DATABASE SYSTEMS

COURSE INFORMATION

What is important:

- being able to apply your knowledge to relevant examples
- being able to be precise about the key concepts of database systems

Lectures are not streamed and no podcasts are provided, but last year's recordings are available.
Exam questions: multiple choice, single choice, know concepts precisely.
Half-open text (write queries) and half single/multiple choice
Final Exam: 27.6.2023, 08:00-12:00 (90 minutes BYOD). Closed book without auxiliary material.
Frequent office hours, do exercises and practice papers.

NOTATIONAL CONVENTIONS

Relational Algebra (RA):

- **constant**: 'abc', 14, 3.14, ...
- **attribute**: *Name*, *X*, ... (upper case)
- **relation name**: *Employee*, *R*, ... (upper case)
- **schema**: *Employee*(*Name*, *Addr*), ...
- **tuple**: t, t_1, ... (lower case)
- **relation**: *emp*, *r*, *s*, ... (lower case)
- **database**: *D*, *DB*, ... (upper case)

Domain Relational Calculus (DRC), First Order Predicate Logic (FOPL):

- **constant**: 'abc', 14, ...
- **variable**: *X*, *Y*, ... (upper case)
- **predicate**: *p*, *q*, ... (lower case)

SQL:

- **constant**: 'abc', 14, ...
- **SQL keyword**: **SELECT**, **FROM**, ... (all cap, blue, bold)
- **attribute**: *Name*, *Salary*, ... (upper case)
- **table name**: *R*, *Dept*, ... (upper case)
- **schema**: $R(A, B)$, $Employee(Name, Addr)$, ...
- **table**: *r*, *dept*, ... (lower case)

Entity Relationship Model (ER Model): → predecessor of UML

- **constant**: 'abc', 14, ...
- **attribute**: *Name*, *Gender*, ... (green, upper case)
- **relationship**: *workFor*, ... (red, lower case)
- **entity**: *Company*, *Emp*, ... (blue, upper case)

DATABASE SYSTEMS

Basic Definitions

Data: facts that can be recorded

Information: data + meaning

Knowledge: information + application

Database (DB): collection of related data

Database Management System (DBMS): software package to facilitate the creation, maintenance, and querying of databases

Database System: DB + DBMS -> data + software for data

Meta Data: information about the structure of the DB (which tables, columns, rows, users). Meta Data is organized as a DB itself.

DBMS Languages

A DBMS offers two types of languages:

- data definition language (DDL) to create and drop tables
- data manipulation language (DML) to select, insert, delete, and update data

The standard language for database systems is SQL.

SQL offers a DDL and a DML.

Distinction between:

- High level (declarative, non-procedural) languages: They are set-oriented (retrieve multiple results) and specify what data to retrieve. Examples: SQL, Google Search, borrow a book from the UZH
- Low level (procedural) languages: they retrieve data one record at a time, and specify how to retrieve data. Examples: cooking recipe, programming languages

Database Schema: description of a database, includes description of the database structure, data types, and its constraints -> top row (header of the DB)

Database Instance: the actual data stored in a database at a particular moment in time -> collection of all data in the database

Schema remains the same (fixed), instance can vary.

Redundancy: is present if information is stored multiple times. A key goal of database design is to minimize it, since it leads to anomalies an inconsistent data

Controlled Redundancy: duplication of information is allowed and controlled by the DBMS

THE RELATIONAL MODEL

A relation schema R, denoted by R(A1, A2, ..., An), is made up of a relation name and a list of attributes. attr(R) denotes the set of attributes of the relation with name R.

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 required to be atomic (indivisible) -> the value of an attribute can be an account number, but cannot be a set of account numbers

The attribute name designates the role played by a domain in a relation: it is used to interpret the meaning of the data elements corresponding to that attribute

The special value null is a member of every domain

A tuple is an ordered set (list) of values. Angular brackets are used as notation

r(R) denotes a relation (or relation instance) r over relation schema with name R

A relation instance is a subset of the Cartesian product of the domains of its attributes. Thus, it is a set of n-tuples, where n is the number of attributes in the domain.

Relations are unordered, this means that the order of tuples is irrelevant.

Examples

1. Is $r = \{(\text{'Tom'}, 27, \text{'ZH'}), (\text{'Bob'}, 33, \text{'Rome'}, \text{'IT'})\}$ a relation?

   No, for first tuple we miss city entry ⇒ schemas are different (not allowed)

2. For $r = \{(1, \text{'a'}), (2, \text{'b'}), (3, \text{'c'})\}$ and $sch(r) = R(X, Y)$ determine:

   - the 2nd attribute of relation r? Y

X	Y
1	'a'
2	'b'
3	'c'

   - the 3rd tuple of relation r? not exist (there is no order!)
   - the tuple in r with the smallest value for attribute X? (1,'a')

3. What is the difference between a set and a relation? Illustrate with an example.

   Every relation is a set but not every set is a relation

   In a set elements can be anything and in a relation elements are tuples with same schema

   $X = \{\underbrace{\text{'a'}}_{\text{atom}}, \underbrace{(\text{'a'}, 1)}_{\text{tuple}}, \underbrace{\{1,2,3\}}_{\text{set}}\} \Rightarrow$ set (w 3 elements)

A database consists of multiple relations.

Summary of the Relational Data Model

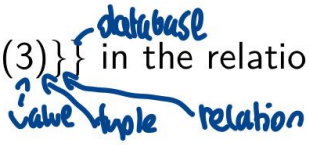
- A **domain** D is a set of atomic data values.
  - phone numbers, names, grades, birthdates, departments
  - each domain includes the special value `null`
- With each domain a **data type** or format is specified.
  - 5 digit integers, yyyy-mm-dd, characters
- An **attribute** A_i describes the role of a domain in a relation schema.
  - PhoneNr, Age, DeptName
- A **relation schema** $R(A_1, ..., A_n)$ is made up of a relation name R and a list of attributes.
  - $Employee(Name, Dept, Salary)$, $Department(DName, Manager, Address)$
- A **tuple** t is an ordered list of values $t = (v_1, ..., v_n)$ with $v_i \in dom(A_i)$.
  - $t = (\text{'Tom'}, \text{'SE'}, 23K)$
- A **relation** $r \subseteq D_1 \times ... \times D_n$ over schema $R(A_1, ..., A_n)$ is a set of n-ary tuples.
  - $r = \{(\text{'Tom'}, \text{'SE'}, 23K), (\text{'Lene'}, \text{'DB'}, 33K)\} \subseteq Names \times Departments \times Integer$
  - $s = \{(\text{'SE'}, \text{'Tom'}, \text{'Boston'}), (\text{'DB'}, \text{'Lena'}, \text{'Tucson'})\}$
- A **database** DB is a set of relations.
  - $DB = \{r, s\}$

Examples

1. Illustrate the following relations graphically:
   $r = \{(1, \text{'a'}), (2, \text{'b'}), (3, \text{'c'})\}$, $sch(r) = R(X, Y)$;
   $s = \{(1, 2, 3)\}, sch(s) = S(A, B, C)$

2. What kind of object is $X = \{\{(3)\}\}$ in the relational model?

   X is a database



3. Are DB1 and DB2 identical databases?
   $DB1 = \{\{(1, 5), (2, 3)\}, \{(4, 4)\}\}$
   $DB2 = \{\{(4, 4)\}, \{(2, 3), (1, 5)\}\}$

   Yes, DB is set of relations. Relation is a set of tuples. No ordering.

A constraint is a condition that must be satisfied by all valid relation instances (tuples). There are four main types of constraints in the relational model:

- Domain constraints: each value in a tuple must be from the domain of its attribute
- key constraints
- Entity constraints
- Referential Integrity constraints

Key Constraints

Superkey: A superkey is a set of one or more attributes that can uniquely identify a tuple in a relation. A superkey does not have to be minimal, meaning that removing one or more attributes from it can still result in a unique identifier for the tuple. Examples of real-life superkeys include the AHV-number and Immatriculation Number. Superkey is a superset of candidate key.

Candidate key: A candidate key is a minimal superkey, meaning that no subset of the candidate key can still uniquely identify a tuple in the relation. A relation can have multiple candidate keys, all of which are minimal. Example UZH: shortname, immatriculation number, email address, AHV-Number.

Primary key: A primary key is a candidate key that has been chosen as the main identifier for tuples within a relation. Ideally, a primary key should be based on attributes that never or very rarely change in value to ensure stable and reliable identification of tuples. Example UZH: shortname.

For a relation there can be more than one (many) candidate key, but only one primary key is allowed.

Entity Constraints

The entity constraint requires that the primary key attributes of each relation may not have null values. Other attributes of the relation may also disallow null value although they are not members of the primary key.

Foreign key: attribute that corresponds to the primary key of a relation. Only values occurring in the primary key attribute of the referenced relation (or null values) may occur in the foreign key attribute of the referencing relation.

Example

1. Determine the candidate keys of relation r:

r(R)

X	Y	Z
1	2	3
1	4	5
2	2	2

- Z could be a candidate key (we don't know about semantics)
- X and Y not
- XY could be
- Any superset of Z and XY could be (e.g XYZ), only if subset is not a candidate key
  if Z is candidate ZY cannot be candidate, because a candidate key is minimal.

Database cant have any duplicates! it contains a set!

In practice DB contain tables and not sets, so there may be duplicates in it. Ignore according to theory!

Determine possible superkeys of relations r and s. Assume that the possible superkeys indeed are superkeys: determine candidate, and possible primary and foreign keys.

r(R)

A	B	C
'a'	'd'	'e'
'b'	'd'	'c'
'c'	'e'	'e'

s(S)

D	E
'd'	'a'
'e'	'a'
'a'	'a'

possible superkeys: A, AB, AC, ABC, BC, D, DE

candidate keys: A, BC, D

possible primary keys: → can only choose 1 per relation
A for R, D for S

possible foreign keys: E with primary key A, B with primary key D
E with primary key D,

Relational algebra is a procedural language (order of operations matters).

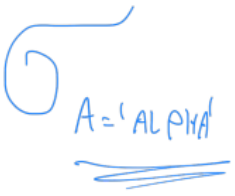
- The relational algebra consists of six basic operators
  - select: σ
  - project: π
  - union: $\cup$
  - set difference: $-$
  - cartesian product: $\times$
  - rename: ρ

The operators take one or two relations as input and produce a new relation as a result. This property makes the algebra closed -> all objects in the relational algebra are relations

Select Operation

Notation: $\sigma_p(r)$, where p is called the selection predicate

The select operation is used to select rows from a table



A	B	C
Alpha	14	X
Beta	15	X
Gamma	2	Y
Theta	6	Y

Project Operation

Used to select specific columns from a table. There are no duplicate rows in the result since relations are sets.

$\Pi_A(R)$

A	B	C
Alpha	14	X
Beta	15	X
Gamma	2	Y
Theta	6	Y

Union Operation

Notation: r U s

For r U s to be valid r and s must be union compatible. Result contains no duplicates, and includes all tuples that are either in R, or in S, or in both R and S.

A	B
Alpha	14
Beta	15
Gamma	2

A	B
Alpha	14
Beta	7
Gamma	2
Theta	6

BETA 7

THETA 6

Set Difference Operation

Notation: r - s

Set differences must be taken between (union) compatible relations. R and s must have the same arity (number of columns in the table).

Attribute domains of r and s must be compatible. The set difference operation between two relations returns the tuples that are in R but not in S. R-S is not the same as S-R.

A

A	B
Alpha	14
Beta	15
Gamma	2

B

A	B
Alpha	14
Beta	7
Gamma	2
Theta	6

A - B

Cartesian Product Operator

Notation: r x s

The attribute names of r and s must be disjoint (different), otherwise a naming conflict exists, To prevent naming conflicts, renaming must be used.

A	B
Alpha	1
Beta	2

X

D
X
Y

A	B	C
Alpha	1	X
Alpha	1	Y
Beta	2	X
Beta	2	Y

Rename Operation

Allows us to name the result of relational algebra expressions by setting relation and attribute names. This operator is also used if there are name clashes.

Various flavors:

- $\rho_r(E)$ changes the relation name to r.
- $\rho_{r(A_1,\ldots,A_n)}(E)$ changes the relation name to r and the attribute names to $A_1, \ldots, A_k$.
- $\rho_{(A_1,\ldots,A_n)}(E)$ changes attribute names to $A_1, \ldots, A_k$.

Example: $\rho_{s(X,Y,U,V)}(r)$

r

A	B	C	D
'α'	'α'	1	7
'β'	'β'	23	10

s

X	Y	U	V
'α'	'α'	1	7
'β'	'β'	23	10

Examples

Identify and correct syntactic mistakes in the following relational algebra expressions. The schema of relation r is $R(A, B)$.

$\sigma_{r.A>5}(r)$ not allowed to use r.A (not an attribute name) fix: $\sigma_{A>5}(r)$

$\sigma_{A,B}(r)$ No selection predicate. fix: $\Pi_{A,B}(r)$

fix 2: $\sigma_{A>B}(r)$

$r \times r$ attribute names are ABAB many times -> naming conflicts

give relation the name T

fix $S_T(r \times S_S(C,D)(r))$

Identify and correct syntactic mistakes in the following relational algebra expressions. Relation *pers* has schema *Pers*(*Name*, *Age*, *City*).

$\sigma_{Name='Name'}(pers)$ ✓

$\sigma_{City='Zuerich'}(pers)$ → not an attribute

There is no attribute named Zürich

$\sigma_{Age>'20'}(pers)$ → is an integer

Banking Example

- *Branch*(*BranchName*, *BranchCity*, *Assets*)
- *Customer*(*CustName*, *CustStreet*, *CustCity*)
- *Account*(*AccNr*, *BranchName*, *Balance*)
- *Loan*(*LoanNr*, *BranchName*, *Amount*)
- *Depositor*(*CustName*, *AccNr*) — account owner
- *Borrower*(*CustName*, *LoanNr*) — loan owner

1) Find all loans larger than $1200

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

2) find the loan number for each loan that is larger than $1200

$\Pi_{loanNr}(\sigma_{Amount>1200}(loan))$

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

$\Pi_{custName}(borrower) \cup \Pi_{custName}(Depositor)$

duplicates set eliminated

- Give a different relational algebra expressions that determines the names of all customers who have a loan at the Perryridge branch. Compare it to the solution in Review 2.5/2.

$\Pi_{custName}(\sigma_{LNr=loanNr}($

$\sigma_{BranchName='Perryridge'}(loan) \times \rho_{(custName,LNr)}^{(Borrower)}))$

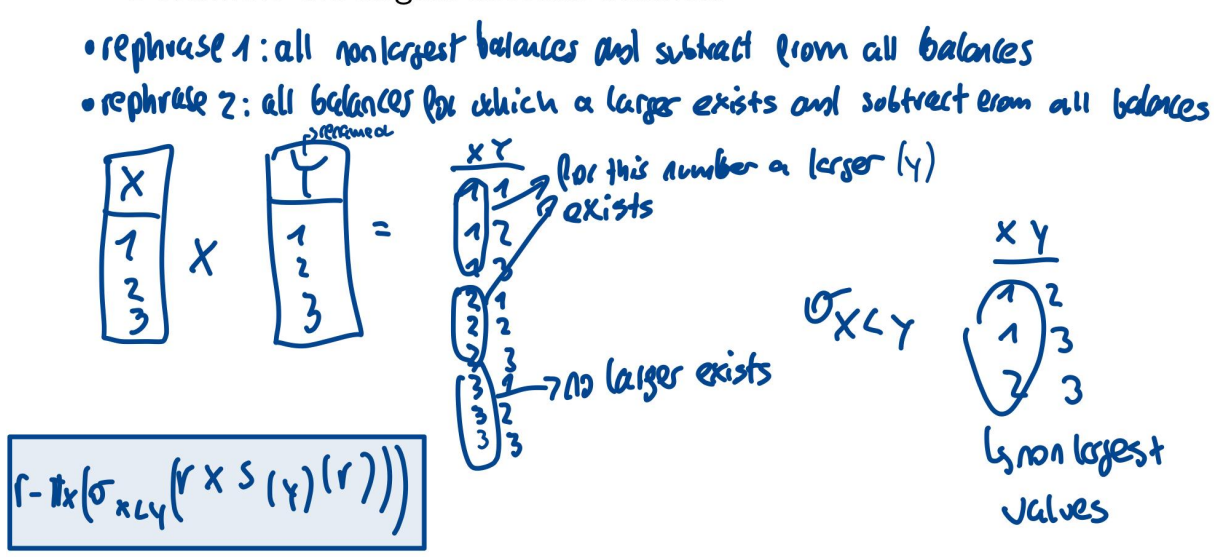
- Names of all customers who have a loan at the Perryridge branch.

$\Pi_{custName}(\sigma_{branchName='Perryridge'}(\sigma_{LNr=loanNr}(\rho_{(custName,LNr)}(Borrower) \times \underbrace{loan}_{①})))$

- 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.

① $- \Pi_{custName}(depositor)$

- Determine the largest account balance.



Intersection Operator

Notation: $\cap$

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

The intersection operation in relational algebra is used to find common tuples between two relations. For the intersection operation to be valid, both relations must have the same set of attributes (arity) and compatible domains (i.e., the same data types) -> the two relations need to be union compatible

A

A	B
Alpha	14
Beta	15
Gamma	2

B

A	B
Alpha	14
Beta	7
Gamma	2
Theta	6

A ∩ B

ALPHA 14

GAMMA 2

Division Operator

Notation: ÷

Used to find tuples in one relation that are associated with all tuples in another relation. It is suited for queries that include the phrase "all".

Assignment Operator

Notation: <-

Used to express complex queries by breaking them up into smaller pieces. The assignment must always be made to a temporary relational variable which is declared on the left of the <- symbol

Natural Join Operator

Notation: ⋈

Used to combine tuples from two relations R and S where they have matching values for all common attributes. Can be performed only if there is a common attribute in the 2 relations (same name, if needed

rename). The resulting relation contains all attributes from both R and S but retains only one copy of each common attribute.

A	B	C
Alpha	14	X
Beta	15	X
Gamma	2	Y
Theta	6	Y

D	C
Ciao	X
Hi	X
Hola	Y
Gruezi	Y

A	B	C	D
Alpha	14	X	Ciao
Beta	15	X	Ciao
Gamma	2	Y	Hola
Theta	6	Y	Hola
Alpha	14	X	Hi
Beta	15	X	Hi
Gamma	2	Y	Gruezi
Theta	6	Y	Gruezi

Theta Join Operator

Notation: $\bowtie_\theta$

Used to combine tuples from two relations R and S based on a specific condition (θ) involving their attributes. The resulting relation contains all attributes from both R and S.

A	B	C
Alpha	14	X
Beta	15	X
Gamma	2	Y
Theta	6	Y

D	E
10	X
16	X
1	Y
5	Y

A	B	C	D	E
Alpha	14	X	10	X
Beta	15	X	10	X
Gamma	2	Y	1	Y
Theta	6	Y	1	Y
Theta	6	Y	5	Y

Formal Definition of Relational Algebra Expressions

A basic expression in the relational algebra consists of either a relation in the database or a constant relation (e.g., {(1, 2), (5, 3)})

Examples

- Find all customers who have an account and a loan.

  $\pi_{CustName}(borrower) \cap \pi_{CustName}(depositor)$

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

  $\pi_{CustName,Amount}(borrower \bowtie loan)$

- Find all customer names who have an account from at least the "Downtown" and the "Uptown" branches.
  - Solution 1

    $\pi_{CustName}(\sigma_{BranchName='Downtown'}(depositor \bowtie account)) \cap$
    $\pi_{CustName}(\sigma_{BranchName='Uptown'}(depositor \bowtie account))$
  - Solution 2

    $r \leftarrow \pi_{CustName,BranchName}(depositor \bowtie account))$
    $s \leftarrow \pi_{BranchName}(\sigma_{BranchName='Downtown' \vee BranchName='Uptown'}(account))$
    $Res \leftarrow r \div s$

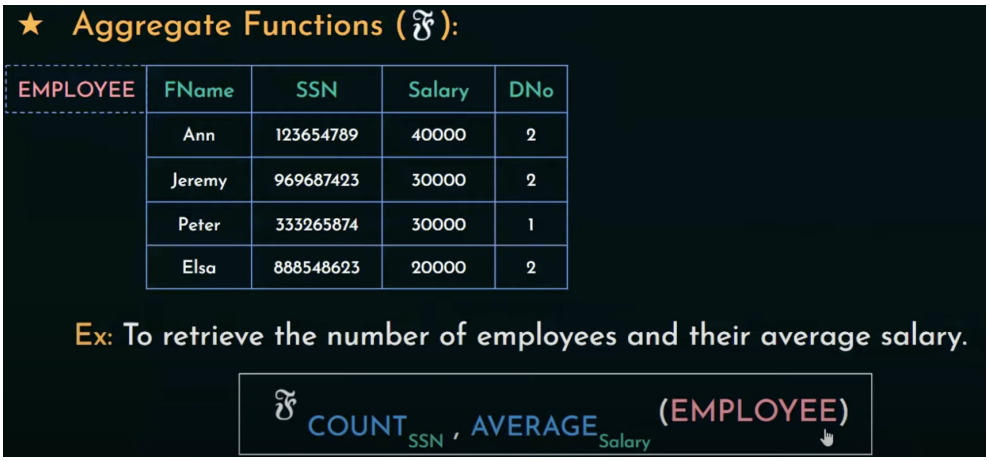
- Find all names of customer names who have an account at all branches located in Brooklyn city.

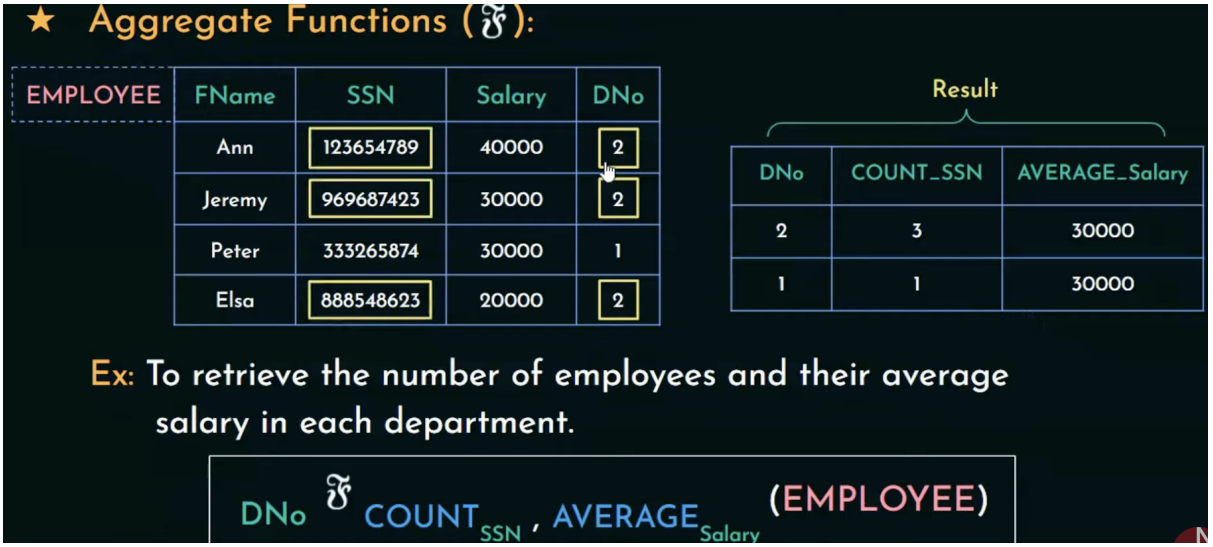
  $\Pi_{CustName, BranchName}(acc \bowtie depos) \div$
  $\Pi_{BranchName}(\sigma_{BranchCity='Brooklyn'}(Branch))$

Extended Relational Algebra Operators

Generalized Projection: extends the projection operation by allowing arithmetic functions to be used in the projection list

Aggregate Function: takes a collection of values and returns a single value as the result. Some examples: avg, min, max, sum, count





Aggregation Operation

$$G_1, G_2, \ldots, G_n \vartheta F_1(A_1), F_2(A_2), \ldots, F_n(A_n)(E)$$

E is any relational algebra expression

- $G_1, G_2 \ldots, G_n$ 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

Outer Join: 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

It uses null values: null signifies that the value is unknown or does not exist. All comparisons involving null are false by definition. Inner join (join) leads to loss of data, outer join can be used when we want all tuples from r, all tuples of s, or all tuples from both relations r and s to be displayed even if the tuples do not match (avoids loss of information). **Left outer join:** keeps all tuples from left relation and only those who match condition from the right relation (fill with null values)

Right outer join: keeps all tuples from right relation and only those who match condition from the left relation (fill with null values)

Full outer join: keeps both tuples from both relations and those who do not satisfy condition are filled with null values

Examples

- Example relations:

loan

LoanNr	BranchName	Amount
'L-170'	'Downtown'	3000
'L-230'	'Redwood'	4000
'L-260'	'Perryridge'	1700

borrower

CustName	LoanNr
'Jones'	'L-170'
'Smith'	'L-230'
'Hayes'	'L-155'

- **Join** (=inner join) return a tuple if loanNr is in both tables

loan ⋈ *borrower*

LoanNr	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'

- Example relations:

loan

LoanNr	BranchName	Amount
'L-170'	'Downtown'	3000
'L-230'	'Redwood'	4000
'L-260'	'Perryridge'	1700

borrower

CustName	LoanNr
'Jones'	'L-170'
'Smith'	'L-230'
'Hayes'	'L-155'

- **Left Outer Join** (preserves tuples from left)

loan ⟕ *borrower*

LoanNr	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'
'L-260'	'Perryridge'	1700	*null*

- Example relations:

loan

LoanNr	BranchName	Amount
'L-170'	'Downtown'	3000
'L-230'	'Redwood'	4000
'L-260'	'Perryridge'	1700

borrower

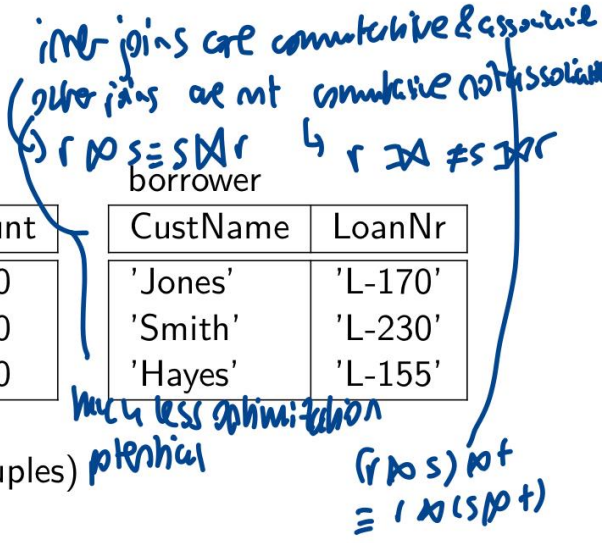
CustName	LoanNr
'Jones'	'L-170'
'Smith'	'L-230'
'Hayes'	'L-155'

- **Right Outer Join** (preserves tuples from right)

loan ⟖ *borrower*

LoanNr	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'
'L-155'	*null*	*null*	'Hayes'

Outer Join Example/4



- Example relations:

loan

LoanNr	BranchName	Amount
'L-170'	'Downtown'	3000
'L-230'	'Redwood'	4000
'L-260'	'Perryridge'	1700

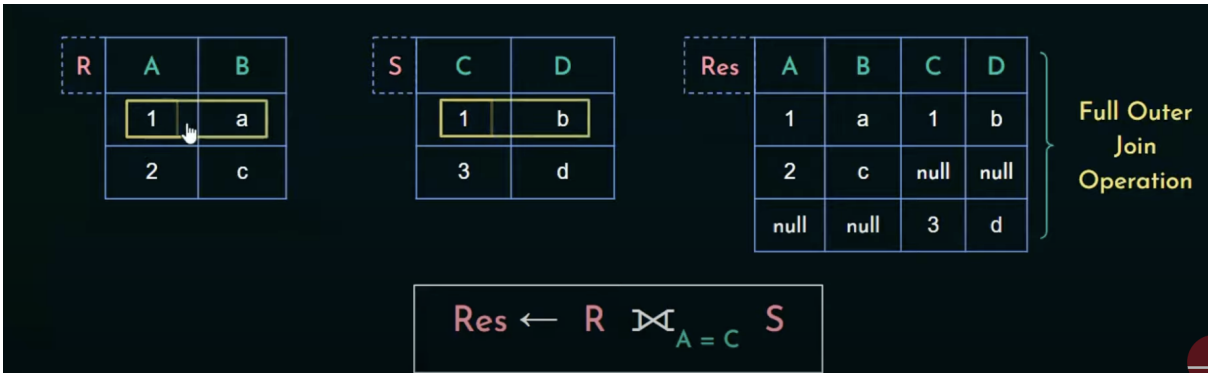
borrower

CustName	LoanNr
'Jones'	'L-170'
'Smith'	'L-230'
'Hayes'	'L-155'

- **Full Outer Join** (preserves all tuples)

loan ⟗ *borrower*

LoanNr	BranchName	Amount	CustName
'L-170'	'Downtown'	3000	'Jones'
'L-230'	'Redwood'	4000	'Smith'
'L-260'	'Perryridge'	1700	*null*
'L-155'	*null*	*null*	'Hayes'



Modification of the Database

Deletion

Instead of displaying tuples to the user similarly to a query, a delete request removes the selected tuples from the database. A deletion is expressed in relational algebra by: r <- r - E, where r is a relation and E is a relational algebra query.

Example

Deletion Examples

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

- Delete all account records in the Perryridge branch.

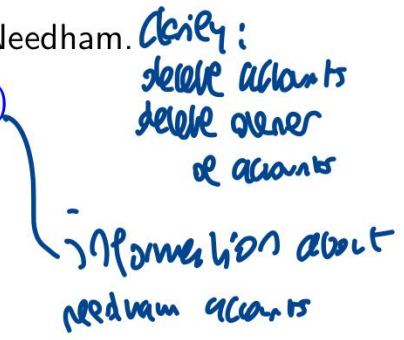
  $accout \leftarrow account - \sigma_{BranchName='\text{Perryridge}'}(account)$

- Delete all loan records with amount in the range of 10 to 50

  $loan \leftarrow loan - \sigma_{Amount \geq 10 \wedge Amount \leq 50}(loan)$

- Delete all accounts at branches located in Needham.

  $r_1 \leftarrow \sigma_{BranchCity='\text{Needham}'}(accout \bowtie branch)$
  $r_2 \leftarrow \pi_{AccNr,BranchName,Balance}(r_1)$
  $r_3 \leftarrow \pi_{CustName,AccNr}(r_2 \bowtie depositor)$
  $account \leftarrow account - r_2$
  $depositor \leftarrow depositor - r_3$



Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. An insertion is expressed in relational algebra by: r <- r U 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

Examples

Insertion Examples

Branch(BranchName, BranchCity, Assets)
Customer(CustName, CustStreet, CustCity)
Account(AccNr, BranchName, Balance)
Loan(LoanNr, BranchName, Amount)
Depositor(CustName, AccNr)
Borrower(CustName, LoanNr)

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

  tuple constants

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

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

- 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.

  $r_1 \leftarrow \sigma_{BranchName = \text{'Perryridge'}}(borrower \bowtie loan)$

  $account \leftarrow account \cup \pi_{LoanNr, BranchName, 200}(r_1)$

  $depositor \leftarrow depositor \cup \pi_{CustName, LoanNr}(r_1)$

Updating

It is a mechanism to change a value in a tuple without changing all values in the tuple. it is faster than deleting + inserting, because of memory management. An update is expressed in relational algebra by: r <- r U E, where r is a relation and E is a relational algebra expression.

overwriting entire relation r

$$r \leftarrow E$$

$$r \leftarrow \pi_{F_1, F_2 \ldots, F_i, \ldots}(r)$$

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, which defines the new value for the attribute

Examples

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

  $account \leftarrow \pi_{AccNr,BranchName,Balance*1.05}(account)$

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

  $account \leftarrow$

  $\pi_{AccNr,BranchName,Balance*1.06}(\sigma_{Balance>10000}(account))$

  $\cup$

  $\pi_{AccNr,BranchName,Balance*1.05}(\sigma_{Balance\leq 10000}(account))$

Relational Algebra in Text Terminals

Standard RA Notation	Typewriter Notation
$\neq, \geq, \leq$	`!=, >=, <=`
$\wedge, \vee, \neg$	`&&, ||, !`
σ, π, $\cup$, $\times$, ρ, $\cap$, $\div$, ϑ	`:S, :P, :U, :C, :R, :I, :D, :A`
$\bowtie_\theta$, $\ltimes_\theta$, $\rtimes_\theta$, $\ltimes\!\!\!\rtimes_\theta$, $\triangleright$	`:TJ, :LTJ, :RTJ, :FTJ, :AJ`
$\leftarrow$	`<-`

- Instead of subscripts we use `[]` brackets.
- Replace `T` with `N` for natural joins (instead of theta joins).
- Example:
  - ${}_{A,B}\vartheta_{SUM(C)}(r) \bowtie_{B\neq Y} s$
  - `[A,B]:A[SUM(C)](r) :TJ[B != Y] s`

Relational Calculus

A relational calculus expression creates a new relation, which is specified in terms of variables that range over attributes of relations (domain relational calculus (DRC)).

In a relational calculus expression, there is no order of operations to specify how to compute the query result. A calculus expression

specifies only what information the result should contain; hence it is a non-procedural (declarative) language. Relational calculus is closely related to and a subset of first order predicate logic (FOPL).

First Order Predicate Logic

Syntax: defines what is syntactically correct/wrong

- **logical symbols**: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\exists$, $\forall$, ...
- **constant**: string, number, ...; 'abc', 14, ...
- **identifier**: character sequence starting with a letter
- **variable**: identifier starting with capital letter; X, Y, ...
- **predicate symbol**: identifier starting with lower case letter
- **build-in predicate symbol**: $=, <, >, \leq, \geq, \neq$, ...
- **term**: constant, variable
- **atom**: predicate, built-in predicate; $p(t_1, ..., t_n)$, $t_1 < t_2$, ... with terms $t_1, ..., t_n$; predicate symbol p
- **formula**: atom, $A \wedge B$, $A \vee B$, $\neg A$, $A \Rightarrow B$, $\exists XA$, $\forall XA$, (A), ... with formulas A, B; variable X

Examples

Decide which of the following formulas are syntactically correct first order predicate logic formulas.

- $less\_than(99, 27)$ → terms (constants) OK
  ↳ predicate
- $loves(mother(\text{'hans'}), france \vee italy)$ wrong
  ↳ operation (cannot be terms) → disjunction is not a term
- $\forall X(danish(X) \Rightarrow danish(\text{'bill\_clinton'}))$ wrong (with parenthesis OK)
- $\forall P(P(\text{'hans'}))$ variable
  ↳ must be predicate → not allowed to place variable in place of predicate
- $\forall C(neighbour(\text{'england'}, C))$ OK
- $\exists C(neighbour(\text{'italien'}, C))$ OK → there exists a C such that italy is a neighbour of C.
- $\forall P(smart(P) \wedge \neg alive(P) \Rightarrow famous(P))$ OK

Your understanding of the bound variable is right. If the variable is not contained in the results, it should be quantified using ∀ or ∃. They should contain all ranges in which the variable occurs.

For example, we have three variables X, Y, Z in table r.

if we want to select all X and Y when Z>5, the DRC query would be: { X, Y | ∃Z(r(X, Y, Z) ∧ Z > 5) }

if we want to select X, Y, and Z when Z>5, the query would be: { X, Y, Z | r(X, Y, Z) ∧ Z > 5 }

I hope this helps.

Selected Properties and Terminology

Terminology

- A variable is **free** if it is not quantified
- A variable is **bound** if it is quantified
- Example: $\forall X(p(X, Y, Y)) \wedge q(X, Y)$ ≡ $\forall x (p(X,Y,Y)) \wedge q(Z,Y)$
  scope of x
  bound free free

FOPL Equivalences

- $\forall X(A) = \neg\exists X(\neg A)$
- $A \Rightarrow B = \neg A \vee B$
- $A \wedge \exists X(B) = \exists X(A \wedge B)$ if X is not free in A
- $A \wedge \forall X(B) = \forall X(A \wedge B)$ if X is not free in A
- $A \wedge \neg\exists X(B) = A \wedge \neg\exists X(A \wedge B)$ if X is not free in A

guideline: make the scope of quantifiers as small as possible (→ "miniscope")

Set theory

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

Domain Independence

Relational calculus only permits expressions that are domain independent, i.e., expressions that permit sensible answers (no infinite results).

Examples

Use first order predicate calculus expressions to express the following natural language statements:

- Anyone who is dedicated can learn databases.

  $\forall x(\text{dedicated}(x) \Rightarrow \text{canlearn}(x,'\text{db}'))$

  If we put $\wedge$ it means that all x in the world, x is dedicated and can learn databases. x is not true, e.g. if x is a dog

- No man is independent.

  $\neg\exists x(\text{man}(x) \wedge \text{indep}(x))$

- Dogs that bark do not bite.

  All is not stated

  $(\text{dog}(x) \wedge \text{barks}(x)) \Rightarrow \neg\text{bite}(x)$

Review 2.9/2

- Not all men can walk.

  $\neg\forall x(\text{man}(x) \Rightarrow \text{walks}(x))$ or ($\equiv$) $\equiv \exists x \neg(\neg\text{man}(x) \vee \text{walks}(x))$ $\equiv \exists x(\text{man}(x) \wedge \neg\text{walks}(x))$

  $\exists x(\text{man}(x) \wedge \neg\text{walks}(x)) \equiv \neg\neg\exists x \neg(\text{man}(y) \Rightarrow \text{walks}(x))$

- Every person owns a computer.

  $\forall x(\text{pers}(x) \Rightarrow \exists y(\text{comp}(y) \wedge \text{owns}(x,y)))$

- Lars likes everyone who does not like ~~himself~~ themselves.

  $\forall x(\neg\text{likes}(x,x) \Rightarrow \text{likes}('\text{lars}',x))$

DRC is relationally complete (it has same expressiveness as RA and SQL).

Syntax:

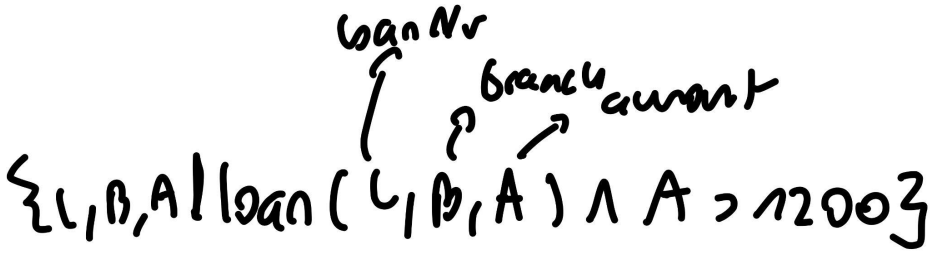
- **logical symbols**: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\exists$, $\forall$, ...
- **constant**: string, number, ...; 'abc', 14, ...
- **identifier**: character sequence starting with a letter
- **variable**: identifier starting with capital letter; X, Y, ...
- **predicate symbol**: identifier starting with lower case letter
- **build-in predicate symbol**: $=, <, >, \leq, \geq, \neq$, ...
- **term**: constant, variable
- **atom**: predicate, built-in predicate; $p(X, ..., 22)$, $X < 5000$, ...
- **formula**: atom, $A \wedge B$, $A \vee B$, $\neg A$, $A \Rightarrow B$, $\exists X\, A$, $\forall X\, A$, (A), ...

- A domain relational calculus query is of the form $\{ X_1, \ldots, X_n \mid \textit{formula} \}$, where $X_1, \ldots, X_n$ are the only free variables in *formula* Yields a result relation

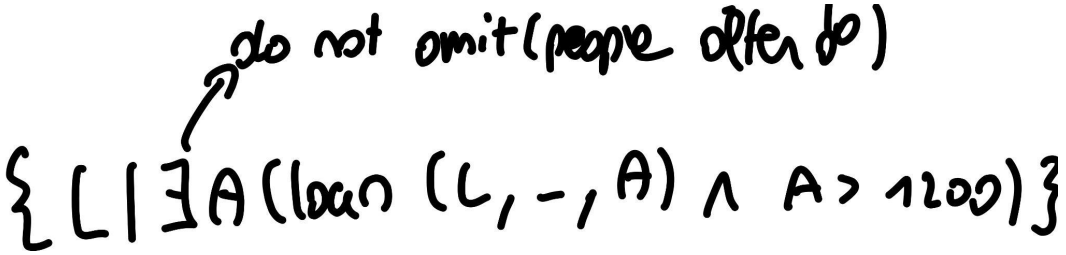
In the domain calculus the position of attributes is relevant. Attribute names are not used.

Examples

4) Find all loans larger than $1200



5) find the loan number for each loan that is larger than $1200



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

$\{ N \mid depositor(N, \_) \vee borrower(N, \_) \}$

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

$\{ N \mid \exists L \; (borrower(N, L) \wedge loan(L, \text{'Perry'}, \_)) \}$

(not interested in loan but needed)

- 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.

$\{ N \mid \exists L \; (borrower(N, L) \wedge loan(L, \text{'Perry'}, \_) \wedge \neg depositor(N, \_)) \}$

- Determine the largest account balance.

$\{ A \mid account(\_, \_, A) \wedge \forall B (account(\_, \_, B) \Rightarrow B \leq A) \}$

- Consider the following DRC expressions. Formulate equivalent relational algebra expressions. Assume column i of relation r has name rci.
  - $\{X, Y \mid p(X) \wedge q(X, Y)\}$

$\Pi_{qc1,qc2}(\sigma_{pc1=qc1}(p \times q))$ (same as) $(\Pi_{pc1,pc2}(\sigma_{pc1=qc1}(p \times q)))$

  - $\{X \mid p(X, 2) \wedge X > 7\}$

$\Pi_{pc1}(\sigma_{pc2=2 \wedge pc1>7}(p))$

  - $\{X \mid p(X) \wedge \neg\exists Y(q(X, Y))\}$

$p - \Pi_{qc1}(q)$

  - $\{X \mid p(X) \wedge \neg\exists Y(p(Y) \wedge Y > X)\}$ (Y is quantified)

$p - \Pi_{pc1}\sigma_{qc1>pc1}(p \times \rho_{q(qc1)}(p))$ (rename)

FOPL and DRC Notation in Text Terminals

- In text terminals math symbols in DRC and FOPL formulas are replaced by text symbols:

Standard DRC Notation	Typewriter Notation
$\neq, \geq, \leq$	`!=, >=, <=`
$\wedge, \vee, \neg, \Rightarrow$	`&&, ||, !, =>`
$\forall, \exists$	`:F, :E`
$\leftarrow$	`<-`

- Example:
  - $\forall X(p(X) \Rightarrow \exists Y(q(X, Y) \wedge Y \leq X))$
  - `:F X (p(X) => :E Y (q(X,Y) && Y <= X))`

SQL

- SQL is based on multisets (or bags) rather than sets. In a multiset an element may occur multiple times.

  We write {...} for a set and {{...}} for a bag.

- SQL does not distinguish between upper and lower case in identifiers and keywords. Small and capital letters are relevant if identifiers are put in quotes

  Comparison of terminology:

SQL	Relational Algebra	Domain Relational Calculus
table	relation	predicate
column	attribute	argument
row	tuple	-
query	RA expression	formula

- In SQL a primary key declaration on a column automatically ensures not null. A primary key is by definition not null and unique.

Domain Types in SQL

- **CHAR**(n) Fixed length character string, with user-defined length *n*.
- **VARCHAR**(n) Variable length character strings, with user-specified maximum length *n*.
- **INTEGER** Integer (a finite subset of the integers that is machine-dependent).
- **SMALLINT** Small integer (a machine-dependent subset of the integer domain type).
- **NUMERIC**(p,d) Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **REAL, DOUBLE PRECISION** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **FLOAT**(n) Floating point number, with user-specified precision of at least *n* digits.

In program code reserved words are usually capitalized (e.g., SELECT).

The end of an SQL statement is often marked with a semicolon.

The drop table command deletes all information about the dropped table from the database.

The alter table command is used to add columns to an existing table
->all tuples in the table are assigned null as the value for the new column

The alter table command can also be used to drop the columns of a table

Examples

- Rewrite
  1. `(X,Y,Z)=(1,2,3)`
  2. `(X,Y,Z)<(1,2,3)`

  to equivalent expressions without row value constructors.

  make decision based on first character if equal compare second and so on.

  1) X = 1 AND Y=2 AND Z=3

  2) (X<1) OR (Y<2 and X=1)OR (X=1 AND Y=2 AND Z<3)

- Identify the problem with the predicate

  ```
  X>0 AND 1/X>0.1
  ```

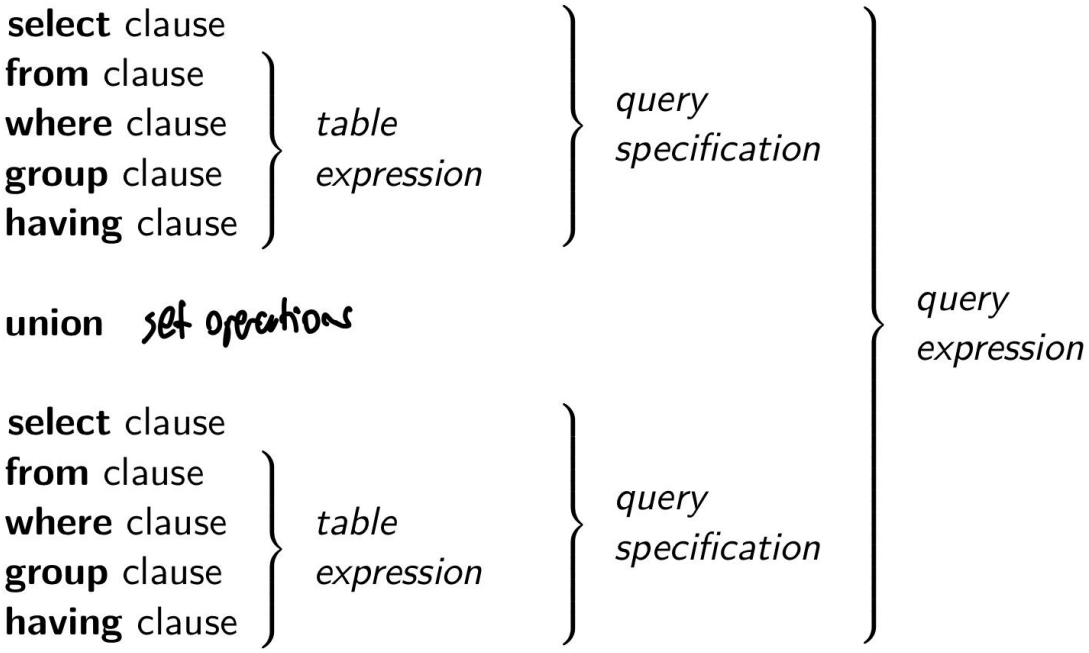
  and propose a solution.

  - AND is not evaluated left to right and stopped as soon as possible
  - division by 0 is possible

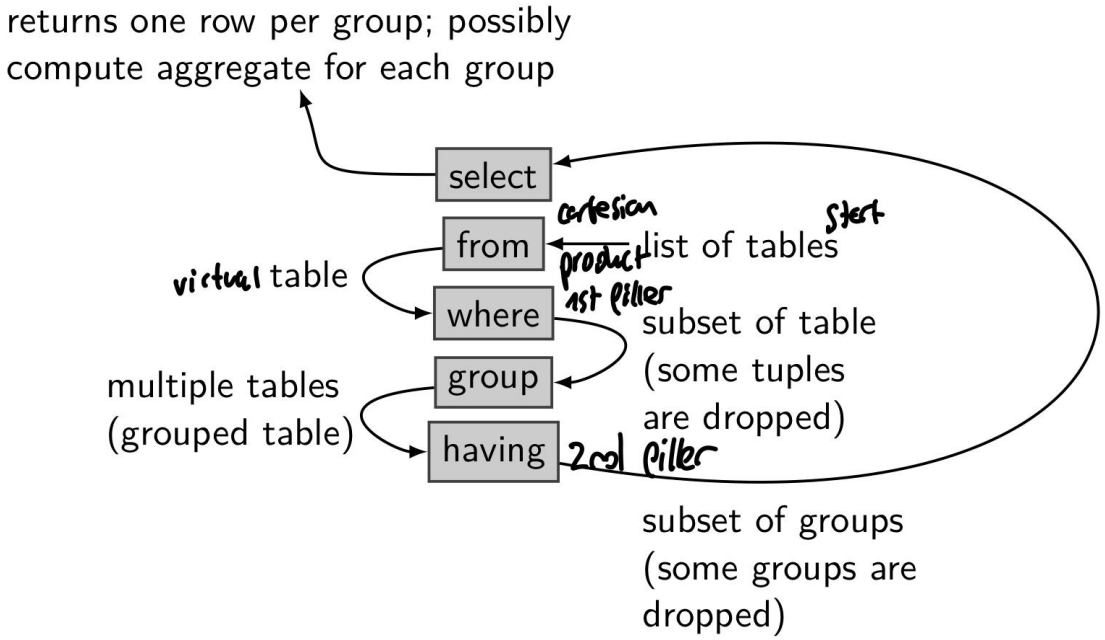
  - case when X >0 then 1/X>0.1 else false end

Structure of SQL Queries

- A typical SQL query has the form:



The result of an SQL query is a (virtual) table.



FROM: from cross product of all tables in from clause. It lists the tables involved in the query -> corresponds to the cartesian product operation in relational algebra.

WHERE: eliminates tuples that do not satisfy the condition in the where clause. It takes the virtual table produced by the from clause and filters out those rows that do not satisfy the condition. It corresponds to the selection predicate in relational algebra.

GROUP BY: groups table according to the columns in the group clause. It takes the table produced by the where clause and returns a grouped table. It groups multiple tuples together. Conceptually, grouping yields multiple tables. For each group one or zero tuples are returned. All queries that are not guaranteed to produce at most one result tuple per group are rejected (compile time error)

HAVING: eliminates groups that do not satisfy the condition of the having clause. It takes a grouped table as input and returns a grouped table. The having condition is applied to each group, and only groups that satisfy the condition are returned. The having clause never returns individual tuples of a group, and may include grouped columns or aggregated columns.

WHERE vs HAVING: both used for filtering of records.

HAVING clause cannot be used without GROUP BY clause. WHERE filters at row level (to individual rows). HAVING filters at group levels (aggregates). WHERE and HAVING can be used together in a query.

WHERE cannot be used with aggregate functions (max, min, count, avg, sum, having, group by), instead use having.

Union vs Union all: union removes duplicates, union all does not remove duplicates. Union operator combines the result set of 2 or more select statements in union compatible relations.

IN vs EXISTS: in is used in multiple or. EXISTS returns either true or false. In is used with big outer query and small inner query. Exists is used with small outer query and big inner query. In compares one value to several values, and exists tells you whether a query returned any results.

JOIN vs UNION: union combines rows, join merges columns. In join it is necessary to have same column name (attribute), for union this is not the case.

SELECT: evaluates the expression in the select clause and produces a result tuple for each group. It corresponds to the projection operation of relational algebra.

- Example: find the names of all branches in the *loan* table:

```
SELECT BranchName
FROM loan
```

- In the relational algebra, the query would be:

$\pi_{BranchName}(loan)$

AS: renames tables and columns (e.g.; old-name as new-name). Keyword as is optional and may be omitted (old-name as new-name = old-name new-name). Renaming can become necessary if the same table is used multiple times in the from clause.

SQL allows qualified column names -> the relation name is put before the column name (e.g., relationName.colName).

In SQL predicates can be combined using the logical connectives and, or, and not. SQL includes a between comparison operator.

The from clause

- SQL supports many different join:
  - `FROM t1 CROSS JOIN t2`
    - Cartesian product
  - `FROM t1 JOIN t2 ON t1.a < t2.b`
    - Theta join
  - `FROM t1 LEFT OUTER JOIN t2 ON t1.a = t2.b`
    - left outer join
  - `FROM t1 NATURAL INNER JOIN t2`
    - Natural join
  - `FROM t1 NATURAL INNER JOIN t2 USING (name)`
    - Limited natural join (not all pair-wise equal columns are used for the natural join; only the ones specified in the using clause)

Examples

Translate the following RA expressions into equivalent SQL fragments.

1. $r \times s$

   FROM r, s

2. $(r \times s) \times t$

   FROM r, s, t

3. $\sigma_{A>5}(r)$

   FROM r WHERE A>5

4. $\sigma_{A>5}(\sigma_{B=4}(r))$

   FROM r WHERE B=4 AND A>5

5. $\sigma_{A=X}(r \times s)$

   FROM r, s WHERE A=X

6. $\sigma_{A>5}(r) \times \sigma_{X=7}(s)$

   From r, s WHERE A>5 AND X=7

Example

- Consider the table expression

  ```
  FROM r1, r2 WHERE x > y AND y > z
  ```

  Which is a join condition and which is a selection condition?

  depends on schema. assume r1(X,Y) and r2(Z,U). then

  - X>Y is selection condition, but Y>Z is a join condition

- Give an example where an expression in the group clause (rather than just column names) makes sense.

  - `GROUP BY (Salary + Bonus) / 1000` (integer division)

Name	Salary	bonUs
pam	4200	0
tom	3200	700
lena	200	3900

SQL allows duplicates in tables as well as in query results. To force the elimination of duplicates, insert the keyword distinct after select.

A * in the select clause denotes "all columns".

The aggregate functions operate on multiple rows. They process all rows of a group and compute an aggregated value for that group.

Columns in select clause outside of aggregate functions must appear in group by list.

Example

- Find the number of non-null balances at the Perryridge branch.

```
SELECT COUNT(Balance)
FROM account
WHERE BranchName = 'Perryridge'
```

Consider duplicates
ignores NULL

- Find the number of tuples in the customer table.

```
SELECT COUNT(*)
FROM customer
```

→ counts all rows
→ consider duplicates
consider NULL

- Find the number of customers per branch. ignore duplicates, ignore NULL

```
SELECT COUNT(DISTINCT CustName), BranchName
FROM account
GROUP BY BranchName
```

SQL allows a query expression to be used in the from clause.

- Why is the following statement strange?

```
SELECT DISTINCT BranchName, SUM(Balance)
FROM account
GROUP BY BranchName
```

→ columns

DISTINCT can be omitted, because group by eliminates duplicates by its own.

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

```
SELECT CustName FROM depositor
UNION
SELECT CustName FROM borrower
```

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

```
SELECT CustName FROM depositor
INTERSECT
SELECT CustName FROM borrower
```

- Find all customers who have an account but no loan:

```
SELECT CustName FROM depositor
EXCEPT
SELECT CustName FROM borrower
```

A subquery is a query expression that is next within another query expression (e.g., SELECT X FROM p WHERE X IN (SELECT y FROM q))

If the inner query uses attributes of the outer query then the queries are correlated.

$(= \textbf{some}) \equiv \textbf{in}$
However, $(\neq \textbf{some}) \not\equiv \textbf{not in}$

Examples

Consider tables

- $p = \{(5), (2)\}, sch(p) = P(X)$
- $q = \{(2), (3)\}, sch(q) = Q(Y)$

Determine the results of the following SQL queries:

1.
```
SELECT *
FROM p
WHERE X IN (
  SELECT Y FROM q )
```
{(2)}

equivalent

2.
```
SELECT *
FROM p
WHERE X = SOME (
  SELECT Y FROM q )
```
{(2)}

3.
```
SELECT *
FROM p
WHERE X NOT IN (
  SELECT Y FROM q )
```
{(5)} ≡ WHERE NOT (X IN (SELECT Y FROM q))

4.
```
SELECT *
FROM p
WHERE NOT ( X = SOME (
  SELECT y FROM q ))
```
{(5)}

5.
```
SELECT *
FROM p
WHERE X <> SOME (
  SELECT Y FROM q)
```
different to

{(2), (5)}

The exists construct returns the value true if the argument subquery is nonempty.

Avoid in, all, any, some and use exists instead.

Example

- Translate the following DRC expressions into SQL. Assume column i of table r has name RCi
  - $\{X \mid p(X,3)\}$

```
SELECT PC1
FROM P
WHERE PC2 = 3
```

  - $\{X, Z \mid \exists Y(p(X,Y) \wedge q(X,Y) \wedge q(Z,3))\}$

```
SELECT pc1, q2.qc1->z
FROM p/q AS q1, q AS q2
WHERE pc1 = q1.qc1
AND pc2 = q1.qc2
AND q2.qc2 = 3
```

- Translate the following DRC expressions into SQL. Assume column i of table r has name RCi
  - $\{X \mid p(X,3) \wedge X > 5\}$

```
SELECT PC1
FROM P
WHERE PC1 > 5 AND PC2 = 3
```

  - $\{X, Y, Z \mid p(X,Y,Z) \wedge \neg q(Y,3)\}$

```
SELECT * -> listing all attributes is perfectly fine as alternative
FROM P
WHERE NOT EXISTS
    SELECT *
    FROM Q
    WHERE QC1 = PC2
    AND QC2 = 3
```

- Translate the following DRC expressions into SQL. Assume column i of table r has name RCi
  - $\{X \mid p(X) \wedge \neg\exists(p(Y) \wedge Y > X)\}$

```
SELECT RC1
FROM P
WHERE NOT EXISTS (SELECT * FROM P as P2 WHERE P2.RC1 > P.RC1)
```

→ computes largest values from P

  - $\{X \mid p(X) \vee (q(X) \wedge \neg r(X))\}$

```
SELECT * FROM P
UNION
(SELECT * FROM Q)
EXCEPT SELECT * FROM R)
```

Duplicates

- Example: Suppose multiset relations $r_1(A, B)$ and $r_2(C)$ are as follows:

because not relations ⇒ same tuple appears twice

$$r_1 = \{\{(1, a)\ \ (2, a)\}\} \qquad r_2 = \{\{(2),\ \ (3),\ \ (3)\}\}$$

- Then $\pi_B(r_1)$ would be $\{\{(a), (a)\}\}$, while $\pi_B(r_1) \times r_2$ 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:

$$\pi^{B}_{A_1, A_2, \ldots, A_n}(\sigma^{B}_{P}(r_1 \times^{B} r_2 \times^{B} \ldots \times^{B} r_m))$$

not removing duplicates

In SQL, duplicates cannot be distinguished, and it is impossible to eliminate duplicates from a table without using distinct, unique, grouping.

Null values

It is possible for tuples to have a null value, denoted by null, for some of their columns. Null signifies an unknown value or that a value does not exist. The predicate `IS NULL` can be used to check for null values.

The result of any arithmetic expression involving null is null (e.g., 5 + null returns null). Arithmetic operations return null if one argument is null.

Any comparison with null returns unknown (e.g., 5 < null, null <> null, null = null)

- Three-valued logic using the truth value *unknown*:
  - **OR**

    $$(\textit{unknown}\ \textbf{or}\ \textit{true}) = \textit{true},$$
    $$(\textit{unknown}\ \textbf{or}\ \textit{false}) = \textit{unknown}$$
    $$(\textit{unknown}\ \textbf{or}\ \textit{unknown}) = \textit{unknown}$$
  - **AND**

    $$(\textit{true}\ \textbf{and}\ \textit{unknown}) = \textit{unknown},$$
    $$(\textit{false}\ \textbf{and}\ \textit{unknown}) = \textit{false}$$
    $$(\textit{unknown}\ \textbf{and}\ \textit{unknown}) = \textit{unknown}$$
  - **NOT**

    $$(\textbf{not}\ \textit{unknown}) = \textit{unknown}$$
  - "P **is unknown**" evaluates to *true* if predicate P evaluates to *unknown*

Null values are not all identical nor all different.

Any row (or group) that does not evaluate to true is eliminated by the where (having) clause.

Two rows are duplicates if corresponding columns are either equal or both are bull (thus, null values are equal for this purpose). Grouping groups null values together.

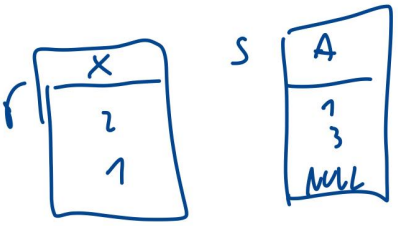
All aggregate operations, except count(*), ignore tuples with null values on the aggregated columns (Thus, sum(x) is different from summing all values in a column).

If all values in a column are null then aggregates over this column return null (except count, which gives 0).

Example

- Explain similarities and differences between the following statements over relations with schemas R(X) and S(A):

1. `SELECT * FROM r WHERE X NOT IN ( SELECT A FROM s )` empty result
2. `SELECT * FROM r WHERE NOT EXISTS ( SELECT * FROM s WHERE X = A)`



These two statements are equivalent if null values are left out

Ordering the Display of Tuples

desc is used for descending ordering and asc for ascending ordering (e.g., ORDER BY CustName DESC).

Insertion: use insert into keyword

Deletion: use delete from

Update: use update, set, and eventually where

DATABASE PROGRAMMING

Views

A view is a table whose rows are not stored in the database. The rows are computed when needed from the view definition. This is used in cases where it is not desirable for all users to see the entire logical method (all the actual tables stored in the database), or the user wants to access computed results (rather than the actual data stored on the disk). A view provides a mechanism to hide data from the view of users, or to give users direct access to the results of (complex) computations.

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

  can be used as table name

  CREATE VIEW v **AS** <query expression>

  where <query expression> is any legal SQL 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.
- When a view is created, the query expression is stored in the database.
- Any table that is not of the conceptual model but is made visible to a user as a "virtual table" is called a **view**.

Tables and views can be used interchangeably in queries. There is no record in a view, it only holds the definition of the table and fetches data from the table and shows it.

With clause

The with clause provides a way of defining temporary views whose definition is available only to the query in which the with clause

occurs. The with clause is useful to structure complex SQL statements and eliminate code repetitions.

Examples

- Find all accounts with the maximum balance

```
WITH
maxBalance(Val) AS (
  SELECT MAX(Balance)
  FROM account
)
SELECT AccNr
FROM account, maxBalance
WHERE account.Balance = maxBalance.Val
```

defines a temporary local table max Balance

main query

- Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.

```
WITH
braTot(BranchName,Val) AS (
  SELECT BranchName, SUM(Balance)
  FROM account
  GROUP BY BranchName
),
braTotAvg(Val) AS (
  SELECT AVG(Val)
  FROM braTot
)
SELECT BranchName
FROM braTot, braTotAvg
WHERE braTot.Val > braTotAvg.Val
```

temp table1

total balance per branch

temp table2

avg total balance

main query

Recursion in SQL

Recursive views are required to be monotonic. That is, if we add tuples to manage the manager the view contains all of the tuples it contained

before, plus possibly more. Negation (not exists, not in, except) in recursive views is not allowed. EXCEPT corresponds to the set difference operator in relational algebra.

Integrity Constraints

They guard against damage to the database, by ensuring that changes to the database do not result in a loss of data consistency.

Integrity constraints on single relations:

- domain constraints
- **not null**
- **primary key**
- **unique**
- **check**(P), where P is a predicate over a single relation

supported by DBMSs (they make sure to check efficiently)

Integrity constraints on multiple relations:

- **foreign key**
- **check**(P), where P is a predicate over multiple relations
- **assertion**

not supported (too costly)

Domain constraints check values inserted in the database, and they check queries to ensure that the comparisons make sense.

Types of domain constraints in SQL:

NOT NULL: This constraint ensures that a column cannot contain NULL values. Candidate keys are permitted to be null (in contrast to primary keys).

UNIQUE: This constraint ensures that each value in a column is unique and cannot be repeated.

PRIMARY KEY: This constraint combines the NOT NULL and UNIQUE constraints and ensures that a column contains unique and non-null values that can be used as a unique identifier for a row in a table. The primary key clause lists attributes that form the primary key. Primary key = unique + not null

FOREIGN KEY: This constraint ensures that values in a column must match the values of another column in a different table. This is used to create relationships between tables. By default, a foreign key references the primary key attributes of the referenced table.

CHECK: This constraint ensures that values in a column must satisfy a specific condition or expression.

Referential Integrity

Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation (e.g., if perryridge is a branch name appearing in one of the tuples in the account relation then there exists a tuple in the branch relation for branch Perryridge).

Examples

```
CREATE TABLE customer (
   CustomerName VARCHAR(20)
   CustStreet VARCHAR(30),
   CustCity VARCHAR(30),
   PRIMARY KEY (CustomerName) )
```

```
CREATE TABLE account (
  AccNr VARCHAR(10),
  BranchN VARCHAR(15),
  Balance INTEGER,
  PRIMARY KEY (AccNr),
  FOREIGN KEY (BranchN) REFERENCES branch )
```

Primary key of relation branch

Assertions

- An **assertion** is a predicate expressing a condition that the database must satisfy.
- An assertion in SQL takes the form

  create assertion <assertion-name> **check** <predicate>
- When an assertion is made, the system tests it for validity, and tests it again on every update that might violate the assertion.
  - This testing may introduce a significant amount of overhead; hence assertions should be used with care.
- Asserting

  $\forall X(p(X))$

  is achieved using

  $\neg\exists\neg(p(X))$

Examples

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

```
CREATE ASSERTION sum_constraint CHECK (
  NOT EXISTS (
    SELECT *
    FROM branch b
    WHERE
      ( SELECT SUM(Amount)
        FROM loan l
        WHERE l.BranchName = b.BranchName )
      >=
      ( SELECT SUM(Balance)
        FROM account a
        WHERE l.BranchName = b.BranchName )))
```

Create a table for cities and a table for states. For each state its capital shall be recorded and for each city the state it is located in shall be recorded. Discuss the properties of the following solution:

```
CREATE TABLE cities(CName VARCHAR(9) PRIMARY KEY, State VARCHAR(9));
CREATE TABLE states(SName VARCHAR(9) PRIMARY KEY, Capital VARCHAR(9));
ALTER TABLE cities ADD FOREIGN KEY (State) REFERENCES states(SName);
ALTER TABLE states ADD FOREIGN KEY (Capital) REFERENCES cities(CName);
```

User-Defined Functions (UDF)

They allow the execution of application logic in the process space of the DBMS. This is good for the performance since it reduces the amount of data that is transferred between client and server.

UDF functions can make arithmetic calculations, query tables, manipulate tables, and return single values or tables.

PL/pgSQL Value Functions

A value function returns a value (or tuple).

Example

- Count of the number of accounts owned by a customer:

```
CREATE FUNCTION accountCnt (CName VARCHAR(9))
RETURNS INTEGER AS
$$
DECLARE
  accCnt INTEGER;
BEGIN
  SELECT COUNT(*) INTO accCnt
  FROM depositor
  WHERE depositor.CustName = CName;
  RETURN accCnt;
END;
$$ LANGUAGE PLPGSQL;
```

- Usage: `SELECT accountCnt('Bob');`

PL/pgSQL Table Functions

A table function returns a table. They can be used instead of tables and allow input parameters.

Example

- All accounts with a balance above a threshold:

```
CREATE FUNCTION highAccnts (limitVal INTEGER)
RETURNS TABLE (AccNum INTEGER,
               BrName VARCHAR(15),
               Bal INTEGER) AS
$$
BEGIN
  RETURN QUERY
    SELECT AccNr, BranchName, Balance
    FROM account A
    WHERE A.Balance > highAccnts.limitVal;
END;
$$ LANGUAGE PLPGSQL;
```

- Usage: `SELECT * FROM highAccnts(1000);`

Triggers

It is a procedural piece of code that is automatically executed in response to certain events (insert, delete, update tuples) on a particular table in a database. They are actions that fire automatically if the condition is satisfied.

In postgreSQL, a trigger consists of two parts:

- A trigger function that defines the action to be performed. It must return null or a row with the schema of the table the trigger was fired for
- A trigger that defines when the trigger must be fired. It is associated with a table or view and executes the specified function when certain events occur. It can fire before or after the operation has completed, or instead of the operation. A trigger can fire for each row or for each statement.

RELATIONAL DATABASE DESIGN

The goal of relational database design is to find a good collection of relation schemas. The main problem is to find a good grouping of the attributes into relation schemas.

We have a good collection of relation schemas if we ensure a simple semantics of tuples and attributed, avoid redundant data, avoid update anomalies, avoid null value as much as possible, and ensure that exactly the original data is recorded and (natural) joins do not generate spurious tuples (tuples that do not actually represent a valid relationship between entities in the database).

Example

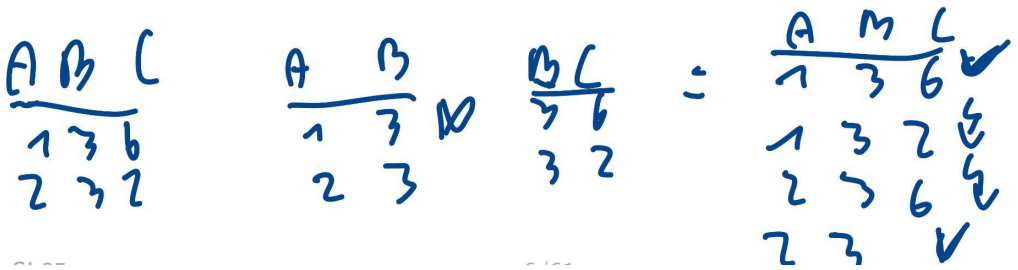
- Consider the relation schema:
  - EmpProj(SSN, PNum, Hours, EName, PName, PLoc)

- Update Anomaly:
  - Changing the name of project location "Houston" to "Dallas" for an employee forces us to make this change for all other employees working on this project.
- Insert Anomaly:
  - Cannot insert a project unless an employee is assigned to it (except by using null values). But nulls are not allowed for primary keys i.e. SSN-Number
- Delete Anomaly:
  - When a project is deleted, it will result in deleting all the employees who work on that project.

Problem of this schema: it mixes information about employees, Proj, and Assignments

Guidelines

1) Each tuple in a relation should only represent one entity or relationship instance
2) Design a schema that does not suffer from insertion, deletion and update anomalies
3) Relations should be designed such that their tuples will have as few null values as possible; attributes that are null shall be placed in separate relations (along with the primary key)
4) Relations should be designed such that no spurious (i.e., wrong) tuples are generated if we fo a natural join of the relations



Functional Dependencies (FDs)

FDs are used to specify formal measures of the goodness of relational design. Functional dependencies and keys are used to define normal forms for relations. Functional dependencies are constraints that are derived from the meaning and interrelationships of the attributes (not values).

A set of attributes X functionally determines a set of attributes Y if the value of X determines a unique value for Y.

Y → X means that X is functionally dependent on Y.

A functional dependency X |-> Y is trivial iff Y $\subseteq$ X. X -> X is trivial.

- $X \rightarrow Y$ denotes a functional dependency.
- $X \rightarrow Y$ means that X functionally determines Y.
- $X \rightarrow Y$ holds if whenever two tuples have the same value for X they have the same value for Y.
  - For any two tuples t_1 and t_2 in any relation instance $r(R)$: If $t_1[X] = t_2[X]$ then $t_1[Y] = t_2[Y]$ ↳ not just one instance
- $X \rightarrow Y$ for R specifies a *constraint* on the schema, i.e., on all possible relation instances r(R). (sets of attributes)
- FDs are derived from the real-world constraints on the attributes.

- Notation: instead of $\{A, B\}$ we write AB (or A, B), e.g., $AB \rightarrow BCD$ (instead of $\{A, B\} \rightarrow \{B, C, D\}$)

Review 5.1

r

A	B	C
1	1	3
2	1	1
3	2	2
4	1	4

Consider the relation instance $r(R)$ and the statements

1. A is a primary key of R
2. $B \to C$ is a functional dependency that holds for R
3. $C \to B$ is a functional dependency that holds for R
4. $BC \to A$ is a functional dependency that relation instance r satisfies

Which of these statements are true?

1) possibly true (depends if this is the only instance it is, in general we don't know)

2) false 1->3 but 1->1

3) possibly true

4) true

Examples of FD constraints:

- Social security number determines employee name
  - $SSN \to EName$ i.e. given SSN, I can find (first and last) employee name
- Project number determines project name and location
  - $PNum \to PName, PLoc$
- Employee ssn and project number determines the hours per week that the employee works on the project
  - $SSN, PNum \to Hours$

A FD constraint must hold on every relation instance r(R). If K is a candidate key of R, then K functionally determines all attributes in R (since there are never two distinct tuples with $t_1[K] = t_2[K]$)

- Given a set of FDs F, we can **infer** additional FDs that hold whenever the FDs in F hold
- Armstrong's inference rules (aka Armstrong's axioms):
  - Reflexivity: $Y \subseteq X \models X \rightarrow Y$
  - Augmentation: $X \rightarrow Y \models XZ \rightarrow YZ$
  - Transitivity: $X \rightarrow Y, Y \rightarrow Z \models X \rightarrow Z$
- Notation:
  - $A \models B$ means that from A we can infer B
  - XZ stands for $X \cup Z$
- Armstrong's inference rules are **sound** and **complete**
  - These rules hold (are correct) and all other rules that hold can be deduced from these

- Additional inference rules that are useful:
  - Decomposition: $X \rightarrow YZ \models X \rightarrow Y, X \rightarrow Z$
  - Union: $X \rightarrow Y, X \rightarrow Z \models X \rightarrow YZ$
  - Pseudotransitivity: $X \rightarrow Y, WY \rightarrow Z \models WX \rightarrow Z$
- The last three inference rules, as well as any other inference rules, can be deduced from Armstrong's inference rules (because of the completeness property).

- The **closure** of a set F of FDs is the set F^+ of all FDs that can be inferred from F.
- The **closure** of a set of attributes X with respect to F is the set X^+ of all attributes that are functionally determined by X.
- F^+ and X^+ can be calculated by repeatedly applying Armstrong's inference rules to F and X, respectively.

- Two sets of FDs F and G are **equivalent** if:
  - Every FD in F can be inferred from G, and
  - Every FD in G can be inferred from F
  - Hence, F and G are equivalent if $F^+ = G^+$ closures are equivalent
- Definition: F **covers** G if every FD in G can be inferred from F (i.e., if $G^+ \subseteq F^+$) not necessarily other way around
- F and G are equivalent if F covers G and G covers F

Example

Consider $F = \{A \to C, AC \to D, E \to AD, E \to H\}$ and $G = \{A \to CD, E \to AH\}$. Are F and G equivalent?

compute attribute closures and compare them

F: A^+ (A determines) $= ACD$

$AC^+ = ACD$

$E^+ = ACDEH$ (from A)

$H^+ = H$

$D^+ = D$

$=$ equivalent $F^+ = G^+$

G: $A^+ = ACD$

$AC^+ = ACD$

$E^+ = ACDEH$

$H^+ = H$

$D^+ = D$

- A set of FDs is **minimal** if it satisfies the following conditions:
  1. No pair of FDs has the same left-hand side.
  2. We cannot remove any dependency from F and have a set of dependencies that is equivalent to F.
  3. We cannot replace any dependency $X \rightarrow A$ in F with a dependency $Y \rightarrow A$, where $Y \subset X$ and still have a set of dependencies that is equivalent to F.

- Every set of FDs has an equivalent minimal set
- There can be several equivalent minimal sets
- There is no simple algorithm for computing a intuitive minimal set of FDs that is equivalent to a set F of FDs
- The first condition can also be changed to "every FD has a single attribute for its right-hand side" (Elmasri and Navathe does this). Note: $X \rightarrow YZ \equiv X \rightarrow Y, X \rightarrow Z$

left side unique — right side is a single attribute

Normal Forms

Normalization: The process of decomposing bad relations by breaking up their attributes into smaller relations that satisfy the normal forms (break up relations into single attributes: all relationships between values get lost, break up into 2 or more attributes: OK, relationships can be preserved). A normalized database consists of a good collection of relation schemas -> no or little redundancy. In practice, normalization is carried out to guarantee that the resulting schemas are of high quality.

properties of single relation schemas

- 1NF
  - attribute values must be atomic → not sets of values
- 2NF, 3NF, BCNF
  - based on candidate keys and FDs of a relation schema
- 4NF
  - based on candidate keys, multi-valued dependencies (MVDs)
- 5NF
  - based on candidate keys, join dependencies (JDs)
- Additional properties may be needed to ensure a good relational design:
  - Losslessness of the corresponding join (very important and cannot be sacrificed)
  - Preservation of the functional dependencies (less stringent and may be sacrificed)

multiple relation schemas

Denormalization: process of storing the join of higher normal form relations as a base relation (which is in a lower normal form since the join destroys the normal form).

First Normal Form (1NF)

Disallows composite attributes, multivalued attributes and nested relations (attributes whose values for an individual tuple are relations)

Example

- The following instance of schema *Department*(*DName*, <u>*DNum*</u>, *DMgrSSN*, *DLoc*) is not in 1NF:

department

composed attributes (not atomic)

DName	DNum	DMgrSSN	DLoc
'Research'	5	334455	{'Bellaire', 'Sugarland', 'Houston' }
'Administration'	4	987654	{ 'Stafford' }
'Headquarters'	1	888666	{ 'Houston' }

Remedy to get 1NF: form new relations for each multivalued attribute or nested relation

Second Normal Form (2NF)

A relation schema R is in second normal form iff each attribute not contained in a candidate key is not partially functional dependent on a candidate key of R. Relation must also be in 1NF (ask and make sure!)

An attribute is partially functional dependent on a candidate key if it is functionally dependent on a proper subset of the candidate key.

Example

The following relation is not in 2NF:

empproj

candidate key

SSN → EName (EName is partially functional dependent on candidate key)

SSN	PNum	Hours	EName	PName	PLoc
1234	1	32.5	'Smith'	'ProductX'	'Bellaire'
1234	2	7.5	'Smith'	'ProductY'	'Sugarland'
6688	3	40.5	'Narayan'	'ProductZ'	'Houston'
4567	1	20.0	'English'	'ProductX'	'Bellaire'
4567	2	20.0	'English'	'ProductY'	'Sugarland'
3334	2	10.0	'Wong'	'ProductY'	'Sugarland'
3334	3	10.0	'Wong'	'ProductZ'	'Houston'
3334	10	10.0	'Wong'	'Computerization'	'Stafford'
3334	20	10.0	'Wong'	'Reorganization'	'Houston'

Consider $R(A, B, C)$ and $F = \{A \to BC, B \to C\}$. Is R in 2NF? Is R a good schema?

1. candidate key: A
   not contained in candidate key: B, C solely dependent on A (not partially func. dependent)

2. R is in 2NF (no non-cand key is part. func. dependent on a cand. key of R.

3.

A	B	C
'a'	'b'	'c'
'b'	'b'	'c'

SSN	PhoneNr	Canton
111	031 1234	'BE'
222	031 1234	'BE'

we store twice that 031 1234 is a phone nr. of canton 'BE', there is redundancy

Remedy to get 2NF: decompose and set up a new relation for each partial key with its dependent attributes. Keep a relation with the original key and any attributes that are functionally dependent on it.

Third Normal Form (3NF)

- A relation schema R is in **third normal form (3NF)** iff for all $X \to A \in F^+$ at least one of the following holds:
  - $X \to A$ is trivial (→ all functional dependencies that exist)
  - X is a superkey for R
  - A is contained in a candidate key of R

- Intuition: "Each non-key attribute must describe the key, the whole key, and nothing but the key." [Bill Kent, CACM 1983]

- A relation that is in 3NF is also in 2NF.

Lecture S.S
Canton describes the phone nr (Phone nr → Canton)
not allowed in 3NF

-> 2NF + No transitive dependencies (when one column depends (is determined by) on a column which is not primary/candidate key)

Boyce-Codd Normal Form (BCNF)

- A relation schema R is in **Boyce-Codd Normal Form (BCNF)** iff for all $X \rightarrow A \in F^+$ at least one of the following holds:
  - $X \rightarrow A$ is trivial
  - X is a superkey for R

- Intuition: "Each attribute must describe the key, the whole key, and nothing but the key." [Chris Date, adaption of Bill Kent for 3NF]

- A relation that is in BCNF is also in 3NF.

- There exist relations that are in 3NF but not in BCNF

With BCNF less redundancy exists but it is no longer possible to check all functional dependencies by looking at one relation only.

Lossless join decomposition: ensures that the decomposition does not introduce wrong tuples when relations are joined together -> must have

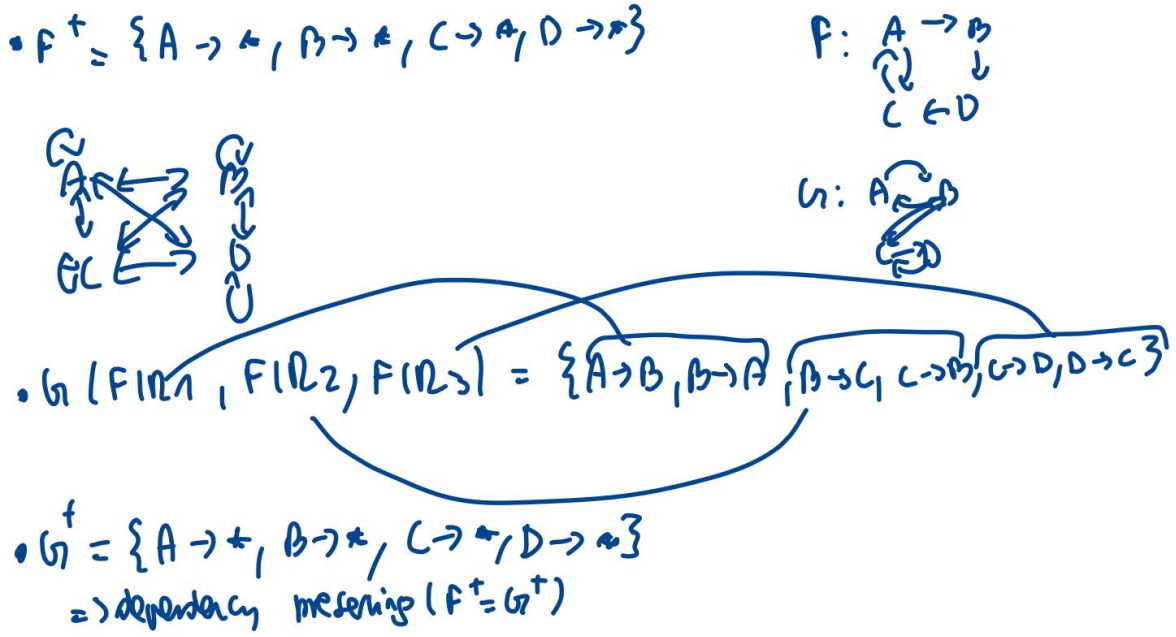
Dependency preservation: ensures that all functional dependency can be checked by considering individual relations Ri only -> nice to have

Dependency Preservation

- Given a set of dependencies F on R, the **projection** of F on Ri, denoted by $F|Ri$ where $attr(Ri)$ is a subset of $attr(R)$, is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \cup Y$ are all contained in $attr(R_i)$.
- Hence, the projection of F on each relation schema Ri in the decomposition D is the set of functional dependencies in F^+, the closure of F, such that all their left- and right-hand side attributes are in $attr(Ri)$.

Example

Consider $R(A, B, C, D)$ and $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow A, A \rightarrow D\}$. Is the decomposition $R1(A, B)$, $R2(B, C)$, and $R3(C, D)$ dependency preserving?



Lossless Join Decomposition

- A decomposition $D = R1, R2, \ldots, Rm$ of R is a **lossless join decomposition** with respect to the set of dependencies F on R if, for every relation instance r of R that satisfies F, the following holds:

left side of a FD goes into both relations

$$\pi_{R1}(r) \bowtie \ldots \bowtie \pi_{attr(Rm)}(r) = r$$

No extra tuples should show up

- Note: The word loss in lossless refers to loss of information, not to loss of tuples. If a join decomposition is not lossless then new spurious tuples are present in the result of the join.
- R1 and R2 form a lossless join decomposition of R with respect to a set of functional dependencies F iff
  - $(R1 \cap R2) \rightarrow (R1 - R2)$ is in F^+ or
  - $(R1 \cap R2) \rightarrow (R2 - R1)$ is in F^+

right side of a FD goes into one relation

Split according to FDs!

Example

Consider $R(A, B, C)$, $F = \{AB \rightarrow C, C \rightarrow B\}$, $R1(A, C)$, $R2(B, C)$.

1. Is $R1, R2$ a lossless decomposition of R?
2. Illustrate your answer for $r = \{(x, 0, a), (z, 1, c), (x, 2, c)\}$.
3. Discuss what happens if we replace tuple $(x, 2, c)$ by $(x, 2, b)$.

2)

A	B	C
x	0	a
z	1	c
x	2	c

A	C
x	a
z	c
x	c

⋈

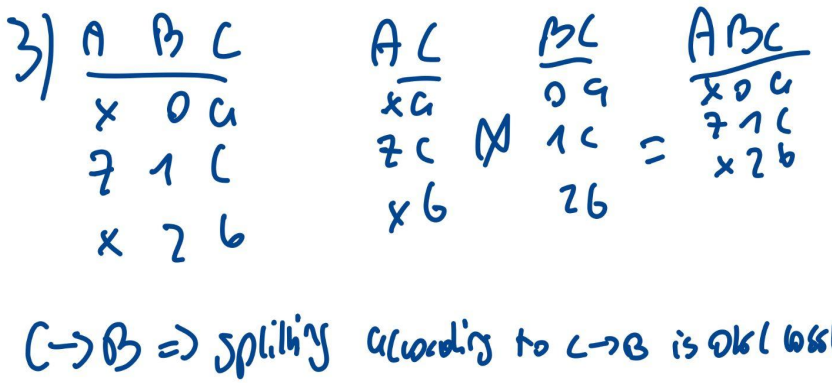
B	C
0	a
1	c
2	c

=

A	B	C
x	0	a
z	1	c
x	1	c
x	2	c

≠

C ↛ B ⇒ we don't split according to a FD is not ok (lossy decomposition, we get extra tuples)



Algorithm for BCNF Normalization

```
Set D := { R };
while a relation schema Q in D is not in BCNF do
    find a functional dependency X → Y in Q that violates BCNF;
    replace Q in D by two relation schemas (Q − Y) and (X ∪ Y);
```

Assumption: No null values are allowed for the join attributes.

The result is a lossless join decomposition of R. The resulting schemas do not necessarily preserve all dependencies. The result is not unique, but it depends on the sequence in which FDs are chosen (problematic, to get a good intuitive schema the same intuition is needed)

The normalization algorithms are not deterministic in general (e.g., not a unique minimal cover). It is not always possible to find a decomposition into relation schemas that preserves dependencies and allows each relation schema in the decomposition to be in BCNF (some FDs are lost because attributes are in different schemas)

3NF vs BCNF

It is possible to construct a decomposition that is in BCNF and is lossless. It is possible to construct a decomposition that is in 3NF, is lossless, and preserves dependency. It is not always possible to

construct a decomposition that is in BCF, is lossless, and is dependency preserving. 3NF allows redundancies that BCNF does not allow.

Multivalued Dependencies (MVDs)

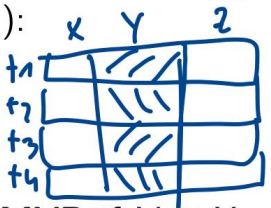
Definition:

multidetermines

- A **multivalued dependency (MVD)** $X \twoheadrightarrow Y$ on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation instance r of R:

  If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in r with the following properties, where we use Z to denote $(R - (X \cup Y))$:

  - $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
  - $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.
  - $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.



- A MVD $X \twoheadrightarrow Y$ for R is called a **trivial MVD** if $Y \subseteq X$ or $X \cup Y = attr(R)$.

t_1, t_2, t_3, t_4 don't have to be different tuples

Inference Rules for Functional and Multivalued Dependencies

- reflexivity FDs: $X \supseteq Y \models X \rightarrow Y$.
- augmentation FDs: $X \rightarrow Y \models XZ \rightarrow YZ$.
- transitivity FDs: $X \rightarrow Y, Y \rightarrow Z \models X \rightarrow Z$.

} FDs

- complementation: $X \twoheadrightarrow Y \models X \twoheadrightarrow (R - (X \cup Y))$.
- augmentation MVDs: $X \twoheadrightarrow Y, W \supseteq Z \models WX \twoheadrightarrow YZ$.
- transitivity MVDs: $X \twoheadrightarrow Y, Y \twoheadrightarrow Z \models X \twoheadrightarrow (Z - Y)$.

} MVDs

- replication: $X \rightarrow Y \models X \twoheadrightarrow Y$. each FD is a MVD
- coalescing: $X \twoheadrightarrow Y, \exists W(W \cap Y = \emptyset, W \rightarrow Z, Y \supseteq Z) \models X \rightarrow Z$.

A FD is a case of a MVD

Fourth Normal Form (4NF)

Definition:

- A relation schema R with a set of functional and multivalued dependencies F is in **4NF** iff, for every multivalued dependency $X \twoheadrightarrow Y$ in F^+ at least one of the following holds:
  - $X \twoheadrightarrow Y$ is trivial
  - X is a superkey for R
- F^+ is called the **closure** of F and is the complete set of all dependencies (functional or multivalued) that will hold in every relation state r of R that satisfies F.

- If a relation is not in 4NF because of the MVD $X \twoheadrightarrow Y$ we decompose R into $R1(X \cup Y)$ and $R2(R - Y)$.
- Such a decomposition is lossless.
- R1 and R2 form a lossless join decomposition of R with respect to a set of functional and multivalued dependencies iff
  - $(R1 \cap R2) \twoheadrightarrow (R1 - R2)$ or
  - $(R1 \cap R2) \twoheadrightarrow (R2 - R1)$