manager
George
Charles



Reports_to ÷ Boss

Name
Harry
Joe
Harriet

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

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

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

$$result = temp1 - temp2$$

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

Bank Example Queries

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

  $\Pi_{customer\_name}\ (borrower) \cap \Pi_{customer\_name}(depositor)$

- Find the name of all customers who have a loan at the bank and the loan amount

  $\Pi_{customer\_name,\ loan\_number,\ amount}\ (borrower \bowtie loan)$

- Find all customers who have an account from at least the “Downtown” and the "Uptown” branches
  - Possibility 1

    $$\Pi_{customer\_name}(\sigma_{branch\_name = \text{“Downtown”}}\ (depositor \bowtie account)) \cap \Pi_{customer\_name}\ (\sigma_{branch\_name = \text{“Uptown”}}\ (depositor \bowtie account))$$

  - Possibility 2

    $$\Pi_{customer\_name,\ branch\_name}\ (depositor \bowtie account) \div \rho_{temp(branch\_name)}\ (\{(\text{“Downtown”}),\ (\text{“Uptown”})\})$$

    - Note, that this version uses a "constant relation"

Extended Relational-Algebra-Operations

- Generalized Projection
- Aggregate Functions
- Outer Join

Generalized Projection

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

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

- E is any relational-algebra expression
- Each of F_1, F_2, …, F_n are are arithmetic expressions involving constants and attributes in the schema of E.
- Is used to compute 'derived' (calculated) attributes
- Given relation

  credit_info(customer_name, limit, credit_balance),

  find how much more each person can spend:

$$\prod_{customer\_name,\ limit - credit\_balance} (credit\_info)$$

Aggregate Functions and Operations

- **Aggregate 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

$$_{G_1, G_2, \ldots, G_n}\vartheta_{F_1(A_1), F_2(A_2, \ldots, F_n(A_n)}(E)$$

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

$\vartheta_{\text{sum}(C)}(r)$:

sum(*c*)
27

- 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

- $_{branch\_name}\vartheta_{\text{sum}(balance)}(account)$:

branch_name	**sum**(*balance*)
Perryridge	1300
Brighton	1500
Redwood	700

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does 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.
    - We shall study precise meaning of comparisons with nulls later

Outer Join – Example

loan

loan_number	branch_name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

borrower

customer_name	loan_number
Jones	L-170
Smith	L-230
Hayes	L-155

Natural join

loan ⋈ *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*

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
- For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same

Null Values (cont.)

- Comparisons with null values return the special logical 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*
- 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
- Examples
  - Delete all account records in the Perryridge branch

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

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

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

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

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

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

  - 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.

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

    $$account \leftarrow account \cup \prod_{loan\_number,\ branch\_name,\ 200}(r_1)$$

    $$\text{depositor} \leftarrow depositor \cup \prod_{customer\_name,\ loan\_number}(r_1)$$

Updating

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

$$r \leftarrow \prod_{F_1, F_2, \ldots, F_l,} (r)$$

- 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
- Examples
  - Make interest payments by increasing all balances by 5 %

$$account \leftarrow \prod_{account\_number,\ branch\_name,\ balance\ *\ 1.05} (account)$$

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

$$account \leftarrow \prod_{account\_number,\ branch\_name,\ balance\ *\ 1.06} (\sigma_{BAL\ >\ 10000} (account\ ))$$
$$\cup \prod_{account\_number,\ branch\_name,\ balance\ *\ 1.05} (\sigma_{BAL\ \leq\ 10000} (account))$$

Lesson 9, part 2
Structured Query Language (SQL)

Create Table Construct

- An SQL relation is defined using the **create table** command:

  `create table` r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$), ...,
  (integrity-constraint$_k$))

  - r is the name of the relation
  - each A_i is an attribute name in the schema of relation r
  - D_i is the data type of values in the domain of attribute A_i
- Integrity constraints in create table
  - **`not null`**
  - **`primary key`**(A_1, ..., A_L)
- Example:

```
create table branch
(     branch_name  char(15) not null,
      branch_city  char(30),
      assets       integer,
      primary key(branch_name)
)
```

- Note:
  - SQL names are case insensitive (i.e., you may use upper- or lower-case letters); e.g.: *Branch_Name* ≡ *BRANCH_NAME* ≡ *branch_name*

Basic Query Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

  select $A_1, A_2, \ldots, A_n$
  from $R_1, R_2, \ldots, R_m$
  where P

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.
- This query is equivalent to the relational algebra expression

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (R_1 ¿ R_2 ¿ \ldots \times R_m))$$

- The result of an SQL query is a relation
- Important remark:
  - SQL is a declarative (query) language while relational algebra is procedural
  - Mapping SQL queries to relational expressions converts declarative queries to procedures
  - Query execution will use procedures implementing relation algebra operations

The `select` clause

- The **`select`** clause lists the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example:
  - Find the names of all branches in the *loan* relation:

    `select` *branch_name* **`from`** *loan*
  - In the relational algebra, the query would be:

    $\prod_{branch\_name}(loan)$
- SQL allows duplicates in relations as well as in query results
  - This violates relational model assumptions but may speed-up processing
- To force the elimination of duplicates, insert the keyword **`distinct`** after **`select`.**
  - Find the names of all branches in the *loan* relations, and remove duplicates

    `select distinct` *branch_name* **`from`** *loan*
  - The keyword **`all`** specifies that duplicates not be removed

    `select all` *branch_name* **`from`** *loan*

The `select` clause (cont.)

- An asterisk in the select clause denotes "all attributes"

  `select` * **`from`** *loan*

- The **`select`** clause can contain arithmetic expressions involving the operation, +, –, $*$, and /, and operating on constants or attributes of tuples
- The query

  `select` *loan_number, branch_name, amount* $*$ 100
  `from` *loan*

  would return a relation that is the same as the *loan* relation, except that the value of the attribute *amount* is multiplied by 100

  - This is, in fact, the generalized projection

    $\Pi_{loan\_number,\ branch\_name,\ amount\ *\ 100}(loan)$

The `where` clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- Example
  - Find all loan numbers for loans made at the Perryridge branch with loan amounts greater than $1200.

```
select loan_number
   from loan
   where branch_name = 'Perryridge'  and amount > 1200
```

- Comparison
  - results can be combined using the logical connectives **and, or,** and **not.**
  - Comparisons may be applied to results of arithmetic expressions.
  - SQL includes a **between** comparison operator
    - Example: Find the loan number of those loans with loan amounts between $90,000 and $100,000 (that is, ≥ $90,000 and ≤ $100,000)

```
select loan_number from loan
       where amount between 90000 and 100000
```

which maps to

$$\Pi_{loan\_number}(\sigma_{(amount \geq 90000) \wedge (amount \leq 100000)}(loan))$$

The `from` clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra
  - Find the Cartesian product *borrower* x *loan*

    select ∗ **from** *borrower, loan*
  - Find the name, loan number and loan amount of all customers having a loan at the Brighton branch

    select *customer_name, borrower.loan_number, amount*
    from *borrower, loan*
    where *borrower.loan_number* = *loan.loan_number* **and**
    branch_name = 'Brighton'

    corresponds to

    $$\Pi_{customer\_name,\ borrower.loan\_number,\ amount}(\sigma_{borrower.loan\_number = loan.loan\_number \wedge branch\_name = 'Brighton'}(borrower \times loan))$$

The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

  old-name **`as`** *new-name*

  - Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id*

```
select customer_name, borrower.loan_number as loan_id, amount
from borrower, loan
where borrower.loan_number = loan.loan_number
```

- Home work:
  - Rewrite this query to relational expression

Tuple Variables

- Tuple variables are defined in the **`from`** clause via the use of the **`as`** clause
- Example
  - Find the customer names and their loan numbers for all customers having a loan at some branch

    `select` *customer_name, T.loan_number, S.amount*
        `from` *borrower* **`as`** *T, loan* **`as`** *S*
        `where` *T.loan_number* = *S.loan_number*
  - Find the names of all branches that have greater assets than some branch located in Brooklyn

    `select distinct` *T.branch_name*
        `from` *branch* **`as`** *T, branch* **`as`** *S*
        `where` *T.assets* > *S.assets* **`and`** *S.branch_city* = 'Brooklyn'

SQL allows duplicates

- **Multiset** versions of some of the relational algebra operators – given multiset relations r_1 and r_2:
  - **$\sigma_\theta(r_1)$:** If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selections $\sigma_{\theta,}$, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
  - **$\Pi_A(r)$:** For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.
  - **r_1 x r_2:** If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 * c_2$ copies of the tuple $t_1 \ldots t_2$ in r_1 x r_2
- Example:
  - Suppose multiset relations r_1 (A, B) and r_2 (C) are as follows:
    $r_1 = \{(1, a)\ (2,a)\}$ $r_2 = \{(2), (3), (3)\}$
  - Then $\Pi_B(r_1)$ would be {(a), (a)}, while $\Pi_B(r_1)$ x r_2 would be
    $\{(a,2), (a,2), (a,3), (a,3), (a,3), (a,3)\}$
- SQL duplicate semantics:
  - **`select`** $A_1, A_2, \ldots, A_n$ **`from`** $r_1, r_2, \ldots, r_m$ **`where`** P

is equivalent to the *multiset* version of the expression:

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (r_1 ¿ r_2 ¿ \ldots \times r_m))$$

Set Operations

- The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations ∪, ∩, −
- Find all customers who have a loan, an account, or both:

  (**select** *customer_name* **from** *depositor*)
  union
  (**select** *customer_name* **from** *borrower*)

- Find all customers who have both a loan and an account:

  (**select** *customer_name* **from** *depositor*)
  intersect
  (**select** *customer_name* **from** *borrower)*

- Find all customers who have an account but no loan

  (**select** *customer_name* **from** *depositor*)
  except
  (**select** *customer_name* **from** *borrower*)

Aggregate Functions in SQL

- These functions operate on the multiset of values of a column of a relation, and return a value

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

- Find the average account balance at the Perryridge branch

```
select avg (balance)
        from account
        where branch_name = 'Perryridge'
```

- Find the number of depositors in the bank

```
select count (distinct customer_name)
        from depositor
```

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 predicate **`is null`** can be used to check for null values.
  - Example: Find all loan number which appear in the *loan* relation with null values for *amount.*

    `select` *loan_number* **`from`** *loan*

    `where` *amount* **`is null`**
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- However, aggregate functions simply ignore nulls
- Any comparison with *null* returns *unknown*
  - Example*: 5 < null or null <> null or null = null*
- Three-valued logic using the truth value *unknown* is the same as above for relations
- "*P* **`is unknown`**" evaluates to true if predicate *P* evaluates to *unknown*

Nested Subqueries

- SQL provides a mechanism for the nesting of queries
- A **subquery** is a **`select-from-where`** expression that is nested within another query
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality
- Example:
  - Find all customers who have both an account and a loan at the bank

```
select distinct customer_name
        from borrower
        where customer_name in (select customer_name
                                 from depositor )
```

Views

- In some cases, it is not desirable for all users to see the entire logical model
  - that is, all the actual relations stored in the database
- Consider a person who needs to know a customer’s name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in SQL, by

  (**`select`** *customer_name, borrower.loan_number, branch_name*
  `from` *borrower, loan*
  `where` *borrower.loan_number = loan.loan_number*)
- A **view** provides a mechanism to hide certain data from the view of certain users.
  - Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.
- A view is defined using the **`create view`** statement which has the form

  `create view` *v* **`as`** <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

Modification of the Database

- Deletion
  - Statement is **`delete-from-where`** with the arguments similar to the **`select-from-where`** construct
  - Delete all account tuples at the Brighton branch

    `delete from` *account* **`where`** *branch_name* = ‘Brighton‘
- Insertion
  - Statement is: **`insert into`** relation **`values`** <compatible_relation>
  - Add a new tuple to *account*

    `insert into` *account* (*branch_name, balance, account_number*)
    `values` ('Perryridge', 1200, 'A-9732')
- Updates
  - Statement is: **`update`** relation **`set`** attribute = expression **`where`** condition
  - Add 6% to accounts over $1000

    `update` *account* **`set`** *balance* = *balance*∗1.06 **`where`** *balance*>1000

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **`from`** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.
- Completely based on relational-algebra joins. SQL syntax described in the SQL standards
- Example
  - Find all customers who have either an account or a loan (but not both) at the bank

    `select` *customer_name*

    `from` (*depositor* **`full outer join`** *borrower*)

    `where` *account_number* **`is null or`** *loan_number* **`is null`**

End of Lesson 10

Questions?