

L3 Info.

Bases de Données, Bases de Connaissances

Marie-Christine Fauvet

Joseph Fourier University of Grenoble Alpes – UFR IM²AG

2015/2016

Chapter 1 – Introduction du cours

1 Organisation

2 Contenu de l'enseignement

3 Overview of DBMSs

Equipe pédagogique

Marie-Christine Fauvet Nawal Ould-Amer	Professeur UJF, LIG Doctorante UGA	Marie-Christine.Fauvet@imag.fr Nawal.Ould-Amer@imag.fr
---	---------------------------------------	---

Chapter 1 – Introduction du cours

- 1 Organisation
- 2 Contenu de l'enseignement
- 3 Overview of DBMSs

Pré requis pour cette UE

- Algorithmique et de programmation.
- Logique : principes de base.
- Théorie des ensembles : ensembles, opérateurs sur les ensembles (\in , \notin , \subset , $\not\subset$, \cap , \cup , $-$, X , etc), relation.
- Modèle relationnel de données : relation, attribut, domaine
- SQL (interrogation) : sélection, projection, produit, agrégation, partition

Plan

- Partie I: Bases de données relationnelles
 - ① Rappels sur le modèle relationnel de données
 - ② Rappels sur SQL
 - ③ Conception de schéma relationnel
- Partie II: Logique et bases de données
 - ① Calculs relationnels (à variable n-uplet/à variable domaine)
 - ② Bases de données et logique
 - ③ Bases de données déductives

Compétences visées

L'objectif est d'acquérir les connaissances minimales sur les bases de données et les bases de connaissances nécessaires à tout informaticien :

- **Maîtriser**

- la compréhension d'un schéma relationnel
- l'expression de requêtes d'interrogation en algèbre relationnelle et en SQL
- la traduction en SQL une requête exprimée dans l'algèbre relationnelle

- **Savoir**

- évaluer et améliorer la qualité d'un schéma de base de données
- exprimer des requêtes dans divers formalismes logiques
- modéliser et raisonner sur des connaissances à l'aide de règles logiques

Volumes horaires

- Cours Magistral (CM) : 16.5
- Travaux Dirigés (TD) : 33
- Heures encadrées : 49.5
- Travail personnel estimé : 33

Evaluation

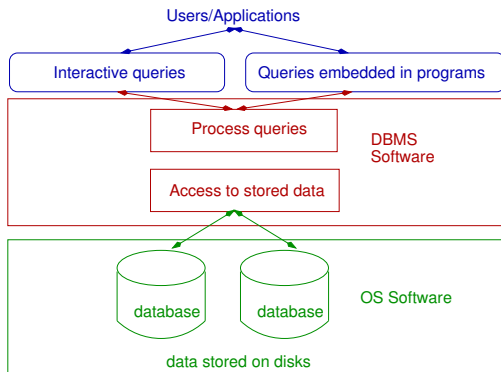
Moy des interrogations	cc1
Apnée 1	ap1
Apnée 2	ap2
Quick	q
	$cc2 = 0.25 * ap1 + 0.25 * ap2 + 0.5 * q$
Contrôle continu	$cc = 0.5 * cc1 + 1.5 * cc2$
Examen (3 heures)	ex
note INF353	$0.67 * ex + 0.33 * cc$

Absence (justifiée ou pas) à une des épreuves du contrôle continu ==>
0 à cette épreuve.

Chapter 1 – Introduction du cours

- 1 Organisation
- 2 Contenu de l'enseignement
- 3 Overview of DBMSs

A DataBase Management System (DBMS) is a set of software that provides means for storing and accessing data.



Databases basics

A database consists of real world information, organized so that it can be queried and modified (possibly deleted) later

A consistent collection of structured data

- which model (a part of) the real world
- designed to fulfill specific needs

Examples:

- My address book: details of my friends and relatives (first-names, last-names, addresses and phone numbers). Size: a few kilo bytes
- World Data Centre for Climate: 220 terabytes¹ of web data, 6 petabytes² of additional data

¹1 tera= 10^{12}

²1 peta= 10^{15}

A Relational Database Management System (RDBMS)

A software system designed to support data-intensive applications

- with high-level access to the data (relational model, SQL)
- providing efficient storage and retrieval (disk/memory management)
- supporting multiple simultaneous users (privilege, protection)
- carrying out large numbers of operations (transaction management)
- maintaining reliable access to the stored data (backup, recovery)

DBMSs deal with many similar issues to operating systems.

As a Summary

DBMSs provide access to valuable information resources in an environment that is:

- *Shared* - concurrent access by multiple users
- *Unstable* - potential for hardware/software failure

It involves techniques to ensure that each user sees the system as:

- *Unshared* - their work is not inadvertently affected by other users
- *Stable* - the data survives in the face of system failures

Reading Material

- J.-D. Ullman, *Principles of Database and Knowledge-Base Systems (Vol. 1)*, Computer Science Press, 1990.
- R. Elmasri, S.B. Navathe *Conception et architecture des bases de données relationnelles*, 4e édition - Pearson Education France, 2004.
Traduction française de *Fundamentals of Database Systems*, 4th Edition, Addison-Wesley Professional

Chapter 2 – Relational Data Model

4 Preliminaries

5 Definitions and Notations

6 Specification of Relations: Summary

A Bit of Culture

The Relational Data Model...

- was first introduced by Ted Codd in 1970,
- is attractive because of its simplicity and its mathematical foundations,
- uses the concept of mathematical relations as a means of modeling information,
- cannot be ignored because of its popularity.

How to Represent Information?

Facts about a store:

- John, Mary, Tom and Peter are employed by the store,

John			
Mary			
Peter			
Tom			

How to Represent Information?

Facts about a store:

- John, Mary, Tom and Peter are employed by the store,
- John's salary is 120, Mary's one is 130, etc.

John	120		
Mary	130		
Peter	110		
Tom	120		

How to Represent Information?

Facts about a store:

- John, Mary, Tom and Peter are employed by the store,
- John's salary is 120, Mary's one is 130, etc.
- Mary lives in Wollongong, John and Peter in Randwick, etc.

John	120	Randwick	
Mary	130	Wollongong	
Peter	110	Randwick	
Tom	120	Botany Bay	

How to Represent Information?

Facts about a store:

- John, Mary, Tom and Peter are employed by the store,
- John's salary is 120, Mary's one is 130, etc.
- Mary lives in Wollongong, John and Peter in Randwick, etc.
- John and Tom are in the Toys department, etc.

John	120	Randwick	Toys
Mary	130	Wollongong	Furniture
Peter	110	Randwick	Garden
Tom	120	Botany Bay	Toys

Operations on Relations

Querying relations

- Projection : Give department names
- Selection : Give employees who earn more than 120
- Selection + projection: what is Tom's address?
- Aggregation: how many employees are in the Toys department?

Operations on Relations

Querying relations

- Projection : Give department names
→ {<Toys>, <Furniture>, <Garden>}
- Selection : Give employees who earn more than 120
- Selection + projection: what is Tom's address?
- Aggregation: how many employees are in the Toys department?

Operations on Relations

Querying relations

- Projection : Give department names
→ {<Toys>, <Furniture>, <Garden>}
- Selection : Give employees who earn more than 120
→ {<John, 120, Randwick, Toys>, <Mary, 130, Wollongong, Furniture>, <Tom, 120, Botany Bay, Toys>}
- Selection + projection: what is Tom's address?
- Aggregation: how many employees are in the Toys department?

Operations on Relations

Querying relations

- Projection : Give department names
→ {<Toys>, <Furniture>, <Garden>}
- Selection : Give employees who earn more than 120
→ {<John, 120, Randwick, Toys>, <Mary, 130, Wollongong, Furniture>, <Tom, 120, Botany Bay, Toys>}
- Selection + projection: what is Tom's address?
→ {<Botany Bay>}
- Aggregation: how many employees are in the Toys department?

Operations on Relations

Querying relations

- Projection : Give department names
→ {<Toys>, <Furniture>, <Garden>}
- Selection : Give employees who earn more than 120
→ {<John, 120, Randwick, Toys>, <Mary, 130, Wollongong, Furniture>, <Tom, 120, Botany Bay, Toys>}
- Selection + projection: what is Tom's address?
→ {<Botany Bay>}
- Aggregation: how many employees are in the Toys department?
→ {<2>}

Updating relations

- Mary's salary has been increased by 10%

John	120	Randwick	Toys
Mary	143	Wollongong	Furniture
Peter	110	Randwick	Garden
Tom	120	Botany Bay	Toys

Updating relations

- Mary's salary has been increased by 10%
- Phil is now employed by the store, his salary is 140, he is assigned to the Furniture department, his address is Newtown.

John	120	Randwick	Toys
Mary	143	Wollongong	Furniture
Peter	110	Randwick	Garden
Tom	120	Botany Bay	Toys
Phil	140	Newtown	Furniture

Adding new kind of information

Each department is led by a particular employee

Two solutions..... add column(s)
add relation(s)

Add a column (# 1):

- Add a flag for each employee

John	yes	120	Randwick	Toys
Mary	yes	143	Wollongong	Furniture
Peter	yes	110	Randwick	Garden
Tom	no	120	Botany Bay	Toys
Phil	no	140	Newtown	Furniture

Hard to read and decode...

Add a column (§ 2):

- Add the leader name for each department

John	120	Randwick	Toys	John
Mary	143	Wollongong	Furniture	Mary
Peter	110	Randwick	Garden	Peter
Tom	120	Botany Bay	Toys	John
Phil	140	Newtown	Furniture	Mary

Data redundancy...

"John leads the department Toys" is said as many times as there are employees in the department.

Add a relation:

John	Toys
Mary	Furniture
Peter	Garden

Queries are a bit more complex...
"find the salary of the leader for the department Toys"

Chapter 2 – Relational Data Model

4 Preliminaries

5 Definitions and Notations

6 Specification of Relations: Summary

Set: Definition and Some Operations

- A set is a collection of elements (enclosed by braces) pairwise different and usually related to a particular domain.

$$A = \{ 'Furniture', 'Toys', 'Garden' \}$$
$$F = \{ p \in Persons: Gender(p) = 'female' \}$$
$$G = \{ 1, 5, 6, 3 \}$$

A and G are given by extension, while F is so by intension.

Set: Definition and Some Operations

- A set is a collection of elements (enclosed by braces) pairwise different and usually related to a particular domain.

$$A = \{ 'Furniture', 'Toys', 'Garden' \}$$

$$F = \{ p \in Persons: Gender(p) = 'female' \}$$

$$G = \{ 1, 5, 6, 3 \}$$

- Membership:

$$3 \in \{ 1, 3, 5, 6 \}$$

$$8 \notin \{ 1, 3, 5, 6 \}$$

A and G are given by extension, while F is so by intension.

- Cartesian product of sets (denoted \times):

$G \times A =$

$\{ \langle 1, 'Furniture' \rangle, \langle 1, 'Toys' \rangle, \langle 1, 'Garden' \rangle, \\ \langle 3, 'Furniture' \rangle, \langle 3, 'Toys' \rangle, \langle 3, 'Garden' \rangle, \\ \langle 5, 'Furniture' \rangle, \langle 5, 'Toys' \rangle, \langle 5, 'Garden' \rangle, \\ \langle 6, 'Furniture' \rangle, \langle 6, 'Toys' \rangle, \langle 6, 'Garden' \rangle \}$

- Cartesian product of sets (denoted \times):

$G \times A =$

$\{ \langle 1, 'Furniture' \rangle, \langle 1, 'Toys' \rangle, \langle 1, 'Garden' \rangle, \\ \langle 3, 'Furniture' \rangle, \langle 3, 'Toys' \rangle, \langle 3, 'Garden' \rangle, \\ \langle 5, 'Furniture' \rangle, \langle 5, 'Toys' \rangle, \langle 5, 'Garden' \rangle, \\ \langle 6, 'Furniture' \rangle, \langle 6, 'Toys' \rangle, \langle 6, 'Garden' \rangle \}$

- Intersection: $\{1, 3, 5, 6\} \cap \{10, 5, 3, 9\} = \{5, 3\}$

- Cartesian product of sets (denoted \times):

$G \times A =$

$\{ \langle 1, 'Furniture' \rangle, \langle 1, 'Toys' \rangle, \langle 1, 'Garden' \rangle, \\ \langle 3, 'Furniture' \rangle, \langle 3, 'Toys' \rangle, \langle 3, 'Garden' \rangle, \\ \langle 5, 'Furniture' \rangle, \langle 5, 'Toys' \rangle, \langle 5, 'Garden' \rangle, \\ \langle 6, 'Furniture' \rangle, \langle 6, 'Toys' \rangle, \langle 6, 'Garden' \rangle \}$

- Intersection: $\{1, 3, 5, 6\} \cap \{10, 5, 3, 9\} = \{5, 3\}$

- Union:

$\{1, 3, 5, 6\} \cup \{10, 5, 3, 9\} = \{1, 5, 3, 6, 10, 9\}$

- Cartesian product of sets (denoted \times):

$G \times A =$

$\{ \langle 1, 'Furniture' \rangle, \langle 1, 'Toys' \rangle, \langle 1, 'Garden' \rangle, \\ \langle 3, 'Furniture' \rangle, \langle 3, 'Toys' \rangle, \langle 3, 'Garden' \rangle, \\ \langle 5, 'Furniture' \rangle, \langle 5, 'Toys' \rangle, \langle 5, 'Garden' \rangle, \\ \langle 6, 'Furniture' \rangle, \langle 6, 'Toys' \rangle, \langle 6, 'Garden' \rangle \}$

- Intersection: $\{1, 3, 5, 6\} \cap \{10, 5, 3, 9\} = \{5, 3\}$

- Union:

$\{1, 3, 5, 6\} \cup \{10, 5, 3, 9\} = \{1, 5, 3, 6, 10, 9\}$

- Minus (asymmetrical):

$\{1, 3, 5, 6\} - \{10, 5, 3, 9\} = \{1, 6\}$

$\{10, 5, 3, 9\} - \{1, 3, 5, 6\} = \{10, 9\}$

- Cartesian product of sets (denoted \times):

$G \times A =$

$\{ \langle 1, 'Furniture' \rangle, \langle 1, 'Toys' \rangle, \langle 1, 'Garden' \rangle, \\ \langle 3, 'Furniture' \rangle, \langle 3, 'Toys' \rangle, \langle 3, 'Garden' \rangle, \\ \langle 5, 'Furniture' \rangle, \langle 5, 'Toys' \rangle, \langle 5, 'Garden' \rangle, \\ \langle 6, 'Furniture' \rangle, \langle 6, 'Toys' \rangle, \langle 6, 'Garden' \rangle \}$

- Intersection: $\{1, 3, 5, 6\} \cap \{10, 5, 3, 9\} = \{5, 3\}$

- Union:

$\{1, 3, 5, 6\} \cup \{10, 5, 3, 9\} = \{1, 5, 3, 6, 10, 9\}$

- Minus (asymmetrical):

$\{1, 3, 5, 6\} - \{10, 5, 3, 9\} = \{1, 6\}$

$\{10, 5, 3, 9\} - \{1, 3, 5, 6\} = \{10, 9\}$

- Inclusion:

$\{\} \subseteq \{10, 5, 3, 9\}$ ($\{\}$ is also denoted \emptyset)

$\{9, 10\} \subseteq \{10, 5, 3, 9\}$

$\{9, 10\} \subset \{10, 5, 3, 9\}$

$\{9, 10, 3, 5\} \not\subseteq \{10, 5, 3, 9\}$

$\{1, 9, 10, 3, 5, 7\} \not\subseteq \{10, 5, 3, 9\}$

Domain, Relation, Attribute, Schema

- A domain is a set of atomic values (strings, numbers,...).
 $\{ 'Furniture', 'Toys', 'Garden' \}, \{ integers > 100 \}$

Domain, Relation, Attribute, Schema

- A domain is a set of atomic values (strings, numbers,...).
 $\{ 'Furniture', 'Toys', 'Garden' \}, \{ integers > 100 \}$
- A relation is a subset of the cartesian product of a set of domains.
 $\{ < 'John', 120 >, < 'Mary', 130 >, < 'Peter', 110 > \}$
 \subseteq
 $\{ 'John', 'Mary', 'Peter', 'Tom' \} \times \{ integers > 100 \}$

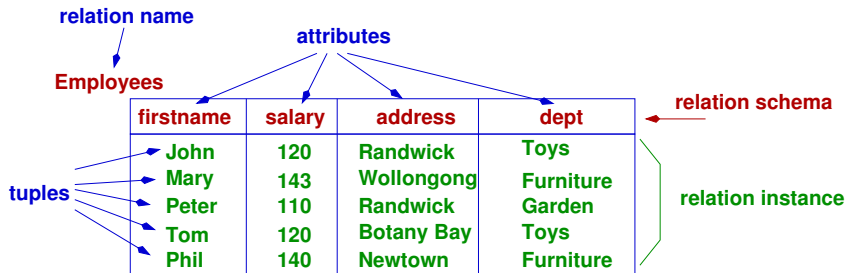
Domain, Relation, Attribute, Schema

- A domain is a set of atomic values (strings, numbers,...).
 $\{ 'Furniture', 'Toys', 'Garden' \}, \{ integers > 100 \}$
- A relation is a subset of the cartesian product of a set of domains.
 $\{ < 'John', 120 >, < 'Mary', 130 >, < 'Peter', 110 > \}$
 \subseteq
 $\{ 'John', 'Mary', 'Peter', 'Tom' \} \times \{ integers > 100 \}$
- An attribute specifies a name for a domain in a relation.
 $dom(Salary) = \{ integer > 100 \}$

Domain, Relation, Attribute, Schema

- A domain is a set of atomic values (strings, numbers,...).
 $\{ 'Furniture', 'Toys', 'Garden' \}, \{ integers > 100 \}$
- A relation is a subset of the cartesian product of a set of domains.
 $\{ < 'John', 120 >, < 'Mary', 130 >, < 'Peter', 110 > \}$
 \subseteq
 $\{ 'John', 'Mary', 'Peter', 'Tom' \} \times \{ integers > 100 \}$
- An attribute specifies a name for a domain in a relation.
 $dom(Salary) = \{ integer > 100 \}$
- A relation schema is specified by its name and a set of attributes.
 $Employees (firstname, salary, address, dept)$

Example:



Attributes and tuples have no order

Interpretation

An interpretation of a relation schema as a predicate:

Employees (firstname, salary, address, dept)

/ $\langle n, s, a, d \rangle \in \text{Employees} \iff$ the employee identified by her(his) name n earns a salary s . She(he) lives at the address a and works in the department d .
/

This is useful to understand a relation schema.

Relational Constraints

- *Domain constraint:*
 $domain(A)=T$ specifies that the values of A must be from the type T.

Relational Constraints

- *Domain constraint:*
 $domain(A)=T$ specifies that the values of A must be from the type T.
- *Key constraint:*
X specifies an uniqueness constraint so that tuples of the relation are pairwise different for X (X set of attributes).

Relational Constraints

- *Domain constraint:*
 $\text{domain}(A)=T$ specifies that the values of A must be from the type T.
- *Key constraint:*
 \underline{X} specifies an uniqueness constraint so that tuples of the relation are pairwise different for X (X set of attributes).
- *Referential integrity constraint:*
R on attributes X refers to S on attributes Y
 all tuples in R, restricted to X must have a corresponding tuple in S that matches on Y. Notation: $R[X] \subseteq S[Y]$.

Chapter 2 – Relational Data Model

4 Preliminaries

5 Definitions and Notations

6 Specification of Relations: Summary

Employees (firstname, salary, address, dept)

/ $\langle n, s, a, d \rangle \in \text{Employees} \iff$ the employee identified by her(his) name n earns a salary s . She(he) lives at the address a and is affected to the department d . */*

Leaders (boss, dept)

/ $\langle b, d \rangle \in \text{Leaders} \iff$ the employee b leads the department d . */*

Domains:

domain (firstname) = domain (boss) =

domain (address) = domain (dept) = strings

domain (salary) = numbers $\neq 0$

Referential integrity constraint:

Leaders[boss, dept] \subseteq Employees[firstname, dept]

Leaders[dept] = Employees[dept]

Chapter 3 – Relational Algebra

- 7 Introduction
- 8 Projection operator
- 9 Selection operator
- 10 Misc
- 11 Joining Relations
- 12 Set Operators
- 13 Division
- 14 Conclusion

Relational algebra consists of

- A set of operators that map relations to relations.
- + Rules for combining those operations into expressions.
- + Rules for evaluating such expressions.

An algebraic expression captures a query

- which is a relation defined by intension
- whose evaluation returns the extension of the relation

Notations:

- $R(X)$ and $S(Y)$ are two relations where
- X and Y are two sets of attributes:
 $X = X_1, \dots, X_n$ ($n \geq 1$) and $Y = Y_1, \dots, Y_p$ ($p \geq 1$)

Chapter 3 – Relational Algebra

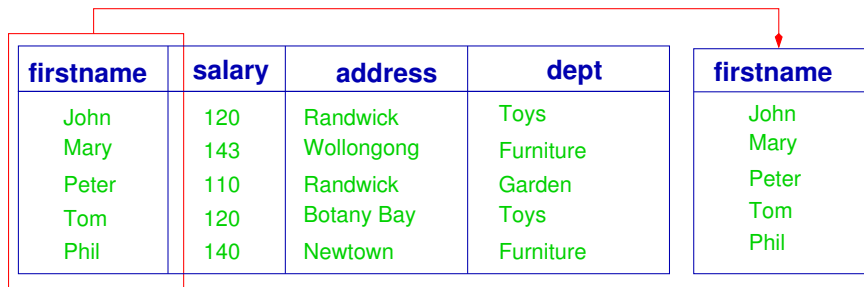
- 7 Introduction
- 8 Projection operator**
- 9 Selection operator
- 10 Misc
- 11 Joining Relations
- 12 Set Operators
- 13 Division
- 14 Conclusion

Definition

- *Notation:* $R[A]$, $A \subseteq X$
- *Relation schema:* A
- *Interpretation:* $R[A] = \{ n[A] \mid n \in R \}$, where $n[A]$ is the projection of the tuple n on attributes A . Because $R[A]$ is a set, all of the tuples are pairwise different.
- Also noted: $\pi_A(R)$

Examples

Give the name of all employees



Examples

Give the name of all employees

Employees[firstname]

firstname	salary	address	dept
John	120	Randwick	Toys
Mary	143	Wollongong	Furniture
Peter	110	Randwick	Garden
Tom	120	Botany Bay	Toys
Phil	140	Newtown	Furniture

firstname
John
Mary
Peter
Tom
Phil

Employees[firstname] is a new relation defined on attribute firstname

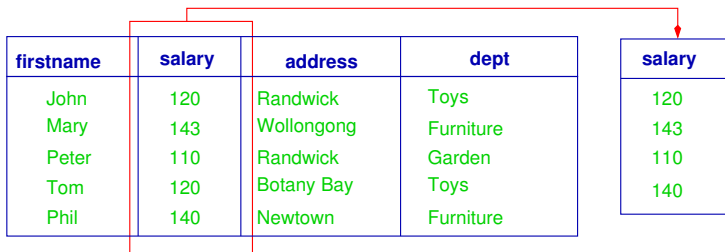
For each employee, find her(his) name and address

Employees[firstname,address]

/ defines a new relation whose attributes are firstname and address */*

Retrieve the salaries

Employees[salary]



firstname	salary	address	dept
John	120	Randwick	Toys
Mary	143	Wollongong	Furniture
Peter	110	Randwick	Garden
Tom	120	Botany Bay	Toys
Phil	140	Newtown	Furniture

salary
120
143
110
140

all values are pairwise different.

Chapter 3 – Relational Algebra

- 7 Introduction
- 8 Projection operator
- 9 Selection operator**
- 10 Misc
- 11 Joining Relations
- 12 Set Operators
- 13 Division
- 14 Conclusion

Definition

- *Notation* : $R:P$ where P is a predicate built as either:
 - A simple condition of the form:
 $\langle \text{att. name} \rangle \langle \text{comp. op.} \rangle \langle \text{att. name} \rangle$ or
 $\langle \text{att. name} \rangle \langle \text{comp. op.} \rangle \langle \text{cst. val.} \rangle$
 - Or a complex boolean expression of the form:
 $\langle \text{cond.} \rangle \langle \text{bool. op.} \rangle \langle \text{cond.} \rangle$
 where $\langle \text{cond.} \rangle$ is either a basic condition or a complex boolean expression.
 - Comparison operators: $=, \neq, <, >, ..$
 - Boolean operators (listed in decreasing precedence): not (\neg), and (\wedge), or (\vee)
 - P may contain brackets
- *Relation Schema*: X (same schema as R)
- *Interpretation*: $R:P = \{n \mid n \in R \wedge P(n)\}$
- Also noted: $\sigma_P(R)$

Boolean Expressions (some examples)

- $A = B$ and $B > 100$
- $A = \text{'Sydney'}$ or $A = \text{'Melbourne'}$ and $B > 100$
- $(A = \text{'Sydney'}$ or $A = \text{'Melbourne'}$) and $B > 100$
- $A = \text{'Sydney'}$ and $B > 100$ or $A = \text{'Melbourne'}$ and $B > 100$
- not $(A = \text{'Sydney'}$ or $A = \text{'Melbourne'}$)
- $A \neq \text{'Sydney'}$ and $A \neq \text{'Melbourne'}$

If you are not familiar with boolean expressions, get a book on Boole's logic and catch up!

Examples

Find the employees whose address is Randwick.

firstname	salary	address	dept
John	120	Randwick	Toys
Mary	143	Wollongong	
Peter	110	Randwick	Garden
Tom	120	Botany Bay	Toys
Phil	140	Newtown	Furniture

Examples

Find the employees whose address is Randwick.

firstname	salary	address	dept
John	120	Randwick	Toys
Mary	143	Wollongong	
Peter	110	Randwick	Garden
Tom	120	Botany Bay	Toys
Phil	140	Newtown	Furniture

Employees:address = 'Randwick'

Find the employees whose address is Randwick. Find also those whose salary is greater than 140 and whose affectation is the Furniture department

firstname	salary	address	dept
John	120	Randwick	Toys
Mary	143	Wollongong	Furniture
Peter	110	Randwick	Garden
Tom	120	Botany Bay	Toys
Phil	140	Newtown	Furniture

Find the employees whose address is Randwick. Find also those whose salary is greater than 140 and whose affectation is the Furniture department

firstname	salary	address	dept
John	120	Randwick	Toys
Mary	143	Wollongong	Furniture
Peter	110	Randwick	Garden
Tom	120	Botany Bay	Toys
Phil	140	Newtown	Furniture

Employees:(address = 'Randwick' or (salary > 140 and dept = 'Furniture'))

Chapter 3 – Relational Algebra

- 7 Introduction
- 8 Projection operator
- 9 Selection operator
- 10 Misc**
- 11 Joining Relations
- 12 Set Operators
- 13 Division
- 14 Conclusion

Introducing New Names

The rename operation \longleftarrow is the means to associate names to the result of a relational algebra expression.

```
A  $\longleftarrow$  Employees:salary  $\geq$  120  
  /* A is a relation with the same schema as Employees */  
Res  $\longleftarrow$  A[firstname]  
  /* Res is a relation whose schema is firstname */  
R (X,Y)  $\longleftarrow$  A[firstname,salary]  
  /* R is a relation whose schema is X,Y */
```

Particularly useful for structuring a solution to a complex problem.

Composition of Operations

Retrieve the name of the employees whose salary is equal or greater than 120

Employees:salary \geq 120[firstname]

Composition of Operations

Retrieve the name of the employees whose salary is equal or greater than 120

Employees:salary \geq 120[firstname]

As we'll see later (see Section 15) projection and selection have the same precedence, and because operators have left-to-right associativity:

Composition of Operations

Retrieve the name of the employees whose salary is equal or greater than 120

Employees:salary \geq 120[firstname]

As we'll see later (see Section 15) projection and selection have the same precedence, and because operators have left-to-right associativity:

Employees[firstname]:salary \geq 120

is incorrect because the projection applies first, and salary is not an attribute of Employees[firstname]

Chapter 3 – Relational Algebra

- 7 Introduction
- 8 Projection operator
- 9 Selection operator
- 10 Misc
- 11 Joining Relations**
- 12 Set Operators
- 13 Division
- 14 Conclusion

Cartesian Product

- *Notation:* $R \times S$
- *Relation schema:* $X \cup Y$ (assuming $X \cap Y = \emptyset$)
- *Interpretation:* $R \times S = \{ \langle r, s \rangle \mid r \in R \text{ and } s \in S \}$
- *Special case:* $X = \{A, B\}$ and $Y = \{B, C\}$

When R_1 and R_2 have attributes in common, fully qualified attribute names have to be used:

Schema($R_1(A, B) \times R_2(B, C)$) is $\{R_1.A, R_1.B, R_2.B, R_2.C\}$

Examples

Employees X Leaders is a relation whose schema is:

firstname, salary, address, Employees.dept, boss, Leaders.dept

/ As firstname, salary, address and boss appear in one relation only, the qualified notation can be omitted. */*

The cartesian is a means for combining two relations.

Employees X Leaders is:

TheEmployees

TheRespon.

firstname	salary	address	dept	boss	dept
John	120	Randwick	Toys	John	Toys
Mary	143	Wollongong	Furniture	John	Toys
Peter	110	Randwick	Garden	John	Toys
Tom	120	Botany Bay	Toys	John	Toys
Phil	140	Newtown	Furniture	John	Toys
John	120	Randwick	Toys	Mary	Furniture
Mary	143	Wollongong	Furniture	Mary	Furniture
Peter	110	Randwick	Garden	Mary	Furniture
Tom	120	Botany Bay	Toys	Mary	Furniture
Phil	140	Newtown	Furniture	Mary	Furniture
John	120	Randwick	Toys	Peter	Garden
Mary	143	Wollongong	Furniture	Peter	Garden
Peter	110	Randwick	Garden	Peter	Garden
Tom	120	Botany Bay	Toys	Peter	Garden
Phil	140	Newtown	Furniture	Peter	Garden

For each employee find her(his) boss

$A \leftarrow \text{Employees} \times \text{Learders}$

$B \leftarrow A:\text{Leaders.dept} = \text{Employees.dept}$

$\text{Res} \leftarrow B[\text{firstname}, \text{boss}]$

For each employee find her(his) boss

TheEmployees

TheRespon.

firstname	salary	address	dept	boss	dept
John	120	Randwick	Toys	John	Toys
Mary	143	Wollongong	Furniture	John	Toys
Peter	110	Randwick	Garden	John	Toys
Tom	120	Botany Bay	Toys	John	Toys
Phil	140	Newtown	Furniture	John	Toys
John	120	Randwick	Toys	Mary	Furniture
Mary	143	Wollongong	Furniture	Mary	Furniture
Peter	110	Randwick	Garden	Mary	Furniture
Tom	120	Botany Bay	Toys	Mary	Furniture
Phil	140	Newtown	Furniture	Mary	Furniture
John	120	Randwick	Toys	Peter	Garden
Mary	143	Wollongong	Furniture	Peter	Garden
Peter	110	Randwick	Garden	Peter	Garden
Tom	120	Botany Bay	Toys	Peter	Garden
Phil	140	Newtown	Furniture	Peter	Garden

Relational (or theta) Join

- *Notation:* $R(P)*S$
 P is a boolean expression whose operands are attribute names only.
 Simple boolean expressions are of the form $X_i \theta Y_i$ where X_i is an attribute of R and Y_i an attribute of S , and $\theta \in \{=, \neq, <, >, \dots\}$
- *Relation schema:* $X \cup Y$ (assuming $X \cap Y = \emptyset$)
- *Interpretation:*
 $R(P)*S = \{ \langle r, s \rangle \mid r \in R \text{ and } s \in S \text{ and } P(\langle r, s \rangle) \}$
- *Property:* $R(P)*S = (R \times S):P$
- Also noted: $R \bowtie_P S$

Example

For each employee find her(his) boss (again!)

A \leftarrow Employees(dept=dept)*Leaders
/ The schema of A is: firstname, salary, address,
 Employees.dept, boss, Leaders.dept */*

Res \leftarrow A[firstname, boss]

Natural Join

When the relational join involves equality comparison only on attributes whose names are identical it is called *natural join*. In the schema of the result, superfluous attributes have been removed.

For each employee find her(his) boss is thus:

$A \leftarrow \text{Employees} * \text{Leaders}$

/ The schema is: firstname, salary, address, dept, boss */*

$\text{Res} \leftarrow A[\text{firstname}, \text{boss}]$

Natural Join

When the relational join involves equality comparison only on attributes whose names are identical it is called *natural join*. In the schema of the result, superfluous attributes have been removed.

For each employee find her(his) boss is thus:

$A \leftarrow \text{Employees} * \text{Leaders}$

/ The schema is: firstname, salary, address, dept, boss */*

$\text{Res} \leftarrow A[\text{firstname}, \text{boss}]$

A direct expression:

$(\text{Employees} * \text{Leaders})[\text{firstname}, \text{boss}]$

Definition

- Assumption: $R(X_1, \dots, X_n)$ and $S(X_p, \dots, X_m)$ where:
 $A = \{X_1, \dots, X_n\} - \{X_p, \dots, X_m\}$ (attributes in R only)
 $B = \{X_1, \dots, X_n\} \cap \{X_p, \dots, X_m\}$ (attributes in both R and S)
 $C = \{X_p, \dots, X_m\} - \{X_1, \dots, X_n\}$ (attributes in S only)
- Notation:* $R * S$
- Relation schema:* $\{X_1, \dots, X_n\} \cup \{X_p, \dots, X_m\}$
- Interpretation:* $R * S = \{ \langle r[A], r[B], s[C] \rangle \mid r \in R \wedge s \in S \wedge r[B] = s[B] \}$
- Property:* $R * S = (R(R.B_1=S.B_1, \dots, R.B_i=S.B_i)*S)[A, B, C]$
 where $B = \{B_1, \dots, B_i\}$
 $B = \emptyset \implies R * S = \emptyset$

Multiple Citation of Relations

For each employee (given by her(his) firstname) retrieve her(his) firstname and the salary of her(his) boss

$A \leftarrow \text{Employees} * \text{Leaders}[\text{firstname}, \text{boss}]$

/ For each employee, her(his) boss */*

$E(\text{boss}, \text{salary}) \leftarrow \text{Employees}[\text{firstname}, \text{salary}]$

/ a relation where to find salary of employees who might be leaders. */*

$\text{Res} \leftarrow A * E$

Because Employees is used twice, the intermediate relation E is mandatory: there is no straight expression!

Chapter 3 – Relational Algebra

- 7 Introduction
- 8 Projection operator
- 9 Selection operator
- 10 Misc
- 11 Joining Relations
- 12 Set Operators**
- 13 Division
- 14 Conclusion

Definition

- *Notation:* $R(X) \cup S(Y)$, $R(X) \cap S(Y)$, $R(X) - S(Y)$
 $X = X_1, \dots, X_n$ and $Y = Y_1, \dots, Y_p$ and $n=p$ and
 $\forall i \in [1, \dots, n]$, $\text{domain}(X_i)$ and $\text{domain}(Y_i)$ are comparable.
- *Relation schema:* when the attribute names of R and S are pairwise identical the result is a relation with the same schema, otherwise the attribute names of the result must be explicitly defined using the rename operation.

For example: $T(Z_1, Z_2, \dots, Z_n) \leftarrow R \cup S$, where:

$\forall i \in [1, \dots, n]$, $\text{domain}(Z_i) = \text{domain}(X_i) = \text{domain}(Y_i)$

- *Interpretation:*

$$R \cup S = \{ t / t \in R \text{ or } t \in S \}$$

$$R - S = \{ t / t \in R \text{ and } t \notin S \}$$

$$R \cap S = \{ t / t \in R \text{ and } t \in S \}$$

- *Property:* $R \cap S = R - (R - S) = S - (S - R)$

Example 1

Retrieve the employees who are not boss

$R(\text{firstname}) \leftarrow \text{Employees}[\text{firstname}] - \text{Leaders}[\text{boss}]$

Example 1

Retrieve the employees who are not boss

$R(\text{firstname}) \leftarrow \text{Employees}[\text{firstname}] - \text{Leaders}[\text{boss}]$

Question: Why is the following expression incorrect?

$(\text{Employees}(\text{firstname} \neq \text{boss}) * \text{Leaders})[\text{firstname}]$

Employees

Leaders

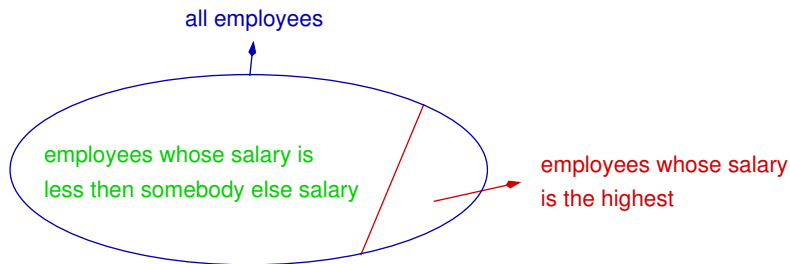
firstname	salary	address	dept	boss	dept
John	120	Randwick	Toys	John	Toys
Mary	143	Wollongong	Furniture	John	Toys
Peter	110	Randwick	Garden	John	Toys
Tom	120	Botany Bay	Toys	John	Toys
Phil	140	Newtown	Furniture	John	Toys
John	120	Randwick	Toys	Mary	Furniture
Mary	143	Wollongong	Furniture	Mary	Furniture
Peter	110	Randwick	Garden	Mary	Furniture
Tom	120	Botany Bay	Toys	Mary	Furniture
Phil	140	Newtown	Furniture	Mary	Furniture
John	120	Randwick	Toys	Peter	Garden
Mary	143	Wollongong	Furniture	Peter	Garden
Peter	110	Randwick	Garden	Peter	Garden
Tom	120	Botany Bay	Toys	Peter	Garden
Phil	140	Newtown	Furniture	Peter	Garden

Example 2

Retrieve the firstname of the employee whose salary is the highest

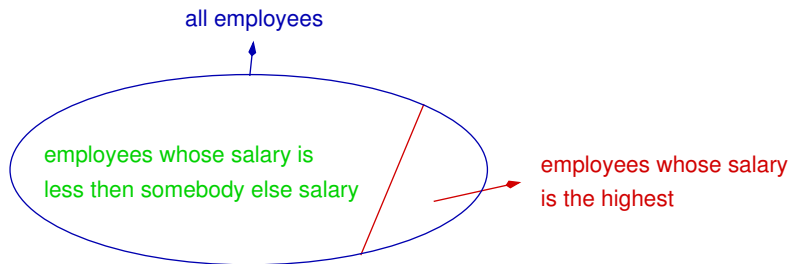
Example 2

Retrieve the firstname of the employee whose salary is the highest



Example 2

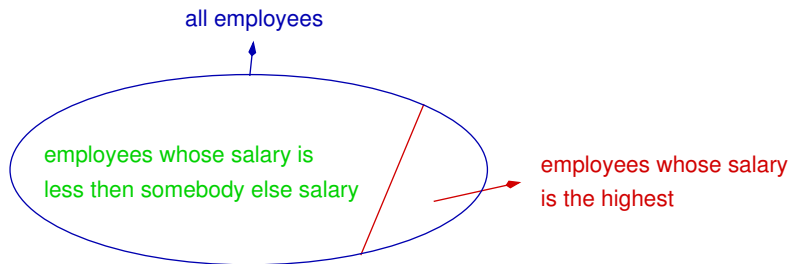
Retrieve the firstname of the employee whose salary is the highest



$E1 \leftarrow \text{Employees}, E2 \leftarrow \text{Employees}$

Example 2

Retrieve the firstname of the employee whose salary is the highest



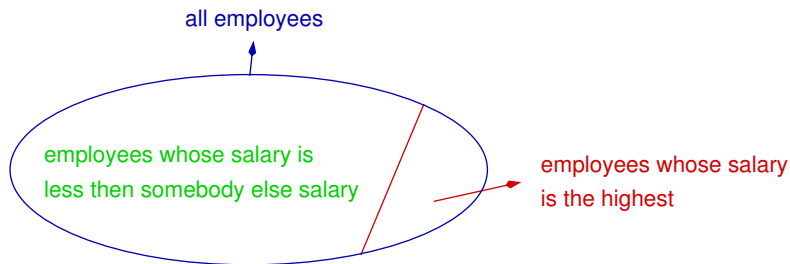
$E1 \leftarrow \text{Employees}, E2 \leftarrow \text{Employees}$

$S \leftarrow (E1(\text{salary} < \text{salary}) * E2)[E1.\text{firstname}]$

/ $\langle x \rangle \in S \iff x$ is an employee who earns less than somebody else. */*

Example 2

Retrieve the firstname of the employee whose salary is the highest



$E1 \leftarrow \text{Employees}, E2 \leftarrow \text{Employees}$

$S \leftarrow (E1(\text{salary} < \text{salary}) * E2)[E1.\text{firstname}]$

/ $\langle x \rangle \in S \iff x$ is an employee who earns less than somebody else. */*

$\text{Res} \leftarrow E1[\text{firstname}] - S$

Chapter 3 – Relational Algebra

- 7 Introduction
- 8 Projection operator
- 9 Selection operator
- 10 Misc
- 11 Joining Relations
- 12 Set Operators
- 13 Division**
- 14 Conclusion

Motivation

Enrolls (student, sport) (the key is $\langle \text{student}, \text{sport} \rangle$) E in short.

Find students who are enrolled in all sports

Step by step:

$E[\text{student}] \times E[\text{sport}]$

/ all possible pairs made of one student and one sport. */*

E */* actual pairs made of one student and one sport. */*

$(E[\text{student}] \times E[\text{sport}] - E)[\text{student}]$

/ students who aren't enrolled in at least one sport */*

$E[\text{student}] - (E[\text{student}] \times E[\text{sport}] - E)[\text{student}]$

/ students who are enrolled in all sports */*

In short: $E / E[\text{sport}]$

Definition

- *Notation* : R / S

We assume that the relation schema of R is X_1, \dots, X_n and the one of S is X_{p+1}, \dots, X_n where $p \neq 0$, $p < n$, and $S \neq \emptyset$.

- *Relation schema*: X_1, \dots, X_p

- *Interpretation*:

$$R / S = \{ \langle x_1, \dots, x_p \rangle / \\ \forall \langle x_{p+1}, \dots, x_n \rangle \in S, \\ \langle x_1, \dots, x_p, x_{p+1}, \dots, x_n \rangle \in R \}$$

- *Property*:

$$R / S = R[X_1, \dots, X_p] - \\ (R[X_1, \dots, X_p] \times S - R)[X_1, \dots, X_p]$$

Another example

Retrieve the firstname of the employee whose salary is the highest

Another example

Retrieve the firstname of the employee whose salary is the highest

$\text{EarnsMore}(P1, P2) \leftarrow (E1(\text{salary} \geq \text{salary}) * E2)[E1.\text{firstname}, E2.\text{firstname}]$
/ $\langle x, y \rangle \in \text{EarnsMore} \iff x$'s salary is higher or equal than y 's salary. */*
 $\text{Res} \leftarrow \text{EarnsMore} / \text{EarnsMore}[P2]$

Chapter 3 – Relational Algebra

- 7 Introduction
- 8 Projection operator
- 9 Selection operator
- 10 Misc
- 11 Joining Relations
- 12 Set Operators
- 13 Division
- 14 Conclusion**

Precedence of Operators

Given from the highest precedence to the lowest:

- Projection, selection, rename
- Cartesian product,
- Natural join, relational join, division
- Set difference
- Union, intersection

All operators have left-to-right associativity. Binary operators, except set difference, are commutative.

Concluding Remarks

Why are we studying such theoretical stuff?

- High level notation lets us focus on reasoning about relations.
- Relational algebra defined the basics of SQL (we'll study SQL later).
- SQL queries are mapped to relational expressions during the query optimisation process.

Chapter 4 – Relational data design

15 Motivations

16 Update Anomalies

17 Functional Dependencies

18 Normalisation

19 Conclusion

As noted earlier, the relational model is Simple, uniform, well-defined, formal, ...

Such properties lead to useful mathematical theories: such as functional dependency (fd) and normalisation.

The basis for yet another approach to relational design:

- *Bottom-up*: unstructured schema, then normalisation (via fds)

Contrast this to the other approach:

- *Top-down*: conceptual (UML CD) design, then conceptual-to-relational mapping

Essentially, functional dependencies

- are a kind of constraint among attributes within a relation
- have implications for "goodness" of relational schema designs

What we study here:

- basic theory and definition of functional dependencies
- methodology for assessing and improving schema design (normalisation)

Chapter 4 – Relational data design

15 Motivations

16 Update Anomalies

17 Functional Dependencies

18 Normalisation

19 Conclusion

Relational Design and Redundancy

A good relational database design:

- Must capture all of the necessary attributes/associations
- Should do so with a minimal data redundancy.
Redundancy leads to difficulties with maintaining consistency during updates

Consider the following relation (Accounts) defining bank accounts and branches:

accNo	bal.	customer	branch	address	assets
101	500	1313131	Downtown	Brooklyn	9000000
102	400	1313131	Perryridge	Horseneck	1700000
113	600	9876543	Round Hill	Horseneck	8000000
101	900	9876543	Brighton	Brooklyn	7100000
215	700	1111111	Mianus	Horseneck	400000
102	700	1111111	Redwood	Palo Alto	2100000
305	350	1234567	Round Hill	Horseneck	8000000

Many redundancies...

Insertion anomaly: insert account 306 at Rd Hill
 we need to check that branch data is consistent with existing tuples

accNo	bal.	customer	branch	address	assets
101	500	1313131	Downtown	Brooklyn	9000000
102	400	1313131	Perryridge	Horseneck	1700000
113	600	9876543	<i>Round Hill</i>	Horseneck	<i>8000000</i>
....					
102	700	1111111	Redwood	Palo Alto	2100000
305	350	1234567	<i>Round Hill</i>	Horseneck	<i>8000000</i>
306	800	1111111	Round Hill	Horseneck	8500000

Update anomaly: update Rd Hill branch address

If a branch's address changes, we need to update all tuples referring to that branch

accNo	bal.	customer	branch	address	assets
101	500	1313131	Downtown	Brooklyn	9000000
102	400	1313131	Perryridge	Horseneck	1700000
113	600	9876543	Round Hill	Palo Alto	8000000
101	900	9876543	Brighton	Brooklyn	7100000
215	700	1111111	Mianus	Horseneck	400000
102	700	1111111	Redwood	Palo Alto	2100000
305	350	1234567	<i>Round Hill</i>	<i>Horseneck</i>	8000000

Deletion anomaly: remove account 101 in Downtown

If we remove information about the last account at a branch, all of the branch information disappears

accNo	bal.	customer	branch	address	assets
<i>101</i>	<i>500</i>	<i>1313131</i>	<i>Downtown</i>	<i>Brooklyn</i>	<i>9000000</i>
102	400	1313131	Perryridge	Horseneck	1700000
113	600	9876543	Round Hill	Horseneck	8000000
101	900	9876543	Brighton	Brooklyn	7100000
215	700	1111111	Mianus	Horseneck	400000
102	700	1111111	Redwood	Palo Alto	2100000
305	350	1234567	Round Hill	Horseneck	8000000

Summary

- To avoid these kinds of update problems: decompose the relation.
- Each relation in the decomposition is about one concept (branch, account, ...)

This is actually the purpose of a conceptual design.

So, why do we need a dependency theory and normalisation procedure to deal with redundancy?

- Normalisation can be viewed as (semi)automated design:
 - determine all of the attributes in the problem domain
 - collect them all together in a *super-relation*
 - provide some simple information about how attributes are related
 - apply normalisation to decompose into non-redundant relations

Normalisation can also be seen as a tool for relation quality assessment

Chapter 4 – Relational data design

- 15 Motivations
- 16 Update Anomalies
- 17 Functional Dependencies**
- 18 Normalisation
- 19 Conclusion

Notations & Terminology

Most texts adopt the following terminology:

- Relation schemas: upper-case letters, denoting set of all attributes (e.g. R, S, P, Q, ...)
- Tuples: lower-case letters (e.g. t, t', t1, u, v, ...)
- Attributes: upper-case letters from start of alphabet (e.g. A, B, C, D, ...)
- Sets of attributes: simple list of attribute names (e.g. $X = ABCD$ rather than $X = A,B,C,D$)
- Tuple component values: `tuple[attrSet]` (e.g. `t[X]`, `t[Y,Z]`)

Definition

A functional dependency between X and Y in R , denoted $X \longrightarrow Y$, specifies a constraint over R :

- $\forall t, u \in R, t[X] = u[X] \implies t[Y] = u[Y]$

In other words, if two tuples in R agree in their values for the set of attributes X , then they must also agree in their values for the set of attributes Y .

- We say that "Y is functionally dependent on X".
- The attribute sets X and Y may overlap, and it is trivially true that $X \longrightarrow X$.

It's worth noting:

- The single arrow \longrightarrow denotes *functional dependency*
- $X \longrightarrow Y$ can also be read as *X determines Y*
- The double arrow \implies denotes *logical implication*

Examples

Few examples: the relation Accounts satisfies:

- $\text{accNo}, \text{branch} \longrightarrow \text{balance}, \text{customer}$
- $\text{branch} \longrightarrow \text{address}, \text{assets}$

But:

- $\text{accNo} \not\rightarrow \text{customer}$
- $\text{customer} \not\rightarrow \text{accNo}$
- $\text{customer} \not\rightarrow \text{branch}$
- $\text{branch} \not\rightarrow \text{customer}$

More DF can be derived..

Inference rules

These rules (6) are known as Amstrong's axioms (1974)

- *R1 : reflexive rule*
if $X \subseteq Y$ then $Y \longrightarrow X$
- *R2 : augmentation rule*
if $X \longrightarrow Y$ then $XZ \longrightarrow YZ$
- *R3 : transitive rule*
if $X \longrightarrow Y$ and $Y \longrightarrow Z$ then $X \longrightarrow Z$

$XYZ \longrightarrow TV$ is a symplified form for
 $\{X, Y, Z\} \longrightarrow \{T, V\}$

- *R4 : decomposition rule*
if $X \longrightarrow Y$ and $Z \subseteq Y$ then $X \longrightarrow Z$
- *R5 : union rule*
if $X \longrightarrow Y$ and $X \longrightarrow Z$ then $X \longrightarrow YZ$
- *R6 : pseudo-transitive rule*
if $X \longrightarrow Y$ and $YW \longrightarrow Z$ then $XW \longrightarrow Z$

Examples

In the relation *Accounts*, a set *F* of FDs is:

- (1) $\text{accNo, branch} \longrightarrow \text{balance}$
- (2) $\text{accNo, branch} \longrightarrow \text{customer}$
- (3) $\text{accNo, branch} \longrightarrow \text{branch}$
- (4) $\text{branch} \longrightarrow \text{address}$
- (5) $\text{branch} \longrightarrow \text{assets}$

From *F* we can derive (among other FDs):

- (6) $\text{accNo, branch} \longrightarrow \text{address}$ (by R3, from 3 and 4)
- (7) $\text{accNo, branch} \longrightarrow \text{assets}$ (by R3, from 3 and 5)

Closure of X under F

Definition: closure of X under F (noted X^+)

- X is set of attributes in a relation R ,
- F is a set of functional dependencies which hold in R
- X^+ is the set of attributes that are functionally determined by X based on F .

Examples

$\{\text{branch}\}$ closure under the set of dependencies which hold in Accounts

- $\{\text{branch}\}^+ = \{\text{branch}, \text{address}, \text{assets}\}$

$\{\text{accNo}, \text{branch}\}$ closure under the set of dependencies which hold in Accounts

- $\{\text{accNo}, \text{branch}\}^+ = \{\text{accNo}, \text{branch}, \text{balance}, \text{customer}, \text{address}, \text{assets}\}$

Key of a relation

- *Full functional dependency*:
 $X \longrightarrow Y$ is FFD $\iff \forall X' \subset X, X' \not\rightarrow Y$
 (Y is fully functionally dependent by X)
- *Key*: let's U be the set of R's attributes. X ($X \subseteq U$) is key in R,
 $\iff X \longrightarrow U$ is a FFD.
- If R has more than one key, each one is called *candidate key*.
- *Prime attribute*: Z is a prime attribute of R if it is a member of some candidate key of R.
- *Non-prime attribute*: Z is non-prime attribute of R, if it is not a member of any candidate key.

Example

- $\text{accNo}, \text{branch} \longrightarrow \text{balance}$ is a *FFD*
- $\text{accNo}, \text{branch} \longrightarrow \text{address}$ is *not a FFD*
- $\text{accNo}, \text{branch}$ is *key* in the relation because:
 $\text{accNo}, \text{branch} \longrightarrow \text{accNo}, \text{branch}, \text{balance}, \text{customer}, \text{address}, \text{assets}$
is a FFD (see the closure of $\{\text{accNo}, \text{branch}\}$ under F)
- both accNo and branch are *prime attributes*
- $\text{balance}, \text{customer}, \text{address}$ and assets are *non-prime attributes*

These properties rely on $\{\text{accNo}, \text{branch}\}^+$ and $\{\text{branch}\}^+$

Chapter 4 – Relational data design

- 15 Motivations
- 16 Update Anomalies
- 17 Functional Dependencies
- 18 Normalisation**
- 19 Conclusion

Aims

- Characterise the level of redundancy in a relational schema
- Provide mechanisms for transforming schemas to remove redundancy

Normalisation relies on functional dependencies.

Normalisation process

Normalisation theory defines six normal forms (NFs).

- Each normal form:
 - involves a set of dependency properties that a schema must satisfy
 - gives guarantees about presence/absence of update anomalies

- The normalisation process:
 - check whether a relation schema is in a particular normal form (xNF)
 - if not, partition into sub-relations where each is closer to xNF
 - repeat until the desired xNF is achieved over all sub-relations

Normal forms: brief history

- First, Second, Third Normal Forms (1NF, 2NF, 3NF) (Codd 1972)
- Boyce-Codd (Kent) Normal Form (BCNF) (1974)
- Fourth Normal Form (4NF) (Zaniolo 1976, Fagin 1977)
- Fifth Normal Form (5NF) (Fagin 1979)

Normal forms are linked to each other

- $5NF \implies 4NF \implies BCNF \implies 3NF \implies 2NF \implies 1NF$
- 1NF permits the most redundancy
- 5NF permits the least redundancy.

First Normal Form

A relation R is in first normal form \iff :

- Its attribute domains are all atomic

In fact, 1NF is part of the formal definition of a relation

<i>Bar</i>	<i>nbBeers</i>	<i>Addr</i>	<i>Drinker</i>
Australia Hotel	1	The Rocks	{John}
Coogee Bay Hotel	4	Coogee	{Adam, John}
Lord Nelson	2	The Rocks	{Gernot, John}
Marble Bar	3	Sydney	{Justin}
Regent Hotel	2	Kingsford	{Justin}

This table is not a relation in first normal form, while Accounts is in 1NF

Second Normal Form

A relation R is in second normal form \iff :

- R is in 1NF
- Every non prime attribute A in R is fully functionally dependent on every key of R .

The relation Accounts is not in 2FN because:

- address (among others) is not fully functionally dependent on the key $\{\text{accountNo}, \text{branch}\}$
(branch \longrightarrow address)

Third Normal Form

A relation R is in third normal form \iff :

- Whenever $X \longrightarrow A$ holds in R and $X \longrightarrow A$ non-trivial (i.e. $A \notin X$), X is a superkey of R (X contains a key) OR is a prime attribute.

Third Normal Form (2nd definition)

We need one more definition:

- An attribute A is *transitively dependent* on X in $R \iff$:

$$\exists Y \not\subseteq X \left\{ \begin{array}{l} A \not\subseteq Y \\ X \longrightarrow Y \\ Y \longrightarrow A \\ Y \not\rightarrow X \end{array} \right.$$

A relation R is in third normal form \iff :

- R is in 2NF
- Every non prime attribute A in R is non transitively dependent on every key of R .

Example: Tutors (student, course, tutor).

Functional dependencies which hold in Tutors are:

- student, course \longrightarrow tutor
- tutor \longrightarrow course

Thus:

- $\{\text{student, course}\}^+ = \{\text{student, course, tutor}\}$
- $\{\text{tutor}\}^+ = \{\text{tutor, course}\}$
- $\{\text{tutor, student}\}^+ = \{\text{student, course, tutor}\}$

Hence:

- student,course and tutor,student are candidate keys.
- Tutors is in 3FN (no non-prime attributes)

Unfortunately, Tutors still has redundancy:

student	course	tutor
Peter	database	John
Peter	french	Mary
Paul	database	John
Mary	database	Alan
Paul	french	Mary

the association (tutor, course) is repeated as many times as there are students enrolled in the course.

Boyce-Codd Normal Form

A relation R is in Boyce-Codd normal form \iff :

- Whenever $X \rightarrow A$ holds in R and $X \rightarrow A$ non-trivial, X is a superkey of R .

Tutors is not in BCNF. because $\text{tutor} \rightarrow \text{course}$ holds in *Tutors* and the left-hand side does not contain any key.

Normal forms: summary

- 1NF all attributes have atomic values
we assume this as part of relational model
- 2NF all non-key attributes fully depend on key
avoids much redundancy still has problems with some fds
- 3NF avoids redundancy related to transitive dependancies between non prime attributes and keys still has redundancy problem
- BCNF has no redundancy, but some fds do not hold anymore
- 4NF removes problems due to multivalued dependencies
- 5NF removes problems due to join dependencies

In practice, BCNF and 3NF are the most important for relational design.
We focus on these two (details of others may be found in any database textbook)

Normalisation Process

Decomposition and non-additive Joins.

Consider the relation:

Accounts (accNo, balance, customer, branch, address, assets)

- $R1 = \text{Accounts}[\text{accNo}, \text{balance}, \text{customer}]$
- $R2 = \text{Accounts}[\text{customer}, \text{branch}, \text{address}, \text{assets}]$
- $R1 * R2 \neq \text{Accounts}$

*$R1 * R2$ has additional spurious tuples*

Such decomposition is called lossy decomposition

The value of R1 * R2 is:

accNo	bal.	customer	branch	address	assets
101	500	1313131	Downtown	Brooklyn	9000000
<i>101</i>	<i>500</i>	<i>1313131</i>	<i>Perryridge</i>	<i>Horseneck</i>	<i>1700000</i>
102	400	1313131	Perryridge	Horseneck	1700000
<i>102</i>	<i>400</i>	<i>1313131</i>	<i>Downtown</i>	<i>Brooklyn</i>	<i>9000000</i>
113	600	9876543	Round Hill	Horseneck	8000000
<i>113</i>	<i>600</i>	<i>9876543</i>	<i>Brighton</i>	<i>Brooklyn</i>	<i>7100000</i>
101	900	9876543	Brighton	Brooklyn	7100000
<i>101</i>	<i>900</i>	<i>9876543</i>	<i>Round Hill</i>	<i>Horseneck</i>	<i>8000000</i>
215	700	1111111	Mianus	Horseneck	400000
<i>215</i>	<i>700</i>	<i>1111111</i>	<i>Redwood</i>	<i>Palo Alto</i>	<i>2100000</i>
102	700	1111111	Redwood	Palo Alto	2100000
<i>102</i>	<i>700</i>	<i>1111111</i>	<i>Mianus</i>	<i>Horseneck</i>	<i>400000</i>
305	350	1234567	Round Hill	Horseneck	8000000

Decomposition with non additive join property

A decomposition $D = \{R_1(X, Y), R_2(X, Z)\}$ of $R(X, Y, Z)$ has the non-additive join property \iff

- $X \longrightarrow Y$ holds in R
- or $X \longrightarrow Z$ holds in R

In DB textbooks, this property is also called lossless join property

Relational decomposition into BCNF relations with lossless join property

- *Step 1:* $D \leftarrow \{R\}$
- *Step 2:* while there is a relation in D that is not in BCNF
do {
 - Choose a relation $Q(X, Y, Z)$ that is not in BCNF
 - Find a FD $X \rightarrow Y$ in Q that *violates* BCNF
 - Replace Q by $Q_1(X, Y)$ and $Q_2(X, Z)$}

Example: consider Tutors (student, course, tutor) (<tutor,student> is another key).

- $\{\text{student, course}\}^+ = \{\text{student, course, tutor}\}$
- $\{\text{tutor}\}^+ = \{\text{course, tutor}\}$
- $\{\text{tutor, student}\}^+ = \{\text{student, course, tutor}\}$

The decomposition of Tutors into BCNF relations with lossless property is:

- Step 1: $D \leftarrow \{\text{Tutors}\}$
- Step 2: Tutors is not in BCNF (see above)
 - $\text{tutor} \rightarrow \text{course}$ violates the BCNF, and tutor itself is not a superkey.
 - Tutors is replaced by R1 (tutor, course) and R2 (tutor, student)
 - Both are in BCNF

Unfortunately, the FD $\text{student, course} \rightarrow \text{tutor}$ is not held any more in any of these relations

Another decomposition process: synthesis algorithm

Closure of F , a set of FD:

- The largest collection of dependencies that can be derived from F (using Armstrong's rules) is called the closure of F and is denoted F^+ .
- $(X \rightarrow Y) \in F^+ \iff Y \subseteq X^+$

C is a minimal cover for F, a set of FD:

- Every dependency in C has a single attribute for its right hand,
- $f \in C, C^+ \neq \{C - \{f\}\}^+$ (using Armstrong's rules, it is not possible to derive f from $C - \{f\}$),
- $f \in F, C^+ = \{C \cup \{f\}\}^+$ (using Armstrong's rules, it is possible to derive f from C).

Relation synthesis algorithm with dependency preservation and lossless property

R is the given relation, F is set of FDs that hold in R.

- Step 1: Find a minimal cover C for F
- Step 2: For each left-hand-side X of a FD in C, create a relation schema with attributes $\{X \cup \{A1\} \cup \{A2\}...\}$ where $X \rightarrow A1, X \rightarrow A2, \dots$ are the only dependencies in C with X left-hand-side.
- Step 3: If none of the resulting relation schemas contains a key of R then create one more relation in D, that contains a key of R.

The resulting relation schemas are 3NF (BCNF is not guaranteed)

Example

In the relation Accounts (accNo, balance, customer, branch, address, assets), a minimal cover is (we have seen that accNo,branch is the key):

- accNo, branch \longrightarrow balance
- accNo, branch \longrightarrow customer
- branch \longrightarrow address
- branch \longrightarrow assets

The previous synthesis algorithm produces:

- R1 (accNo, branch, balance, customer)
- R2 (branch, address, assets)

*Both are in BCNF. All dependencies are preserved and
 $R1 * R2 = \text{Accounts}$*

Chapter 4 – Relational data design

- 15 Motivations
- 16 Update Anomalies
- 17 Functional Dependencies
- 18 Normalisation
- 19 Conclusion**

Why did we study this theoretical stuff?

- Functional dependencies and normalisation: nice tools to assess relation schemas
- Designing schema from scratch, how do we figure out
 - an attribute list to start with?
 - a set of functional dependencies?
- All these questions are answered when we design a UML class diagram.
- Denormalisation: for performance reasons relations may be left in a lower normal form.

Chapter 5 – SQL, a relational language (LMD part)

- 20 Introduction
- 21 Select Statement
- 22 Joining relations
- 23 Set Operators
- 24 Nested queries
- 25 Aggregation and Grouping
- 26 More about sub-queries
- 27 Missing Values
- 28 SQL Types
- 29 Language limitations
- 30 As a conclusion

If you meet the course prerequisite jump directly to Section 7.

Preliminaries

SQL is an ANSI/ISO standard language for querying and manipulating relational data.

Designed to be a "human readable" language comprising:

- Data definition facilities
- Database modification operations
- Relational algebra operations
- Aggregation operations

Preliminaries

SQL is an ANSI/ISO standard language for querying and manipulating relational data.

Designed to be a "human readable" language comprising:

- Data definition facilities
- Database modification operations
- Relational algebra operations
- Aggregation operations

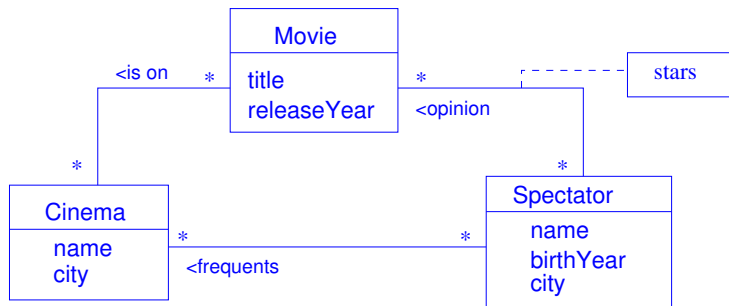
*It is easy to express simple queries
VERY tricky to express complex queries!*

SQL: history

- SQL was developed at IBM during the 1970's, and standardised during the 80's.
- SQL-92 the previous standard
- SQL:1999 the current standard, also called SQL-99
- SQL:2003 was supposed to extend the language with objects.
- Most systems extend the standard language in various dialects.

It appears that SQL (in some form) will survive the rise of object-relational database systems

A Toy databases



Relation specifications

Movies (title, releaseYear) /* *title is the key.* */
 Cinemas (name, city)
 Spectators (name, birthYear, city)
 IsOn (movie, cinema) /* *movie, cinema is the key.* */
 IsOn[movie] \subseteq Movies[title]
 IsOn[cinema] \subseteq Cinemas[name]
 Opinions (spectator, movie, stars)
 domain(stars) = {0, 1, 2, 3, 4, 5}
 Opinions[spectator] \subseteq Spectators[name]
 Opinions[movie] \subseteq Movies[title]
 Frequents (spectator, cinema)
 Frequents[spectator] \subseteq Spectators[name]
 Frequents[cinema] \subseteq Cinemas[name]

Domains need to be detailed.

Relation Values

Opinions

spectator	movie	stars
Marie	The Inbetweeners 2	0
Adrian	The Inbetweeners 2	0
Phil	The Inbetweeners 2	2
Jackie	The Inbetweeners 2	2
Tom	The Inbetweeners 2	5
Alizee	The Inbetweeners 2	4
Lauranne	The Inbetweeners 2	0
Marie	Pretty Woman	5
Adrian	Pretty Woman	4
Phil	Pretty Woman	4
Jackie	Pretty Woman	3
Tom	Pretty Woman	5
Alizee	Pretty Woman	4
Marie	Edward Scissorhands	3
Adrian	Edward Scissorhands	4

IsOn

cinema	movie
Hoyts CBD	Guardian
Hoyts	Guardian
Event Cinema Myer	Guardian
Event Cinema	Guardian
Birch Carroll and Coyles	Guardian
Hoyts CBD	Crooks i
Hoyts	Crooks i
Event Cinema Myer	Crooks i
Event Cinema	Crooks i

Notion of query

A query is a *declarative program* that is meant to retrieve data from a database.

2 modes:

- Interpreted: the result is displayed on the output
- Embedded in an other program: the result is processed by the program itself.

Notion of query

A query is a *declarative program* that is meant to retrieve data from a database.

2 modes:

- Interpreted: the result is displayed on the output
- Embedded in an other program: the result is processed by the program itself.

Example:

```
select name, city  
from Spectators  
where city = 'Sydney'
```

```
/* projection */  
/* relation joins */  
/* selection */
```

Notion of query

A query is a *declarative program* that is meant to retrieve data from a database.

2 modes:

- Interpreted: the result is displayed on the output
- Embedded in an other program: the result is processed by the program itself.

Example:

```
select name, city
from Spectators
where city = 'Sydney'
```

```
/* projection */
/* relation joins */
/* selection */
```

Algebraic expression:

$(\text{Spectators:city} = \text{'Sydney'})[\text{name, city}]$

When interacting with Oracle

```
SQL> select name, city  
2 from Spectators  
3 where city = 'Sydney';
```

NAME	CITY
Marie	Sydney
Phil	Sydney
Jackie	Sydney

3 rows selected

```
SQL>
```

Chapter 5 – SQL, a relational language (LMD part)

- 20 Introduction
- 21 Select Statement**
- 22 Joining relations
- 23 Set Operators
- 24 Nested queries
- 25 Aggregation and Grouping
- 26 More about sub-queries
- 27 Missing Values
- 28 SQL Types
- 29 Language limitations
- 30 As a conclusion

Projection, selection

Retrieve the names and addresses of spectators who were born before 1990

The algebraic expression is:

$(\text{Spectators:birthYear} < 1990)[\text{name,city}]$

Projection, selection

Retrieve the names and addresses of spectators who were born before 1990

The algebraic expression is:

$(\text{Spectators:birthYear} < 1990)[\text{name,city}]$

In SQL:

```
select name, city      /* projection */
from Spectators       /* relations (one or more) */
where birthYear < 1990 /* selection */
```


In *select ... from ... where P*, *P* is a boolean condition:

- A simple condition such as
 <att. name> <comp. op.> <att. name> or
 <att. name> <comp. op.> <lit. value>

In *select ... from ... where P*, *P* is a boolean condition:

- A simple condition such as
 <att. name> <comp. op.> <att. name> or
 <att. name> <comp. op.> <lit. value>
- A composed condition such as
 <cond.> <obool. op.> <cond.>
 where <cond.> is either a simple condition or a composed condition.

In *select ... from ... where P*, *P* is a boolean condition:

- A simple condition such as
 $\langle \text{att. name} \rangle \langle \text{comp. op.} \rangle \langle \text{att. name} \rangle$ or
 $\langle \text{att. name} \rangle \langle \text{comp. op.} \rangle \langle \text{lit. value} \rangle$
- A composed condition such as
 $\langle \text{cond.} \rangle \langle \text{obool. op.} \rangle \langle \text{cond.} \rangle$
 where $\langle \text{cond.} \rangle$ is either a simple condition or a composed condition.

Comparison operators: $=$, \neq , $<$, $>$, etc.

Boolean operators : not, and, or

P might contain brackets.

How do the following expressions evaluate?

- A and B or C
- A and (B or C)
- not A and A
- A and not A
- not A and C

when A=true, B=false, C=true

Duplicated values

Retrieve the movies whose rating is 2

According to the relational data model:

(Opinions:stars=2)[movie]

The expected result is:

MOVIE
The Inbetweeners 2
Lucy

Duplicated values

Retrieve the movies whose rating is 2

According to the relational data model:

(Opinions:stars=2)[movie]

The expected result is:

MOVIE
The Inbetweeners 2
Lucy

In SQL:

```
select movie from Opinions
where stars = 2
```

The actual result is:

MOVIE
The Inbetweeners 2
The Inbetweeners 2
Lucy

Eliminating Duplicates

To avoid duplicated values:

```
select distinct movie from Opinions where stars = 2
```

Eliminating Duplicates

To avoid duplicated values:

```
select distinct movie from Opinions where stars = 2
```

Needs to be carefully used:

Eliminating Duplicates

To avoid duplicated values:

```
select distinct movie from Opinions where stars = 2
```

Needs to be carefully used:

- Think about the cost of this operation!

Eliminating Duplicates

To avoid duplicated values:

```
select distinct movie from Opinions where stars = 2
```

Needs to be carefully used:

- Think about the cost of this operation!
- Might be unnecessary: select name from Spectators

Star (*) Convention

The symbol * denotes a list of all attributes.

Example: *Retrieve information about Spectators*

Algebraic expression: Spectators

In SQL: select * from Spectators

similar to select name, birthYear, city from Spectators

Result:

name	birthYear	city
Marie	1970	Sydney
Adrian	1950	Cairns
Phil	1960	Sydney
Jackie	1965	Sydney
Tom	1986	Brisbane
Alizee	1988	Alice Spring
Lauranne	1986	Amsterdam

- Handy for displaying both schema and value of relations
- Not to be used in queries embedded in a program

Renaming Attributes in the Select

Renaming of attributes is implemented via the AS clause within the select statement

`select name as spectator, birthYear, city from Spectators`
Has the result:

spectator	birthYear	city
Marie	1970	Sydney
Adrian	1950	Cairns
Phil	1960	Sydney
Jackie	1965	Sydney
Tom	1986	Brisbane
Alizee	1988	Alice Spring
Lauranne	1986	Amsterdam

Ordering rows

The clause order by applies only on attributes contained in the clause select:

```
select spectator, movie, stars
```

```
order by stars desc, movie asc, spectator asc
```

Has the result:

Adrian	Australia	5
Alizee	Crooks in Clover	5
Lauranne	Crooks in Clover	5
Marie	Crooks in Clover	5
Tom	Crooks in Clover	5
Lauranne	Edward Scissorhands	5
Adrian	I, Robot	5
Marie	I, Robot	5
Alizee	Lucy	5
Lauranne	Lucy	5
Phil	Lucy	5
Marie	Pretty Woman	5
Tom	Pretty Woman	5

Chapter 5 – SQL, a relational language (LMD part)

- 20 Introduction
- 21 Select Statement
- 22 Joining relations**
- 23 Set Operators
- 24 Nested queries
- 25 Aggregation and Grouping
- 26 More about sub-queries
- 27 Missing Values
- 28 SQL Types
- 29 Language limitations
- 30 As a conclusion

Inner joins

```
select A1, A2, ..., An  
from R1 join R2 on ... join Rp on...  
where C
```

/ → to express a projection */*
/ → to express relation joins */*
/ → to express a selection */*

Inner joins

```
select A1, A2, ..., An
from R1 join R2 on ... join Rp on...
where C
```

/ → to express a projection */*
/ → to express relation joins */*
/ → to express a selection */*

For each movie which is on at Hoyts, find its title and release date

Inner joins

```
select A1, A2, ..., An
from R1 join R2 on ... join Rp on...
where C
```

/ → to express a projection */*
/ → to express relation joins */*
/ → to express a selection */*

For each movie which is on at Hoyts, find its title and release date

In the algebra:

$(\text{Movies}(\text{title} = \text{movie}) * (\text{IsOn:Cinema} = \text{'Hoyts'}))[\text{title}, \text{releaseYear}]$

Inner joins

select A1, A2, ..., An	<i>/* → to express a projection */</i>
from R1 join R2 on ... join Rp on...	<i>/* → to express relation joins */</i>
where C	<i>/* → to express a selection */</i>

For each movie which is on at Hoyts, find its title and release date

In the algebra:

$(\text{Movies}(\text{title} = \text{movie}) * (\text{IsOn}:\text{Cinema} = \text{'Hoyts'}))[\text{title}, \text{releaseYear}]$

In SQL:

select title, releaseYear	<i>/* The projection */</i>
from Movies join IsOn on (title=movie)	<i>/* The join and its condition(s) */</i>
where cinema ='Hoyts'	<i>/* The selection condition */</i>
<i>/* The clause on may contain any kind of predicate. */</i>	

To express joins between two relations (or more).

Name Clashes

When the query refers more than once the same name

For each movies which are on at Hoyts, retrieve its title and the rating gave by Adrian

```
select movie, stars
from Opinions join IsOn on (movie = movie)
where cinema = 'Hoyts' and spectator = 'Adrian'
```

ERROR at line 1:
ORA-00918: column ambiguously defined

Qualified notation

In fact, full attribute names should be fully qualified by a relation name:

```
select IsOn.movie, Opinions.stars  
from Opinions join IsOn on (Opinions.movie = IsOn.movie)  
where IsOn.cinema = 'Hoyts' and Opinions.spectator = 'Adrian'
```

- Each attribute is qualified by the relation it belongs to (easier to read)
- Necessary for attributes defined in more than one relation in the `from` statement.

Natural join: form 1

The joining condition applies on all attributes in common

Algebraic expression:

$((\text{Opinions:spectator} = \text{'Adrian'}) * (\text{IsOn:cinema} = \text{'Hoyts'}))[\text{movie, stars}]$

select movie, stars

/ The attribut movie appears only once in th*

from Opinions natural join IsOn

/ movie appears in both relations */*

where cinema='Hoyts' and spectator='Adrian'

movie (which is defined in both relations) cannot be qualified.

Natural join: form 2

The joining condition does not apply on all attributes in common

Algebraic expression:

$(\text{Spectators}(\text{city}=\text{city}) * \text{Cinemas})[\text{Spectators.name}, \text{Spectators.city}, \text{Cinemas.name}]$

What does the following query return?

Algebraic expression:

$(\text{Spectators} * \text{Cinemas})[\text{name}, \text{city}]$

select Spectators.name, city, Cinemas.name

/ city cannot be qualified by any relation*

from Spectators join Cinemas using (city)

/ The join applies on city only */*

A variant:

select Spectators.name, Spectators.city, Cinemas.name

from Spectators join Cinemas on (Spectators.city=Cinemas.city)

/ The joining condition is explicit. */*

Mixing natural and relational joins

For each movie, retrieve its name, and release year, the cinemas where its on, and for each one who has rated it more than 4 stars, the person's name and address.

R1 (movie,relYear,cinema,name) \leftarrow
 (Movies(name=movie)*IsOn * (Opinions:stars \geq 4))[movie, releaseYear, cinema, s

Res \leftarrow (R1 * Spectators)[movie, releaseYear, cinema, spectator, address]

/ Operators are evaluated form left to right */*

select movie, releaseYear, cinema, spectator, address

from Movies join IsOn on (M.name=I.movie)

natural join Opinions

join Spectators on (O.spectator=S.name) where stars \geq 4

Mixing natural and relational joins

For each movie, retrieve its name, and release year, the cinemas where its on, and for each one who has rated it more than 4 stars, the person's name and address.

```
R1 (movie,relYear,cinema,name) ←  
  (Movies(name=movie)*IsOn * (Opinions:stars≥4))[movie, releaseYear, cinema, s
```

```
Res ← (R1 * Spectators)[movie, releaseYear, cinema, spectator, address]
```

/ Operators are evaluated form left to right */*

```
select movie, releaseYear, cinema, spectator, address  
from Movies join IsOn on (M.name=I.movie)  
    natural join Opinions  
    join Spectators on (O.spectator=S.name) where stars ≥ 4
```

/ What happens with the query below? */*

```
select movie, releaseYear, cinema, spectator, address  
from Spectators S join Opinions O on (S.name=O.spectator)  
    natural join Movies M
```


Mixing natural and relational joins

For each movie, retrieve its name, and release year, the cinemas where its on, and for each one who has rated it more than 4 stars, the person's name and address.

```
R1 (movie,relYear,cinema,name) ←  
  (Movies(name=movie)*IsOn * (Opinions:stars≥4))[movie, releaseYear, cinema, s
```

```
Res ← (R1 * Spectators)[movie, releaseYear, cinema, spectator, address]
```

/ Operators are evaluated form left to right */*

```
select movie, releaseYear, cinema, spectator, address  
from Movies join IsOn on (M.name=I.movie)  
    natural join Opinions  
    join Spectators on (O.spectator=S.name) where stars ≥ 4
```

/ What happens with the query below? */*

```
select movie, releaseYear, cinema, spectator, address  
from Spectators S join Opinions O on (S.name=O.spectator)  
    natural join Movies M
```

Mixing natural and relational joins

For each movie, retrieve its name, and release year, the cinemas where its on, and for each one who has rated it more than 4 stars, the person's name and address.

```
R1 (movie,relYear,cinema,name) ←  
  (Movies(name=movie)*IsOn * (Opinions:stars≥4))[movie, releaseYear, cinema, s
```

```
Res ← (R1 * Spectators)[movie, releaseYear, cinema, spectator, address]
```

/ Operators are evaluated form left to right */*

```
select movie, releaseYear, cinema, spectator, address  
from Movies join IsOn on (M.name=I.movie)  
    natural join Opinions  
    join Spectators on (O.spectator=S.name) where stars ≥ 4
```

/ What happens with the query below? */*

```
select movie, releaseYear, cinema, spectator, address  
from Spectators S join Opinions O on (S.name=O.spectator)  
    natural join Movies M
```

Cartesian Product

Cinemas X (Spectators.name='Phil')

```
select .... from Cinemas cross join Spectators  
where Spectators.name = 'Phil'
```

Joining relations: summary

Let R and S be defined as: $R(\underline{X}, Y, Z)$ and $S(\underline{Y}, Z, T)$

- Cross join: $\text{from } R \text{ cross join } S$
 schema: $R.X, R.Y, R.Z, S.Y, S.Z, S.T$
- Relational Join: $\text{from } R \text{ join } S \text{ on } (P)$
 schema: $R.X, R.Y, R.Z, S.Y, S.Z, S.T$
 P is a valid predicate on $R.X, R.Y, R.Z, S.Y, S.Z, S.T$
 Joining condition is: P
- Natural Join (form 1): $\text{from } R \text{ natural join } S$
 schema: $R.X, Y, Z, S.T$
 Joining condition is $R.Y = S.Y$ and $R.Z = S.Z$
- Natural Join (form 2): $\text{from } R \text{ natural join } S \text{ using } (Y)$
 schema: $R.X, Y, R.Z, S.Z, S.T$
 Joining condition is $R.Y = S.Y$

Join operators as defined in SQL99 standard are not supported by all systems. In this case, the preceding query is:

```
select Cinemas.name, Cinemas.city, Frequents.spectator      /* projection */
from  Cinemas, Frequents                                    /* cartesian product */
where Frequents.cinema = Cinemas.name and                    /* joining condition */
      Cinemas.name = 'Hoyts'                                /* selection */
```

Joining conditions and selection are mixed up, which violates the good practices in Software Engineering³

³a.k.a. separation of concern.

Multiple citations of the same relation

Retrieve pairs of different spectators who have rated the same movie. Give also, the movie name and how much stars each one gave to this movie.

O1 \leftarrow Opinions

O2 \leftarrow Opinions

O3 \leftarrow O1(O1.movie = O2.movie and O1.spectator \neq O2.spectator)*O2

Res \leftarrow O3[O1.spectator, O1.movie, O1.stars, O2.spectator, O2.stars]

Multiple citations of the same relation

Retrieve pairs of different spectators who have rated the same movie. Give also, the movie name and how much stars each one gave to this movie.

O1 \leftarrow Opinions

O2 \leftarrow Opinions

O3 \leftarrow O1(O1.movie = O2.movie and O1.spectator \neq O2.spectator)*O2

Res \leftarrow O3[O1.spectator, O1.movie, O1.stars, O2.spectator, O2.stars]

Relations are renamed in the from clause, qualified notation is used to refer to attributes:

```
select O1.spectator, O1.movie, O1.stars, O2.spectator, O2.stars
from Opinions O1 join Opinions O2
    on (O1.movie = O2.movie and O1.spectator  $\neq$  O2.spectator)
```

Multiple citations of the same relation

Retrieve pairs of different spectators who have rated the same movie. Give also, the movie name and how much stars each one gave to this movie.

O1 \leftarrow Opinions

O2 \leftarrow Opinions

O3 \leftarrow O1(O1.movie = O2.movie and O1.spectator \neq O2.spectator)*O2

Res \leftarrow O3[O1.spectator, O1.movie, O1.stars, O2.spectator, O2.stars]

Relations are renamed in the from clause, qualified notation is used to refer to attributes:

```
select O1.spectator, O1.movie, O1.stars, O2.spectator, O2.stars
from Opinions O1 join Opinions O2
    on (O1.movie = O2.movie and O1.spectator  $\neq$  O2.spectator)
```

Questions for fun:

- Is the clause *select distinct* necessary to eliminate duplicates?
- How to ensure antisymmetry?

R is antisymmetric if $\langle X, Y \rangle \in R \implies (\langle Y, X \rangle \notin R \text{ or } X = Y)$

We already know that for all X, $\langle X, X \rangle \notin R$

Multiple citations of the same relation

Retrieve pairs of different spectators who have rated the same movie. Give also, the movie name and how much stars each one gave to this movie.

O1 \leftarrow Opinions

O2 \leftarrow Opinions

O3 \leftarrow O1(O1.movie = O2.movie and O1.spectator \neq O2.spectator)*O2

Res \leftarrow O3[O1.spectator, O1.movie, O1.stars, O2.spectator, O2.stars]

Relations are renamed in the from clause, qualified notation is used to refer to attributes:

```
select O1.spectator, O1.movie, O1.stars, O2.spectator, O2.stars
from Opinions O1 join Opinions O2
    on (O1.movie = O2.movie and O1.spectator  $\neq$  O2.spectator)
```

Questions for fun:

- Is the clause *select distinct* necessary to eliminate duplicates?
- How to ensure antisymmetry?

R is antisymmetric if $\langle X, Y \rangle \in R \implies (\langle Y, X \rangle \notin R \text{ or } X = Y)$

We already know that for all X, $\langle X, X \rangle \notin R$

Chapter 5 – SQL, a relational language (LMD part)

- 20 Introduction
- 21 Select Statement
- 22 Joining relations
- 23 Set Operators**
- 24 Nested queries
- 25 Aggregation and Grouping
- 26 More about sub-queries
- 27 Missing Values
- 28 SQL Types
- 29 Language limitations
- 30 As a conclusion

Union, Intersection, Difference

In the following Q1 and Q2 are queries such as `select... from....` whose schemas are compatible:

- `select A, B from...` is not compatible with `select C from...`
- `select A, B from...` is compatible with `select C, D from...` only if A and C are comparable as well as for B and D.

These rules are those which apply on algebraic set operators.

Operators

- Union [all]
Duplicates not eliminated when all present.
- Intersect (stands for intersection)
- Minus (stands for difference)

Results contain no duplicated values except for union all.

Minus: example

Find movies rated 5 stars which are not on at Hoyts.

$(\text{Opinions:stars} = 5)[\text{movie}] - (\text{IsOn:cinema} = \text{'Hoyts'})[\text{movie}]$

select movie from Opinions where stars = 5

minus

select movie from IsOn where cinema = 'Hoyts'

Both queries must return relations whose schemas are compatible.

Minus: example

Find movies rated 5 stars which are not on at Hoyts.

$(\text{Opinions:stars} = 5)[\text{movie}] - (\text{IsOn:cinema} = \text{'Hoyts'})[\text{movie}]$

select movie from Opinions where stars = 5

minus

select movie from IsOn where cinema = 'Hoyts'

Both queries must return relations whose schemas are compatible.

Intersection: example

Find the movies and spectators such that the spectator has rated the movie with more than 4 stars and frequents a cinema that has it on.

$(\text{Opinions:stars} \geq 4)[\text{spectator, movie}] \cap (\text{IsOn} * \text{Frequents})[\text{spectator, movie}]$

select spectator, movie from Opinions where stars \geq 4

intersect

select spectator, movie from IsOn natural join Frequents

A variant:

$(\text{Opinions:stars} \geq 4) * \text{IsOn} * \text{Frequents})[\text{spectator, movie}]$

select distinct spectator, movie

Union: example

Find the spectators who have rated "Crooks in Clover" 5 stars or frequent the Hoyts

select spectator from Opinions

where movie = 'Crooks in Clover' and stars=5

union

→

select spectator from Frequents

where cinema = 'Hoyts'

spectator
Marie
Tom
Alizée
Lauranne
Adrian

Union: example

Find the spectators who have rated "Crooks in Clover" 5 stars or frequent the Hoyts

select spectator from Opinions

where movie = 'Crooks in Clover' and stars=5

union

→

select spectator from Frequents

where cinema = 'Hoyts'

spectator
Marie
Tom
Alizée
Lauranne
Adrian

select spectator from Opinions

where movie = 'Crooks in Clover' and stars=5

union all

→

select spectator from Frequents

where cinema = 'Hoyts'

spectator

Chapter 5 – SQL, a relational language (LMD part)

- 20 Introduction
- 21 Select Statement
- 22 Joining relations
- 23 Set Operators
- 24 Nested queries**
- 25 Aggregation and Grouping
- 26 More about sub-queries
- 27 Missing Values
- 28 SQL Types
- 29 Language limitations
- 30 As a conclusion

Queries in the from clause

Principle: the from clause accepts SQL queries as well as relation names

For each movie rated 5 stars which is not on at Hoyts, return its name and year of release.

We know that the following query returns movies rated 5 stars which are not on at Hoyts:

```
select movie from Opinions where stars = 5
minus
select movie from IsOn where cinema = 'Hoyts'
```

Let's call R the corresponding relation.

Hence the query is (using R):

```
select title, releaseYear
from Movies join R on (movie = title)
```

Now, change R with the query that returns it:

```
select title, releaseYear
from Movies join (select movie from Opinions where stars = 5
                  minus
                  select movie from IsOn where cinema = 'Hoyts') R
  /* < m > ∈ R ⇔ m is movie rated 5 stars which is not on at
  Hoyts. */
on (movie = title)
```

Hence the query is (using R):

```
select title, releaseYear
from Movies join R on (movie = title)
```

Now, change R with the query that returns it:

```
select title, releaseYear
from Movies join (select movie from Opinions where stars = 5
                  minus
                  select movie from IsOn where cinema = 'Hoyts') R
  /* < m > ∈ R ⇔ m is movie rated 5 stars which is not on at
     Hoyts. */
on (movie = title)
```

The sub-query must be named.

Advantages

- Split the query into (simpler) queries.
- Code and test the sub-queries independently from each others.

Rule: the sub-query nested in the from clause must be specified

Advantages

- Split the query into (simpler) queries.
- Code and test the sub-queries independently from each others.

Rule: the sub-query nested in the from clause must be specified

Clumsy use:

```
select title, releaseYear
from (select distinct movie from Opinions where stars = 5) X
/* < m > ∈ X ⇔ m is a movie rated 5 stars, at least once. */
join Movies in (movie = title)
```

Advantages

- Split the query into (simpler) queries.
- Code and test the sub-queries independently from each others.

Rule: the sub-query nested in the from clause must be specified

Clumsy use:

```
select title, releaseYear
from (select distinct movie from Opinions where stars = 5) X
/* < m > ∈ X ⇔ m is a movie rated 5 stars, at least once. */
join Movies in (movie = title)
```

A more concise expression, easier to read:

```
select distinct title, releaseYear
from Opinions join Movies on (movie=title)
where stars=5
```

Chapter 5 – SQL, a relational language (LMD part)

- 20 Introduction
- 21 Select Statement
- 22 Joining relations
- 23 Set Operators
- 24 Nested queries
- 25 Aggregation and Grouping**
- 26 More about sub-queries
- 27 Missing Values
- 28 SQL Types
- 29 Language limitations
- 30 As a conclusion

Aggregation

To reduce a list of values to one value.

- Count (*) \rightarrow number of tuples
- Count (A) \rightarrow number of values in A
- Count (distinct A) \rightarrow number of different values in A
- Avg (A) \rightarrow average value of values in A
- Min (A) (resp. Max) \rightarrow minimum (resp. maximum) value of values in A
- Sum (A) \rightarrow sum value of values in A

Examples

What is the average rating given to 'Australia'?

Examples

What is the average rating given to 'Australia'?

```
select      stars from Opinions where movie = 'Australia'
```

Examples

What is the average rating given to 'Australia'?

```
select avg (stars) from Opinions where movie = 'Australia'
```

Examples

What is the average rating given to 'Australia'?

`select avg (stars) from Opinions where movie = 'Australia'`

How many cinemas are located in Brisbane?

Examples

What is the average rating given to 'Australia'?

```
select avg (stars) from Opinions where movie = 'Australia'
```

How many cinemas are located in Brisbane?

```
select      *  from Cinemas where city = 'Brisbane'
```

Examples

What is the average rating given to 'Australia'?

```
select avg (stars) from Opinions where movie = 'Australia'
```

How many cinemas are located in Brisbane?

```
select count (*) from Cinemas where city = 'Brisbane'
```

How many movies are rated more than 2?

```
select count (movie) from Opinions where stars > 2
```


How many movies are rated more than 2?

```
select count (movie) from Opinions where stars > 2
```

The query above is incorrect, why?

How many movies are rated more than 2?

```
select count (movie) from Opinions where stars > 2
```

The query above is incorrect, why?

It counts a movie as many times there exists a spectator who rated it more than 2 stars

How many movies are rated more than 2?

```
select count (movie) from Opinions where stars > 2
```

The query above is incorrect, why?

It counts a movie as many times there exists a spectator who rated it more than 2 stars

A correct expression is:

```
select count (distinct movie) from Opinions where stars > 2
```

Give names of spectators who rated the movie Australia with the less stars

```
select spectator
```

```
from Opinions join
```

```
  (select min(stars) as minS from Opinions
```

```
   where movie='Australia') Min
```

```
/* < s > ∈ Min ⇔ s is the minimum number of stars given to Australia.
```

```
*/
```

```
  on (stars = minS)
```

```
where movie='Australia'
```

Grouping

Partition a relation to apply an aggregation on each class separately

Consider the query: *How many cinemas does each spectator frequent?*

Expected result:

spectator	cinema
Adrian	2
Alizée	1
Jackie	2
Marie	2
Phil	2
Tom	2

Processing a grouping

1. *Build the partition according to the spectator value*

Adrian	Hoyts CBD
Adrian	Hoyts
Alizee	Event Cinema
Jackie	Hoyts CBD
Jackie	Event Cinema
Marie	Hoyts
Marie	Hoyts CBD
Phil	Event Cinema
Phil	Hoyts CBD
Tom	Hoyts
Tom	Birch Carroll and Coyles

Processing a grouping

1. Build the partition according to the spectator value

Adrian	Hoyts CBD
Adrian	Hoyts
Alizee	Event Cinema
Jackie	Hoyts CBD
Jackie	Event Cinema
Marie	Hoyts
Marie	Hoyts CBD
Phil	Event Cinema
Phil	Hoyts CBD
Tom	Hoyts
Tom	Birch Carroll and Coyles

select ...
 In SQL: from Likes
 group by spectator

2. *Reduce to one tuple only each class of this partition*

list of spectator (partition criteria)

→ spectator (one of the values)

list of cinema

→ integer (the number of values)

2. *Reduce to one tuple only each class of this partition*

list of spectator (partition criteria)

→ spectator (one of the values)

list of cinema

→ integer (the number of values)

In SQL:

```
select spectator, count (cinema)
from Frequents
group by spectator
```


Impact of partitioning on the select clause

In queries that contain the group by clause, the select clause contains only:

- One or more attributes among those present in the partition criteria
- One or more aggregations which apply on other attributes

How about, if we need to add an attribute in the select clause to answer the query?

Incorrect query:

```
select spectator, city, count(cinema)
from Frequents join Spectators on (spectator=name)
group by spectator
```

Correct query:

```
select spectator, city, count(cinema)
from Frequents join Spectators on (spectator=name)
group by spectator, city
```

Filtering a partition

The clause `having`, which comes with the clause `group by` is meant to filter classes resulting from the partition.

For each spectator who has rated more than 2 movies, retrieve the maximum rating he/she gave and the number of cinema he/she frequents.

Filtering a partition

The clause having, which comes with the clause group by is meant to filter classes resulting from the partition.

For each spectator who has rated more than 2 movies, retrieve the maximum rating he/she gave and the number of cinema he/she frequents.

from Opinions natural join Frequents F

Filtering a partition

The clause having, which comes with the clause group by is meant to filter classes resulting from the partition.

For each spectator who has rated more than 2 movies, retrieve the maximum rating he/she gave and the number of cinema he/she frequents.

from Opinions natural join Frequents F
group by spectator

Filtering a partition

The clause `having`, which comes with the clause `group by` is meant to filter classes resulting from the partition.

For each spectator who has rated more than 2 movies, retrieve the maximum rating he/she gave and the number of cinema he/she frequents.

from Opinions natural join Frequents F
group by spectator
having count (distinct movie) > 2

Filtering a partition

The clause `having`, which comes with the clause `group by` is meant to filter classes resulting from the partition.

For each spectator who has rated more than 2 movies, retrieve the maximum rating he/she gave and the number of cinema he/she frequents.

```
select spectator,
       max(stars) as maxRating,
       count (distinct cinema) as nbCines
from Opinions natural join Frequents F
group by spectator
having count (distinct movie) > 2
```

More examples

Give spectators who have rated all movies (limit the query to movies rated at least once).

Algebraic expression:

$\text{Opinions}[\text{spectator}, \text{movie}] / \text{Opinions}[\text{movie}]$

No division in SQL...

However, we know:

$$|A| = |B| \wedge A \subseteq B \implies A = B$$

For each spectator, how many rated movies?

```
select spectator, count(movie) as nbM from Opinions group by spectator
```

How many movies in total?

```
select count(distinct movie) as nbTot from Opinions
```

Hence, the query is:

```
select spectator
```

```
from (select spectator, count(movie) as nbM from Opinions group by spectator) X1
join (select count(distinct movie) as nbTot from Opinions) X2
on (nbM=nbTot)
```


Give cinemas which are frequented by the most spectators.

- For each cinema, how many spectators?
select cinema, count(spectators) as nbS from Frequents group by cinema
- In the result returned by the query above (let's call it Q), which cinema has the most spectators.
select cinema from Q join (select max(nbS) as maxS from Q) R
on (nbS = maxS)

Eventually, the query is:

```
select cinema
from (select cinema, count(spectator) as nbS from IsOn group by spectator) X1
join (select max(count(spectator)) as nbM
      from IsOn group by spectator) Y
on (nbS = nbM)
```

Yet another expression:

```
select cinema
from (select cinema, count(spectator) as nbS from Frequents group by cinema) Q
join (select max(count(spectator)) as maxS
      from Frequents group by cinema ) Y
on (nbS = maxS)
```

For each spectator who has rated all movies on at Hoyts, return his name and the cinemas he/she frequents.

Algebraic expression:

$R1 \leftarrow \text{Opinions}[\text{spectator}, \text{movie}] / (\text{Opinion} * \text{IsOn:cinema} = \text{'Hoyts'})[\text{movie}]$

$\text{Res} \leftarrow (R1(\text{spectator} = \text{name}) * \text{Frequents})[\text{spectator}, \text{cinema}]$

For each spectator who has rated all movies on at Hoyts, return his name and the cinemas he/she frequents.

Algebraic expression:

$R1 \leftarrow \text{Opinions}[\text{spectator}, \text{movie}] / (\text{Opinion} * \text{IsOn:cinema} = \text{'Hoyts'})[\text{movie}]$

$\text{Res} \leftarrow (R1(\text{spectator} = \text{name}) * \text{Frequents})[\text{spectator}, \text{cinema}]$

/ How many movies are on at Hoyts? */*

select count(movie) as nbTot from IsOn

where cinema = 'Hoyts'

/ How many movies each spectator has rated? */*

select spectator, count(movie) as nbM

from Opinions group by spectator

/ Hence: */*

select spectator, cinema

from (select count(movie) as nbTot from IsOn

where cinema = 'Hoyts') X1

/ < t > ∈ X1 ⇔ t is the total number of movies on at Hoyts. */*

join

(select spectator, count(movie) as nbM

The correct expression is:

```

select spectator, cinema
from (select count(movie) as nbTot from IsOn
      where cinema = 'Hoyts') X1
/* < t > ∈ X1 ⇔ t is the total number of movies on at Hoyts. */
join
(select spectator, count(movie) as nbM
 from Opinions natural join IsOn
 where cinema = 'Hoyts' group by spectator) X2
/* < s, n > ∈ X2 ⇔ the spectator s has rated n movies among those
   which ar on at Hoyts. */
on (nbTot = nbM)
natural join Frequents

```

Query operational semantics

Query operational semantics

from 1. *cartesian product/join*

Query operational semantics

from 1. *cartesian product/join*
where 2. *selection on input relations*

Query operational semantics

from 1. *cartesian product/join*
where 2. *selection on input relations*
group by 3. *partition*

Query operational semantics

from 1. *cartesian product/join*
where 2. *selection on input relations*
group by 3. *partition*
having 4. *filtering the partition*

Query operational semantics

select 5. *projection xor aggregations*

select 5. *projection on partition criteria and aggregations*

from 1. *cartesian product/join*

where 2. *selection on input relations*

group by 3. *partition*

having 4. *filtering the partition*

Grouping: common mistake

For each cinema retrieve its name and city, the number of movies which are on and the spectators who frequent it.

```
select cinema, count (distinct movies) as nbM, city, spectator
*
```

```
from Cinema join IsOn on (name = cinema)
      natural join Frequents
```

```
group by cinema
```

ERROR at line 1:

ORA-00979: not a GROUP BY expression

Why?

Expected result

name	city	nbM	spectator
Hoyts CBD	Sydney	3	{Adrian, Jackie, Marie, Phil}
Hoyts	Brisbane	3	{Tom, Marie, Adrian}
Event Cinema	Cairns	5	{Alizée, Phil, Jackie}
Birch Carroll and Coyles	Brisbane	5	{Tom}

*The type of attribut spectator is a set.
Sets are not domains of atomic values!*

Possible result

name	city	nbM	spectator
Hoyts CBD	Sydney	3	Adrian
Hoyts CBD	Sydney	3	Jackie
Hoyts CBD	Sydney	3	Marie
Hoyts CBD	Sydney	3	Phil
Hoyts	Brisbane	3	Tom
Hoyts	Brisbane	3	Marie
Hoyts	Brisbane	3	Adrian
Event Cinema	Cairns	5	Alizée
Event Cinema	Cairns	5	Phil
Event Cinema	Cairns	5	Jackie
Birch Carroll and Coyles	Brisbane	5	Tom

How about in Mysql?

Mysql (Ver 14.14 Distrib 5.5.19, for osx10.6 (i386)) returns:

name	city	nbM	spectator
Hoyts CBD	Sydney	3	Adrian
Hoyts	Brisbane	3	Tom
Event Cinema	Cairns	5	Alizée
Birch Carroll and Coyles	Brisbane	5	Tom

which is incorrect....

To fix city problem

```
select cinema, count(movie) as nbM, city  
from Cinema join IsOn on (name=cinema)  
group by cinema, city
```

Both expressions

group by cinema and *group by cinema, city*

build the same partition because each cinema has only one city.

To fix spectator problem

```

select cinema, nbM, city, spectator
from (select cinema, count(movie) as nbM, city
      from Cinema join IsOn on (name=cinema)
      group by cinema, city) X
/* <c, n, a> ∈ X ⇔ the cinema c has n movies on and is located in city
a. */
natural join Frequent

```


Chapter 5 – SQL, a relational language (LMD part)

- 20 Introduction
- 21 Select Statement
- 22 Joining relations
- 23 Set Operators
- 24 Nested queries
- 25 Aggregation and Grouping
- 26 More about sub-queries**
- 27 Missing Values
- 28 SQL Types
- 29 Language limitations
- 30 As a conclusion

IN operator

- *A query result is a relation.*
- tuple IN relation \iff tuple \in relation
- Conversely for tuple NOT IN relation

IN operator

- *A query result is a relation.*
- tuple IN relation \iff tuple \in relation
- Conversely for tuple NOT IN relation

Find the name and city of cinema Phil frequents

which has no equivalent in algebra...

IN operator

- *A query result is a relation.*
- tuple IN relation \iff tuple \in relation
- Conversely for tuple NOT IN relation

Find the name and city of cinema Phil frequents

(select cinema from Frequents
where spectator = 'Phil')

which has no equivalent in algebra...

IN operator

- *A query result is a relation.*
- tuple IN relation \iff tuple \in relation
- Conversely for tuple NOT IN relation

Find the name and city of cinema Phil frequents

```
select name, city from Cinemas
where name in (select cinema from Frequents
              where spectator = 'Phil')
```

which has no equivalent in algebra...

IN operator

- *A query result is a relation.*
- tuple IN relation \iff tuple \in relation
- Conversely for tuple NOT IN relation

Find the name and city of cinema Phil frequents

```
select name, city from Cinemas
where name in (select cinema from Frequents
               where spectator = 'Phil')
```

which has no equivalent in algebra...

Another expression:

```
(Cinemas(name=cinema)*(Frequents:spectator='Phil'))[name, city]
```

```
select C.name, C.city
from Cinemas C join Frequents F on (C.name = F.cinema)
where F.spectator = 'Phil'
```

General form of IN operator

The In operator can be generalised to fit tuples of any length

- $\langle X_1, X_2, \dots, X_n \rangle \text{ IN relation (whose attributes are } A_1, A_2, \dots, A_n)$
is true $\iff \langle X_1, X_2, \dots, X_n \rangle \in \text{relation}.$

General form of IN operator

The In operator can be generalised to fit tuples of any length

- $\langle X_1, X_2, \dots, X_n \rangle \text{ IN relation (whose attributes are } A_1, A_2, \dots, A_n)$
is true $\iff \langle X_1, X_2, \dots, X_n \rangle \in \text{relation}.$

select ... from ...
where (X_1, X_2, \dots, X_n) in
(select A_1, A_2, \dots, A_n from)

*Of course, for each $i=1..n$,
types of X_i and A_i must be comparable*

Sub-queries: good practice

Give movies which have the same rating as 'Australia' given by Marie

select movies from Opinions

where stars = select stars from Opinions

where spectator = 'Marie' and movie = 'Australia')

The query is successfully executed when the sub-query returns one tuple only, otherwise an exception is raised.

Sub-queries: good practice

Give movies which have the same rating as 'Australia' given by Marie

```
select movies from Opinions
where stars = select stars from Opinions
               where spectator = 'Marie' and movie = 'Australia')
```

The query is successfully executed when the sub-query returns one tuple only, otherwise an exception is raised.

Better:

```
select movies from Opinions
where stars in select stars from Opinions
               where spectator = 'Marie' and movie = 'Australia')
```

Sub-queries: good practice

Give movies which have the same rating as 'Australia' given by Marie

```
select movies from Opinions
where stars = select stars from Opinions
               where spectator = 'Marie' and movie = 'Australia')
```

The query is successfully executed when the sub-query returns one tuple only, otherwise an exception is raised.

Better:

```
select movies from Opinions
where stars in select stars from Opinions
               where spectator = 'Marie' and movie = 'Australia')
```

Even better:

```
select movies from Opinions o1 join Opinions o2 using (stars)
where o1.stars = o2.stars and
      o2.spectator = 'Marie' and o2.movie = 'Australia')
```

Scope of variables

Scoping rule: named tuple variables can be referred in any inner subquery
For each cinema, return the movies which are on and have the less rating (in the cinema). Give the movie rating as well.

```
select cinema, movie, stars
from Opinions natural join IsOn i1 /* i1 is called tuple variable */
where stars in (select min (stars)
                from Opinions natural join IsOn i2
                where i1.cinema = i2.cinema)
/* The subquery is called correlated subquery */
```

A better variant:

```
select cinema, movie, stars
from Opinions O natural join IsOn I join
  (select cinema, min(stars) as minS
   from Opinions natural join IsOn) as X
on (O.stars = X.minS and I.cinema = X.cinema)
```

Chapter 5 – SQL, a relational language (LMD part)

- 20 Introduction
- 21 Select Statement
- 22 Joining relations
- 23 Set Operators
- 24 Nested queries
- 25 Aggregation and Grouping
- 26 More about sub-queries
- 27 Missing Values**
- 28 SQL Types
- 29 Language limitations
- 30 As a conclusion

Motivation

Sometimes we need missing values:

- Value temporarily unknown:
I don't know his address now, but I can find it soon
- Value temporarily forbidden:
She is not married, so she can't have a marital name
- Value definitively forbidden:
He didn't attend this exam, so he cannot get a mark for it.
- etc.

NULL value

- The missing value is denoted *NULL*.
- In outputs, NULL values are not displayed (nothing appears).
- NULL belongs to all types.
- All operators are revisited.
- Two new operators: IS NULL - IS NOT NULL

Operators with null values

Operators with null values

- Comparison operators: let A be an attribute.
 $\forall \text{ op} \in \{<, >, <=>, =\}$, $A \text{ op Null}$, $\text{Null op } A$ are Null

Operators with null values

- Comparison operators: let A be an attribute.
 $\forall \text{ op} \in \{<, >, <>, =\}$, $A \text{ op Null}$, $\text{Null op } A$ are Null
- Arithmetic operators:
 $\forall \text{ op} \in \{+, -, /, *, \dots\}$, $A \text{ op Null}$, $\text{Null op } A$ are Null

Operators with null values

- Comparison operators: let A be an attribute.
 $\forall \text{ op} \in \{<, >, <=>, =\}, A \text{ op Null}, \text{Null op } A \text{ are Null}$
- Arithmetic operators:
 $\forall \text{ op} \in \{+, -, /, *, \dots\}, A \text{ op Null}, \text{Null op } A \text{ are Null}$
- Logical operators:

AND	True	False	Null
True	True	False	Null
False	False	False	False
Null	Null	False	Null

Operators with null values

- Comparison operators: let A be an attribute.
 $\forall \text{ op} \in \{<, >, <=>, =\}$, A op Null, Null op A are Null
- Arithmetic operators:
 $\forall \text{ op} \in \{+, -, /, *, \dots\}$, A op Null, Null op A are Null
- Logical operators:

AND	True	False	Null
True	True	False	Null
False	False	False	False
Null	Null	False	Null

OR	TRUE	FALSE	NULL
True	True	True	True
False	True	False	Null
Null	True	Null	Null

Operators with null values

- Comparison operators: let A be an attribute.
 $\forall \text{ op} \in \{<, >, <=>, =\}$, A op Null, Null op A are Null
- Arithmetic operators:
 $\forall \text{ op} \in \{+, -, /, *, \dots\}$, A op Null, Null op A are Null
- Logical operators:

AND	True	False	Null
True	True	False	Null
False	False	False	False
Null	Null	False	Null

OR	TRUE	FALSE	NULL
True	True	True	True
False	True	False	Null
Null	True	Null	Null

Similarly NOT null is null

- min, max, sum, avg ignore null values.
If all values are null, then the result is null.

- min, max, sum, avg ignore null values.
If all values are null, then the result is null.
- count(*) doesn't care null values: it counts tuples.

- min, max, sum, avg ignore null values.
If all values are null, then the result is null.
- count(*) doesn't care null values: it counts tuples.
- count (distinct A) and count (A) count only non null values

- min, max, sum, avg ignore null values.
If all values are null, then the result is null.
- count(*) doesn't care null values: it counts tuples.
- count (distinct A) and count (A) count only non null values

*NULL is neither 0, nor { }, nor "", nor ()
NULL is nothing!*

Queries with null values

R	idStudent	Mark	Course
	12	45	math
	11	90	french
	11	75	math
	12	65	physics
	12		french

Find information about students who have got a mark different than 90

select * from R where mark <> 90

Find information about students who have got a mark different than 90

select * from R where mark <> 90

R	idStudent	Mark	Course
	12	45	math
	11	75	math
	12	65	physics

Find information about students who have got a mark different than 90

select * from R where mark \neq 90

R	idStudent	Mark	Course
	12	45	math
	11	75	math
	12	65	physics

*A tuple is selected if it satisfies the where condition
(if the condition evaluates to TRUE)*

Find information about students who have got a mark different than 90 (including those who didn't attend)

select * from R where mark <> 90 or mark is null

R	idStudent	Mark	Course
	12	45	math
	11	75	math
	12	65	physics
	12		french

For each student, compute its average mark. Unattended exam counts for 0

```
select idStudent, avg (nvl (mark, 0)) as avgMark
from R group by idStudent
```

/ nvl (x, y): if x is not null then x else y */*

R	idStudent	AvgMark
	12	36.67
	11	82.5

Generating null values

For each cinema, find the spectators who frequent it (including cinemas where nobody goes!)

```
select cinema, spectator from Frequents
union
select name, null from Cinemas
where name not in (select cinema from Frequents)
```

The expected result is:

spectator	cinema
Marie	Hoyts CBD
Adrian	Hoyts CBD
Phil	Hoyts CBD
Jackie	Hoyts CBD
Tom	Hoyts
Alizee	Event Cinema
Marie	Hoyts
Adrian	Hoyts

For each cinema, find the spectators who frequent it (including those where nobody goes!)

```
select F.spectator, C.name from Cinemas C left outer join
    Frequents F on (C.name = F.cinema)
```

For each row in Cinemas that does not satisfy the join condition with any row in Frequents, a joined row is returned with NULL values in columns of Frequents.

What is the result of?:

```
select F.spectator, F.cinema
from Cinemas C left outer join
    Frequents F on (C.name = F.cinema)
```

spectator	cinema
Marie	Hoyts CBD
Adrian	Hoyts CBD
Phil	Hoyts CBD
Jackie	Hoyts CBD
Tom	Hoyts
Alizee	Event Cinema

Another example :

Students (stuld, firstName, lastName)

Courses (courseld, name) Enrolls (stuld, courseld)

Another example :

Students (stuld, firstName, lastName)

Courses (courseld, name) Enrolls (stuld, courseld)

For each student (id, firstname and lastname) find course ids in which she/he is enrolled (retrieve also students who aren't enrolled in any course)

```
select stuld, firstName, lastName, courseld  
from Students natural left outer join Enrolls
```

Another example :

Students (stuld, firstName, lastName)

Courses (courseld, name) Enrolls (stuld, courseld)

For each student (id, firstname and lastname) find course ids in which she/he is enrolled (retrieve also students who aren't enrolled in any course)

```
select stuld, firstName, lastName, courseld
from Students natural left outer join Enrolls
```

For each row in *Students* that does not satisfy the join condition with any row in *Enrolls*, a joined row is returned with NULL values in columns of *Enrolls*.

Yet another example:

For each student (id, firstname and lastname) find the id and name of the courses in which she/he is enrolled (retrieve also students who aren't enrolled in any course)

Yet another example:

For each student (id, firstname and lastname) find the id and name of the courses in which she/he is enrolled (retrieve also students who aren't enrolled in any course)

from Students natural left outer join Enrolls
natural left outer join Courses

Yet another example:

For each student (id, firstname and lastname) find the id and name of the courses in which she/he is enrolled (retrieve also students who aren't enrolled in any course)

```
select stuld, firstName, lastName, courseId, name  
from Students natural left outer join Enrolls  
      natural left outer join Courses
```

Yet another example:

For each student (id, firstname and lastname) find the id and name of the courses in which she/he is enrolled (retrieve also students who aren't enrolled in any course)

```
select stuld, firstName, lastName, courseId, name
from Students natural left outer join Enrolls
      natural left outer join Courses
```

The following query returns the same result:

```
select stuld, firstName, lastName, courseId, name
from Courses natural join Enrolls natural right outer join Students
```


As a summary...

For each cinema, give its name, who frequent it, and among all movies on at it how many have been given 5 stars. Consider as well, cinemas with no regular spectators and cinemas which have no movies rated 5 stars.

As a summary...

For each cinema, give its name, who frequent it, and among all movies on at it how many have been given 5 stars. Consider as well, cinemas with no regular spectators and cinemas which have no movies rated 5 stars.

Expected result:

cinema	spectator	nb5stars	
c1	p1	12	among all movies on at c1, 12 have been p2 is another person who frequents c1.
c1	p2	12	
c2	p2	120	
c4	p3	0	no movies on at c4 received 5 stars.
c5	p1		p1 frequents c5, no movies on at c4.
c5	p10		p10 frequents c5 as well.
c6		11	c6 has no regular spectators. 11 movies h stars among all on at c6.
c7		0	c7 has no regular spectators, no movies c received 5 stars.

The query relies on a relation X (cinema, nb5stars):

$\langle c, nb \rangle \in X \iff$ Among all movies on at c , nb have received 5 stars.
 nb might be equal to 0.

/ X expression is: */*

```
select cinema, count(distinct movies) as nb
from IsOn natural left outer join Opinions
where stars=5
group by C.name
```

The query relies on a relation X (cinema, nb5stars):

$\langle c, nb \rangle \in X \iff$ Among all movies on at c , nb have received 5 stars.
 nb might be equal to 0.

/ X expression is: */*

```
select cinema, count(distinct movies) as nb
from IsOn natural left outer join Opinions
where stars=5
group by C.name
```

/ Thus the query refering X is: */*

```
select C.name, F.spectators, X.nb
from Cinemas C left outer join Frequents F on (C.name=F.cinema)
      left outer join X on (C.name=X.cinema)
```

/ Eventually, after having replaced X by its expression above, the final query is:*
**/*

```
select C.name, F.spectators, X.nb
from Cinemas C left outer join Frequents F on (C.name=F.cinema)
      left outer join (select cinema, count(distinct movies) as nb
                        from IsOn natural left outer join Opinions
                        where stars=5
```

Chapter 5 – SQL, a relational language (LMD part)

- 20 Introduction
- 21 Select Statement
- 22 Joining relations
- 23 Set Operators
- 24 Nested queries
- 25 Aggregation and Grouping
- 26 More about sub-queries
- 27 Missing Values
- 28 SQL Types**
- 29 Language limitations
- 30 As a conclusion

SQL Identifiers

- Objects: relations, attributes, views, etc. are identified by their names, with similar conventions to common programming language, BUT identifier names are case INSENSITIVE.
Examples: Spectators, name, select, From, wherE, And, OR, etc

SQL Identifiers

- Objects: relations, attributes, views, etc. are identified by their names, with similar conventions to common programming language, BUT identifier names are case INSENSITIVE.
Examples: Spectators, name, select, From, wherE, And, OR, etc
- Constants: litteral values for strings, numbers, etc. are implicitly typed.
Examples: 'This is a string', 'John\'s bike', 123.84, 4.23e-12, .01, 23., etc

- Tuple and set literals: list of literal values between brackets (and)
Exemples: (12, 3, 9, 10) is a set literal (could be a tuple literal!) (123, 'John', '12-3-2001') is a tuple literal

SQL Data Types

- Strings:

char [(n)]: uses n characters (fixed length), left-justified, blank-padded

varchar [(n)]: uses 0..n characters (variable length), no padding

SQL Data Types

- Strings:
 - char [(n)]: uses n characters (fixed length), left-justified, blank-padded
 - varchar [(n)]: uses 0..n characters (variable length), no padding
- Numbers
 - number (p, s): a fixed-point number with precision p and scale s
 - number (p): a fixed-point number with precision p
 - number: a floating-point number with decimal precision 38

- Date
an instant (in Oracle: unit second; RDBMS dependant)

- Date
an instant (in Oracle: unit second; RDBMS dependant)
- Long, Raw, Blob, etc.

Operators and functions (RDBMS dependant)

- Comparison operators (on all types):
<, >, <>, !=, =

Operators and functions (RDBMS dependant)

- Comparison operators (on all types):
<, >, <>, !=, =
- Boolean operators:
NOT, AND, OR (given in decreased order of precedence)

Operators and functions (RDBMS dependant)

- Comparison operators (on all types):
<, >, <>, !=, =
- Boolean operators:
NOT, AND, OR (given in decreased order of precedence)
- Arithmetic operators:
+ Addition, - Subtraction, * Multiplication, / Division

Operators and functions (RDBMS dependant)

- Comparison operators (on all types):
<, >, <>, !=, =
- Boolean operators:
NOT, AND, OR (given in decreased order of precedence)
- Arithmetic operators:
+ Addition, - Subtraction, * Multiplication, / Division
- Other numeric operators:
abs, ceil, floor, trunc, round, exp, pow, mod, sqrt, etc.
sin, cos, tan, atan, etc.

- String functions:
concat, lower, upper, substring, length, to_number, etc.

- String functions:
concat, lower, upper, substring, length, to_number, etc.
- Date functions:
 to_date (string,string) → a date
/ to_date ('05 Dec 2001', 'DD Mon YYYY') */*
 to_char (date, string) → a string
/ to_char (d,f) returns a string representing d according to the format f. */*
/ to_char (to_date ('05 Dec 2001', 'DD Mon YYYY'), 'DD/MM/YY')*
*= '05/12/01' */*

- Some other date functions:
add_month (date, integer) \longrightarrow a date
/ add_months (d1, n) returns the date d plus n months. */*
- Arithmetic operators extended to dates:
d1, d2: date, n: integer ≥ 0
d1 - d2 is the number of days between d1 and d2
d1 + n is the date n days in front of d1
d1 - n is the date n days behind d1
- Other functions: sysdate \longrightarrow a date, etc.

*For comprehensive description of functions
see the Oracle documentation!*

Expressions in the Select Statement

For each movie, retrieve its title and year of release, and compute since how long time (years) it has been released

```
select title, releaseYear  
      to_number (to_char (sysdate, 'YYYY'))  
    - to_number (to_char (releaseYear, 'YYYY')) as ReleasedSince  
from Movies
```

*Various functions and operators are available
For a comprehensive list, SEE ORACLE Documentation!*

Chapter 5 – SQL, a relational language (LMD part)

- 20 Introduction
- 21 Select Statement
- 22 Joining relations
- 23 Set Operators
- 24 Nested queries
- 25 Aggregation and Grouping
- 26 More about sub-queries
- 27 Missing Values
- 28 SQL Types
- 29 Language limitations**
- 30 As a conclusion

No iteration

Let us consider the relation `Persons (firstname, friendname)`.

- Assuming that all friends of my friends are also my friends
- How to answer *For each person give all of her(his) friend?*

Build the closure of Persons

Example:

The initial relation instance is:

firstname	friendname
Peter	Paul
Peter	John
Mary	Peter
Mary	David
David	Cath
Paul	Mary

The first step is: for each person p , find the friends of p 's friends (if any, only those I don't know yet)).

```
select distinct p1.firstname, p2.friendname  
from Persons p1 join Persons p2 on (p1.friendname = p2.firstname)
```


The first step is: for each person p, find the friends of p's friends (if any, only those I don't know yet)).

```
select distinct p1.firstname, p2.friendname
from Persons p1 join Persons p2 on (p1.friendname = p2.firstname)
where (p1.firstname, p2.friendname) not in
      (select firstname,friendname from Persons)
```

The union of the result with the original relation is:

firstname	friendname
Peter	Paul
Peter	John
Mary	Peter
Mary	David
David	Cath
Paul	Mary

(cont.)	
<i>Peter</i>	<i>Mary</i>
<i>Mary</i>	<i>Cath</i>
<i>Mary</i>	<i>John</i>
<i>Mary</i>	<i>Paul</i>
<i>Paul</i>	<i>David</i>
<i>Paul</i>	<i>Peter</i>

Run again the query.

The union of the result with the original relation is:

firstname	friendname
Peter	Paul
Peter	John
Peter	Mary
Mary	Peter
Mary	David
Mary	Cath

(cont.)	
Mary	John
Mary	Paul
Paul	Mary
Paul	David
Paul	Peter
David	Cath

(cont.)	
<i>Peter</i>	<i>Cath</i>
<i>Peter</i>	<i>David</i>
<i>Peter</i>	<i>Peter</i>
<i>Mary</i>	<i>Mary</i>
<i>Paul</i>	<i>Cath</i>
<i>Paul</i>	<i>John</i>
<i>Paul</i>	<i>Paul</i>

Run again the query: the result is empty.

No more tuples are generated. The closure has been built.

Chapter 5 – SQL, a relational language (LMD part)

- 20 Introduction
- 21 Select Statement
- 22 Joining relations
- 23 Set Operators
- 24 Nested queries
- 25 Aggregation and Grouping
- 26 More about sub-queries
- 27 Missing Values
- 28 SQL Types
- 29 Language limitations
- 30 As a conclusion**

Summary

- We studied SQL Data Manipulation Language only (Data Definition Language still to be seen)
- Many ways to express a given query: how to choose?
 - Not that simple...
 - As queries are optimised by the DBMS, this is not a matter the programmer has to worry about..
 - However, depending on the DBMS, query evaluation response time could be an issue.