

# CSCI235 Database Systems

## Database Design Quality

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Database Design Quality

## Outline

Why not ONE BIG TABLE !?

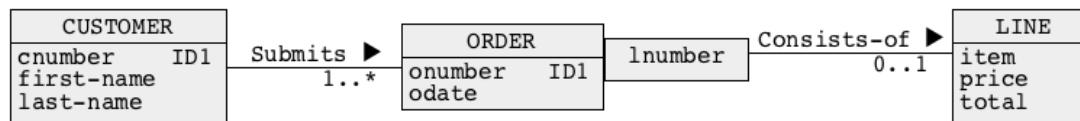
Where is a problem ?

Functional dependency

# Why not ONE BIG TABLE !?

Let us consider the following database domain:

- A **customer** is described by a unique **customer number**, **first**, and **last name**
- **Customers** submit **orders**. An **order** is described by a **unique order number** and **order date**
- Orders consist of **lines**. A **line** contains information about a **name of ordered item**, **price per single item**, and **total number of ordered items**



Logical design provides the following relational schemas:

CUSTOMER(cnumber, first-name, last-name)  
PRIMARY KEY = (cnumber)

Relational schemas

ORDERS(onumber, odate, cnumber) PRIMARY KEY = (onumber)  
FOREIGN KEY = (cnumber) REFERENCES CUSTOMER(cnumber)

LINE(onumber, lnumber, item, price total)  
PRIMARY KEY = (onumber, lnumber)  
FOREIGN KEY = (onumber) REFERENCES ORDERS(onumber)

# Why not ONE BIG TABLE !?

Why not one relational schema ?

Big relational schema

```
CUSTOMER(cnumber, first-name, last-name, onumber, odate, cnumber, onumber, lnumber, item, price total)
PRIMARY KEY = (cnumber, onumber, lnumber)
```

Insertion of information about one customer who submitted 2 orders such that each order consists several lines reveals a problem !

Big relational table

cnumber	fname	lname	onumber	odate	lnumber	item	price	total
7	James	Bond	7	2017-01-01	1	bolt	23.04	5
7	James	Bond	7	2017-01-01	2	screw	29.01	3
7	James	Bond	7	2017-01-01	3	nut	4.55	2
7	James	Bond	8	2018-01-01	1	bolt	23.04	1
7	James	Bond	8	2018-01-01	2	screw	23.04	1
7	James	Bond	8	2018-01-01	3	nut	23.04	2
7	James	Bond	8	2018-01-01	4	lock	23.04	1

# Why not ONE BIG TABLE !?

Big relational table									
cnumber	fname	lname	onumber	odate	lnumber	item	price	total	
7	James	Bond	7	2017-01-01	1	bolt	23.04	5	
7	James	Bond	7	2017-01-01	2	screw	29.01	3	
7	James	Bond	7	2017-01-01	3	nut	4.55	2	
7	James	Bond	8	2018-01-01	1	bolt	23.04	1	
7	James	Bond	8	2018-01-01	2	screw	23.04	1	
7	James	Bond	8	2018-01-01	3	nut	23.04	2	
7	James	Bond	8	2018-01-01	4	lock	23.04	1	

A **number**, **first name**, and **last name** of a customer is repeated as many times as the total number of different items purchased in all orders and

...

... and **order number** is repeated together with **order date** as many times as the total number of different items purchased in an order

# Why not ONE BIG TABLE !?

A multitable design does not have such a problem:

CUSTOMER(cnumber, first-name, last-name)  
**PRIMARY KEY** = (cnumber)

cnumber	fname	lname
7	James	Bond

CUSTOMER schema

CUSTOMER table

ORDERS(onenumber, odate, cnumber) **PRIMARY KEY** = (onenumber)  
**FOREIGN KEY** = (cnaumber) **REFERENCES** CUSTOMER(cnumber)

onenumber	odate	cnumber
7	2017-01-01	7
8	2018-01-01	7

ORDERS schema

ORDERS table

# Why not ONE BIG TABLE !?

A multitable design does not have such a problem:

```
LINE(onumber, lnumber, item, price total)
PRIMARY KEY = (onumber, lnumber)
FOREIGN KEY = (onumber) REFERENCES ORDERS(onumber)
```

LINE schema

onumber	lnumber	item	price	total
7	1	bolt	23.04	5
7	2	screw	29.01	3
7	3	nut	4.55	2
8	1	bolt	23.04	1
8	2	screw	23.04	1
8	3	nut	23.04	2
8	4	lock	23.04	1

LINE table

# Database Design Quality

## Outline

Why not ONE BIG TABLE !?

Where is a problem ?

Functional dependency

# Where is a problem ?

Why do we get redundancies in an incorrectly designed relational table ?

TABLE_NAME	COLUMN_1	COLUMN_2	...	COLUMN_N
	green	red	...	blue
	green	red	...	orange
	green	red	...	red
	blue	yellow	...	yellow
	blue	yellow	...	magenta
	orange	red	...	yellow
	orange	red	...	green

Data dependencies:

- If `COLUMN_1` is green then `COLUMN_2` is red
- If `COLUMN_1` is blue then `COLUMN_2` is yellow
- If `COLUMN_1` is orange then `COLUMN_2` is red

For any colour x if `COLUMN_1` is x then `COLUMN_2` is y

# Where is a problem ?

Data dependencies can be represented as a separate relational table ...

TABLE_1	
COLUMN_1	COLUMN_2

... and COLUMN\_2 can be removed from the original table

COLUMN_1	...	COLUMN_N

# Where is a problem ?

Do data dependencies exist in BIG TABLE ?

Big relational table									
cnumber	fname	lname	onumber	odate	lnumber	item	price	total	
7	James	Bond		2017-01-01	1	bolt	23.04	5	
7	James	Bond		2017-01-01	2	screw	29.01	3	
7	James	Bond		2017-01-01	3	nut	4.55	2	
7	James	Bond		2018-01-01	1	bolt	23.04	1	
7	James	Bond		2018-01-01	2	screw	23.04	1	
7	James	Bond		2018-01-01	3	nut	23.04	2	
7	James	Bond		2018-01-01	4	lock	23.04	1	

Data dependencies:

- If **cnumber** = 7 then **fname** = James
- If **cnumber** = 7 then **lname** = Bond

For any customer number x if **cnumber** = x then **fname** = y and **lname** = z

# Where is a problem ?

Do data dependencies exist in BIG TABLE ?

Big relational table									
cnumber	fname	lname	onumber	odate	lnumber	item	price	total	
7	James	Bond	7	2017-01-01	1	bolt	23.04	5	
7	James	Bond	7	2017-01-01	2	screw	29.01	3	
7	James	Bond	7	2017-01-01	3	nut	4.55	2	
7	James	Bond	8	2018-01-01	1	bolt	23.04	1	
7	James	Bond	8	2018-01-01	2	screw	23.04	1	
7	James	Bond	8	2018-01-01	3	nut	23.04	2	
7	James	Bond	8	2018-01-01	4	lock	23.04	1	

Data dependencies:

- If **onumber** = 7 then **odate** = 2017-01-01
- If **onumber** = 8 then **odate** = 2018-01-01

For any order number x if **onumber** = x then **odate** = y

# Database Design Quality

## Outline

Why not ONE BIG TABLE !?

Where is a problem ?

Functional dependency

# Functional dependency

What does it mean: if a value in column A is x then a value in column B is always y ?

It means that every value x in a column A is associated with only one value y in a column B

For example, every customer number in a column **cnumber** is associated with only one first name in a column **fname**, i.e. a customer has only one first name

For example, every customer number in a column **cnumber** is associated with only one last name in a column **lname** i.e. a customer has only one last name

For example, every **order number** in a column **onumber** is associated with only one **order date** in a column **odate** i.e. an order has only one date

# Functional dependency

Such data dependency does not hold for **item name** and **order number** because an **item name** in a column **item** can be associated with many **order numbers** in a column **onumber** and the opposite ...

... an **order number** in a column **onumber** can be associated with many **item names** in a column **item**

# Functional dependency

If every value in a column **A** is associated with only one value in a column **B** then it means that the columns **A** and **B** represent a function **f** that maps the values in a column **A** into the values in a column **B**

$$f : \text{domain}(A) \rightarrow \text{domain}(B)$$

If every value in a column **cnumber** is associated with only one value in a column **fname** then the columns **cnumber** and **fname** represent a function

$$f : \text{domain}(\text{cnumber}) \rightarrow \text{domain}(\text{fname})$$

If every value in a column **cnumber** is associated with only one value in a column **lname** then the columns **cnumber** and **lname** represent a function

$$f : \text{domain}(\text{cnumber}) \rightarrow \text{domain}(\text{lname})$$

# Functional dependency

If every value in a column **onumber** is associated with only one value in a column **odate** then the columns **onumber** and **odate** represent a function

$$f : \text{domain}(\text{onumber}) \rightarrow \text{domain}(\text{odate})$$

If the columns **A** and **B** in a relational table **R** represent a function

$$f : \text{domain}(A) \rightarrow \text{domain}(B)$$

then in the future it will be denoted by

$$A \rightarrow B$$

and we shall say that a **functional dependency**  $A \rightarrow B$  is valid in a relational table **R** or that **A** functionally determines **B**

# Functional dependency

Therefore, the following functional dependencies are valid in a big table

**CUSTOMER:**

$cnumber \rightarrow fname$

$cnumber \rightarrow lname$

$onumber \rightarrow odate$

$onumber \rightarrow cnumber$

$onumber \rightarrow fname, lname$

$onumber, lnumber \rightarrow item$

$onumber, lnumber \rightarrow price, total$

... and the others

# Functional dependency

Functional dependency is a special kind of so called data dependency which is a reflection of the real world consistency constraint.

Functional dependencies can be used to describe the semantics (meaning) of data

Functional dependencies can be used to determine whether a relational schema (header of relational table) is constructed in a correct way

Functional dependencies can be used to design a database, however such approach is used very rarely

# References

T. Connolly, C. Begg, *Database Systems, A Practical Approach to Design, Implementation, and Management*, Chapter 14.1 The Purpose of Normalization, Chapter 14.2 How Normalization Supports Database Design, 14.3 Data Redundancies and Update Anomalies, Pearson Education Ltd, 2015

# CSCI235 Database Systems

## Functional Dependencies

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Functional dependencies

## Outline

Functional dependency ? What is it ?

Functional dependencies versus classes of objects

Functional dependencies versus associations

Derivations of functional dependencies

Armstrong axioms

Other inference rules

Using inference rules

# Functional dependency? What is it?

Let  $R = (A_1, \dots, A_n)$  be a relational schema (a header of relational table) and let  $X, Y$  be the nonempty subsets of  $R$

We say that a functional dependency  $X \rightarrow Y$  is valid in a relational schema  $R$  if

for any contents of a relational table  $R$ , it is not possible that  $R$  has two rows that agree in the components for all attributes in a set  $X$  yet disagree on one or more component for the attributes in a set  $Y$

## Examples

- A warehouse is located at exactly one address:  $\text{warehouse} \rightarrow \text{address}$
- An address is related to exactly one warehouse:  $\text{address} \rightarrow \text{warehouse}$
- At a warehouse, the parts of the same sort have only one total quantity:  $\text{warehouse,part} \rightarrow \text{quantity}$
- A car has one owner:  $\text{registration} \rightarrow \text{driving license}$
- A student has one first name and one last name and one date of birth:  $\text{student-number} \rightarrow \text{first-name, last-name-date-of-birth}$

# Functional dependency? What is it?

## More examples

- An employee belongs to one department:  
 $\text{employee-number} \rightarrow \text{department-name}$
- A manager manages one department:  $\text{manager-number} \rightarrow \text{department-name}$
- An employee has one manager:  $\text{employee-number} \rightarrow \text{manager-number}$
- A student enrols a subject one time:  
 $\text{student-number}, \text{subject-code} \rightarrow \text{enrolment-date}$
- An employee is located in one building in one office:  
 $\text{employee-number} \rightarrow \text{building-number}, \text{office-number}$
- An office in a building hosts one employee:  
 $\text{building-number}, \text{office-number} \rightarrow \text{employee-number}$
- An office in a building at a campus hosts one employee:  
 $\text{campus-name}, \text{building-number}, \text{office-number} \rightarrow \text{employee-number}$
- A department has one manager:  $\text{department-name} \rightarrow \text{manager-number}$
- A department is located in one building:  $\text{department-name} \rightarrow \text{building-number}$
- A department has one manager and it is located in one building:  
 $\text{department-name} \rightarrow \text{manager-number}, \text{building-number}$

# Functional dependency? What is it?

How to discover the **functional dependencies** in a relational table ?

- Is it possible to discover the **functional dependencies** in a **relational schema** (a header of relational table)  $R(A, B, C, D, E)$  ?
- Of course it is impossible to do it because we do not know the **semantics** (**the meanings**) of the names:  $R, A, B, C, D, E$
- To discover the **functional dependencies** in a relational table we must use the **semantics** of a **relational table name** and the **names of attributes**
- For example consider a relational schema (a header of relational table)  $TRIP(\text{rego\#}, \text{licence\#}, \text{tdate})$  of a relational table that contains information about the **trips** made by the **drivers** ( $\text{licence\#}$ ) who used the **trucks** ( $\text{rego\#}$ ) on a given **day** ( $\text{tdate}$ )
- Can a truck be used only one time ? If yes then  $\text{rego\#} \rightarrow \text{tdate}$
- Can a driver make only one trip ? If yes then  $\text{licence\#} \rightarrow \text{tdate}$
- Can a driver use more than one truck ? If yes then  $\text{licence\#} \not\rightarrow \text{rego\#}$
- Can a truck be used by more than one driver ? If yes then  $\text{rego\#} \not\rightarrow \text{licence\#}$
- And so on ...

# Functional dependencies

## Outline

[Functional dependency ? What is it ?](#)

[Functional dependencies versus classes of objects](#)

[Functional dependencies versus associations](#)

[Derivations of functional dependencies](#)

[Armstrong axioms](#)

[Other inference rules](#)

[Using inference rules](#)

# Functional dependencies versus classes of objects

A class of objects **STUDENT**

STUDENT	
s#	ID1
fname	ID2
lname	ID2
dob	ID2
average	
language	[1...*]

validates (satisfies) the following functional dependencies:

$s\# \rightarrow fname$

$s\# \rightarrow lname$

$s\# \rightarrow dob$

$s\# \rightarrow average$

$fname, lname, dob \rightarrow s\#$

$fname, lname, dob \rightarrow average$

# Functional dependencies versus classes of objects

The functional dependencies:

$s\# \rightarrow fname$

$s\# \rightarrow lname$

$s\# \rightarrow dob$

$s\# \rightarrow average$

are equivalent to a functional dependency

$s\# \rightarrow fname, lname, dob, average$

The functional dependencies

$fname, lname, dob \rightarrow s\#$

$fname, lname, dob \rightarrow average$

are equivalent to a functional dependency

$fname, lname, dob \rightarrow s\#, average$

# Functional dependencies

## Outline

Functional dependency ? What is it ?

Functional dependencies versus classes of objects

Functional dependencies versus associations

Derivations of functional dependencies

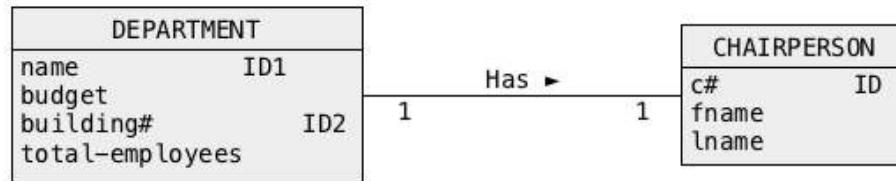
Armstrong axioms

Other inference rules

Using inference rules

# Functional dependencies versus associations

The classes of objects **DEPARTMENT** and **CHAIRPERSON** and association  
**Has**



validate (satisfy) the following functional dependencies:

$\text{name} \rightarrow \text{budget, building\#, total-employees}$

$\text{building\#} \rightarrow \text{name, budget, total-employees}$

$\text{c\#} \rightarrow \text{fname, lname}$

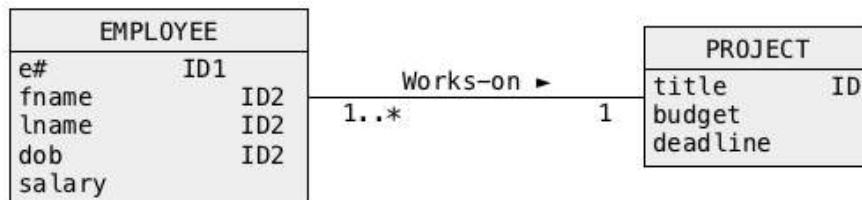
$\text{name} \rightarrow \text{c\#, fname, lname}$

$\text{building\#} \rightarrow \text{c\#, fname, lname}$

$\text{c\#} \rightarrow \text{name, building\#, budget, total-employees}$

# Functional dependencies versus associations

The classes of objects **EMPLOYEE** and **PROJECT** and association **Works-on**



validate (satisfy) the following functional dependencies:

$e\# \rightarrow fname, lname, dob, salary$

$fname, lname, dob \rightarrow e\#, salary$

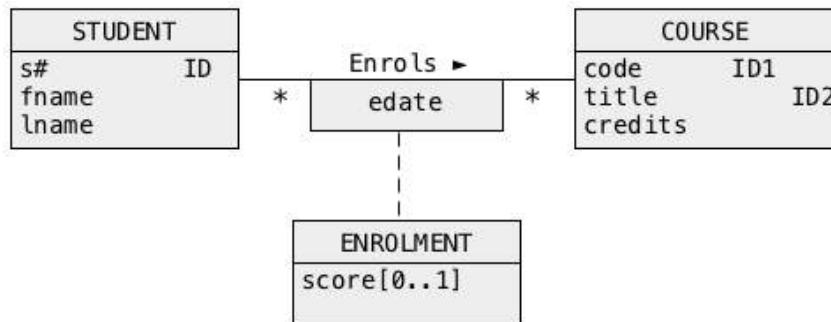
$title \rightarrow budget, deadline$

$e\# \rightarrow title, budget, deadline$

$fname, lname, dob \rightarrow title, budget, deadline$

# Functional dependencies versus associations

The classes of objects **STUDENT** and **COURSE** and association **Enrols**



validate (satisfy) the following functional dependencies:

$s\# \rightarrow fname, lname$

$code \rightarrow title, credits$

$title \rightarrow code, credits$

$s\#, code, edate \rightarrow score$

$s\#, title, edate \rightarrow score$

# Functional dependencies

## Outline

Functional dependency ? What is it ?

Functional dependencies versus classes of objects

Functional dependencies versus associations

Derivations of functional dependencies

Armstrong axioms

Other inference rules

Using inference rules

# Derivations of functional dependencies

Consider a relational schema (a header of relational table)

**EMPLOYEE (e#, ename, department, address, chairperson)**

If  $e\# \rightarrow ename$  and  $e\# \rightarrow department$  then  $e\# \rightarrow ename, department$

If  $e\# \rightarrow department$  and  $department \rightarrow address$  then  $e\# \rightarrow address$

If  $e\# \rightarrow department$  and  $department \rightarrow chairperson$  then  
 $e\# \rightarrow chairperson$

If  $e\# \rightarrow department$  then  $e\#, ename \rightarrow department$

If  $e\#, ename \rightarrow department$  then  $e\#, ename, address \rightarrow department$

It is always true that  $e\# \rightarrow e\#$

Functional dependency  $e\# \rightarrow e\#$  is called as a **trivial functional dependency**

It is always true that  $e\#, ename \rightarrow e\#$

A functional dependency  $e\#, ename \rightarrow e\#$  is also called as a **trivial functional dependency**

# Derivations of functional dependencies

A **trivial functional dependency** is a functional dependency that is always true no matter what its left and right hand sides are

For example,

$e\# \rightarrow e\#$ ,

$\text{department} \rightarrow \text{department}$

$e\#, \text{ename} \rightarrow e\#$ ,

$e\#, \text{ename}, \text{department} \rightarrow e\#, \text{department}$ ,

and so on

# Derivations of functional dependencies

Consider a relational schema  $R(A, B, C)$

It is always true that  $A \rightarrow A$

It is always true that  $A, B \rightarrow A$

It is always true that  $A, B, C \rightarrow A$

If  $A \rightarrow B$  then  $A, C \rightarrow B$

If  $A \rightarrow B, C$  then  $A \rightarrow B$  and  $A \rightarrow C$

If  $A \rightarrow B$  and  $B \rightarrow C$  then  $A \rightarrow C$

# Functional dependencies

## Outline

Functional dependency ? What is it ?

Functional dependencies versus classes of objects

Functional dependencies versus associations

Derivations of functional dependencies

Armstrong axioms

Other inference rules

Using inference rules

# Armstrong axioms

Let  $R = (A_1, \dots, A_n)$  be a relational schema (a header of relational table)

and

let  $X, Y, Z$  be the nonempty subsets of  $\{A_1, \dots, A_n\}$

- (i) If  $Y \subseteq X$  then  $X \rightarrow Y$  (reflexivity axiom)
- (ii) If  $X \rightarrow Y$  then  $X, Z \rightarrow Y, Z$  (augmentation axiom)
- (iii) If  $X \rightarrow Y$  and  $Y \rightarrow Z$  then  $X \rightarrow Z$  (transitivity axiom)

The axioms (i),(ii), and (iii) form a minimal and complete set of axioms

# Functional dependencies

## Outline

Functional dependency ? What is it ?

Functional dependencies versus classes of objects

Functional dependencies versus associations

Derivations of functional dependencies

Armstrong axioms

Other inference rules

Using inference rules

# Other inference rules

Let  $R = (A_1, \dots, A_n)$  be a relational schema (a header of relational table)

and

let  $X, Y, Z$  be the nonempty subsets of  $\{A_1, \dots, A_n\}$

If  $X \rightarrow Y$  and  $X \rightarrow Z$  then  $X \rightarrow Y, Z$  (union rule)

If  $X \rightarrow Y$  and  $W, Y \rightarrow Z$  then  $W, X \rightarrow Z$  (pseudotransitivity rule)

If  $X \rightarrow Y$  and  $Z \subseteq Y$  then  $X \rightarrow Z$  (decomposition rule or reduce right hand side rule)

If  $X \rightarrow Y$  then  $X, Z \rightarrow Y$  (extend left hand side rule)

# Functional dependencies

## Outline

Functional dependency ? What is it ?

Functional dependencies versus classes of objects

Functional dependencies versus associations

Derivations of functional dependencies

Armstrong axioms

Other inference rules

Using inference rules

# Using inference rules

Let  $R = (A, B, C)$  be a relational schema

Given set of functional dependencies  $F = \{A \rightarrow B, B \rightarrow C\}$  valid in  $R$

Is it true that  $A \rightarrow C$  ?

If  $A \rightarrow B$  and  $B \rightarrow C$  then application of **transitivity axiom** provides  $A \rightarrow C$

# Using inference rules

Let  $R = (A, B, C)$  be a relational schema

Given set of functional dependencies  $F = \{A \rightarrow B, C\}$  valid in  $R$

Is it true that  $A \rightarrow B$  and  $A \rightarrow C$ ?

Reflexivity axiom provides  $B, C \rightarrow C$

If  $A \rightarrow B, C$  and  $B, C \rightarrow C$  then transitivity axiom provides  $A \rightarrow C$

Reflexivity axiom provides  $B, C \rightarrow B$

If  $A \rightarrow B, C$  and  $B, C \rightarrow B$  then transitivity axiom provides  $A \rightarrow B$

# Using inference rules

Let  $R = (A, B, C)$  be a relational schema

Given set of functional dependencies  $F = \{A \rightarrow B, A \rightarrow C\}$  valid in  $R$

Is it true that  $A \rightarrow B, C$  ?

If  $A \rightarrow B$  then **augmentation axiom** provides  $A \rightarrow A, B$

If  $A \rightarrow C$  then **augmentation axiom** provides  $A, B \rightarrow B, C$

If  $A \rightarrow A, B$  and  $A, B \rightarrow B, C$  then **transitivity axiom** provides  $A \rightarrow B, C$

# Using inference rules

Let  $R = (A, B, C)$  be a relational schema

Given set of functional dependencies  $F = \{A \rightarrow B\}$  valid in  $R$

Is it true that  $A, C \rightarrow B$  ?

Reflexivity axiom provides  $A, C \rightarrow A$

If  $A, C \rightarrow A$  and  $A \rightarrow B$  then transitivity axiom provides  $A, C \rightarrow B$

# Using inference rules

A relational schema **STUDENT(s#, fname, lname, dob, average)** validates (satisfies) the following functional dependencies:

$s\# \rightarrow fname$

$s\# \rightarrow lname$

$s\# \rightarrow dob$

$s\# \rightarrow average$

$fname, lname, dob \rightarrow s\#$

$fname, lname, dob \rightarrow average$

We proved that if  $A \rightarrow B$  and  $A \rightarrow C$  then  $A \rightarrow B, C$

Hence,

$s\# \rightarrow fname, lname, dob, average$  and ...

$fname, lname, dob \rightarrow s\#, average$

Note, that both functional dependencies **cover** entire relational schema and **no other** functional dependencies that **do not cover** entire relational schema validate in the schema e.g.  $fname \rightarrow s\#$

# References

T. Connolly, C. Begg, *Database Systems, A Practical Approach to Design, Implementation, and Management*, Chapter 14.4 Functional Dependencies, Chapter 15.1 More on Functional Dependencies, Pearson Education Ltd, 2015

# CSCI235 Database Systems

## Database Normalization

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Database normalization

## Outline

### First Normal Form (1NF)

#### Keys

#### Functional dependencies and keys

#### Attributes

#### Full and partial functional dependencies

### Second Normal Form (2NF)

#### Transitive functional dependencies

### Third Normal Form (3NF)

### Boyce-Codd Normal Form (BCNF)

### Normalization of relational schemas

# First Normal Form (1NF)

A relational schema is in the **First Normal Form (1NF)** if all occurrences of rows in the respective relational table contain the same number of fields and include the atomic values only, i.e. there are no repeating fields and groups

1NF relational table									
cnumber	fname	lname	onumber	odate	lnumber	item	price	total	
7	James	Bond	7	2017-01-01	1	bolt	23.04	5	
7	James	Bond	7	2017-01-01	2	nut	29.01	3	
7	James	Bond	8	2017-01-02	1	nut	4.55	2	
7	James	Bond	8	2017-01-02	2	pin	14.25	2	

A relational schema which is not in the **First Normal Form (1NF)** is in **Zeroth Normal Form (0NF)** or it is called as a **nested relational schema**

A relational table built over a **nested relational schema** is called as a **nested relational table** or **0NF relational table**.

# First Normal Form (1NF)

1NF relational table

customers			orders			parts		
customer_id	customer_name	customer_address	order_id	order_date	order_qty	part_id	part_name	part_price
1	John Doe	123 Main St	101	2017-01-01	5	1	bolt	23.04
						2	nut	29.01
2	Jane Smith	456 Elm St	102	2017-01-02	3	1	nut	4.55
						2	pin	14.25

# Database normalization

## Outline

### First Normal Form (1NF)

#### Keys

#### Functional dependencies and keys

#### Attributes

#### Full and partial functional dependencies

### Second Normal Form (2NF)

#### Transitive functional dependencies

### Third Normal Form (3NF)

### Boyce-Codd Normal Form (BCNF)

### Normalization of relational schemas

# Keys

A **superkey** is a nonempty subset  $X$  of relational schema  $R = (A_1, \dots, A_n)$  such that for any two rows  $t_1, t_2$  in a relational table created over a relational schema  $R$   $t_1[X] \neq t_2[X]$

If  $X$  is a **superkey** in  $R$  then  $X \rightarrow A_1, \dots, A_n$

A **minimal key** is a **superkey**  $K$  with an additional property such that removal of any attribute from  $K$  causes  $K$  not to be a **superkey**

For example, a relational schema `TRIP(rego#, licence#, tdate)` has one **minimal key** (`rego#, licence#, tdate`)

For example, a relational schema `DRIVER(licence#, employee#, first-name, last-name)` has two **minimal keys** (`licence#`) and (`employee#`)

# Keys

For example, relational schema `DRIVER(licence#, employee#, first-name, last-name)` has many **superkeys**: `(licence#)`, `(employee#)`, `(licence#, employee#)`, `(licence#, first-name)`, `(licence#, last-name)`, `(licence#, first-name, last-name)`, and so on ...

A **primary key** is an arbitrarily selected **minimal key**

A **candidate key** is any other **minimal key** which is not a **primary key**

For example, a relational schema

`TRIP(rego#, licence#, tdate)`

has a **primary key** `(rego#, licence#, tdate)`

For example, a relational schema

`DRIVER(licence#, employee#, first-name, last-name)`

has a **primary key** `(licence#)` and a **candidate key** `(employee#)` or the opposite

# Database normalization

## Outline

[First Normal Form \(1NF\)](#)

[Keys](#)

[Functional dependencies and keys](#)

[Attributes](#)

[Full and partial functional dependencies](#)

[Second Normal Form \(2NF\)](#)

[Transitive functional dependencies](#)

[Third Normal Form \(3NF\)](#)

[Boyce-Codd Normal Form \(BCNF\)](#)

[Normalization of relational schemas](#)

# Functional dependencies and keys

Let  $R = (A_1, \dots, A_n)$  be a relational schema and let  $X, Y$  be nonempty subsets of  $R$  such that  $X \cup Y = R$

If a functional dependency  $X \rightarrow Y$  is valid in  $R$  then  $X$  is a **superkey**

If  $X$  is a **superkey** then a functional dependency  $X \rightarrow Y$  is valid in  $R$

For example, if a functional dependency `licence#, employee# → first-name, last-name` is valid in a relational schema

`DRIVER(licence#, employee#, first-name, last-name)`  
then `(licence#, employee#)` is a **superkey**

For example, if `(licence#)` is a superkey in a relational schema

`DRIVER(licence#, employee#, first-name, last-name)`  
then a functional dependency `licence# → employee#, first-name, last-name` is valid in `DRIVER`

# Database normalization

## Outline

[First Normal Form \(1NF\)](#)

[Keys](#)

[Functional dependencies and keys](#)

[Attributes](#)

[Full and partial functional dependencies](#)

[Second Normal Form \(2NF\)](#)

[Transitive functional dependencies](#)

[Third Normal Form \(3NF\)](#)

[Boyce-Codd Normal Form \(BCNF\)](#)

[Normalization of relational schemas](#)

# Attributes

A **prime attribute** is an attribute from relational schema **R** which is a member of at least one minimal key in **R**

A **nonprime attribute** is an attribute which is not **prime**

For example, the attributes **licence#** and **employee#** are **prime attributes** in a relational schema

**DRIVER(licence#, employee#, first-name, last-name)**

For example, the attributes **first-name** and **last-name** are **nonprime attributes** in a relational schema

**DRIVER(licence#, employee#, first-name, last-name)**

# Database normalization

## Outline

[First Normal Form \(1NF\)](#)

[Keys](#)

[Functional dependencies and keys](#)

[Attributes](#)

[Full and partial functional dependencies](#)

[Second Normal Form \(2NF\)](#)

[Transitive functional dependencies](#)

[Third Normal Form \(3NF\)](#)

[Boyce-Codd Normal Form \(BCNF\)](#)

[Normalization of relational schemas](#)

# Full and partial functional dependencies

A **full functional dependency** is a functional dependency  $X \rightarrow Y$  such that removal of any attribute  $A$  from  $X$  causes that  $(X-A) \not\rightarrow Y$

A **partial functional dependency** is a functional dependency which is not full functional dependency

For example, a functional dependency

`student#, subject#, edate → grade`

valid in a relational schema

`ENROLMENT (student#, subject#, edate, grade)` is a **full functional dependency** because none of the attributes `student#`, `subject#`, `edate` can be removed from the left hand side of the functional dependency such that it is still valid

For example, a functional dependency

`licence#, employee# → first-name, last-name` valid in a relational

schema `DRIVER (licence#, employee#, first-name, last-name)`

is a **partial functional dependency** because if either `licence#` or `employee#` attributes are removed from the left hand side of the functional dependency then it is still valid

# Database normalization

## Outline

[First Normal Form \(1NF\)](#)

[Keys](#)

[Functional dependencies and keys](#)

[Attributes](#)

[Full and partial functional dependencies](#)

[Second Normal Form \(2NF\)](#)

[Transitive functional dependencies](#)

[Third Normal Form \(3NF\)](#)

[Boyce-Codd Normal Form \(BCNF\)](#)

[Normalization of relational schemas](#)

## Second Normal Form (2NF)

A relational schema  $R$  is in the **Second Normal Form (2NF)** if every nonprime attribute  $A$  in  $R$  is fully functionally dependent on all minimal keys of schema  $R$

A relational table based on a relational schema

`INVENTORY(part, quantity, warehouse, warehouse address)`  
contains information about parts stored in warehouses, quantities of parts, and addresses of warehouse

The following functional dependencies are valid in a relational schema

`INVENTORY(part, quantity, warehouse, warehouse address)`

$\text{warehouse} \rightarrow \text{warehouse-address}$

$\text{part, warehouse} \rightarrow \text{quantity}$

If  $\text{warehouse} \rightarrow \text{warehouse-address}$  then

$\text{part, warehouse} \rightarrow \text{warehouse-address}$

If  $\text{part, warehouse} \rightarrow \text{warehouse-address}$  and  $\text{part, warehouse} \rightarrow \text{quantity}$  then  $\text{part, warehouse} \rightarrow \text{warehouse-address, quantity}$

## Second Normal Form (2NF)

If  $\text{part}, \text{warehouse} \rightarrow \text{warehouse-address}, \text{quantity}$  then a minimal key is  
 $(\text{part}, \text{warehouse})$

A relational schema

`INVENTORY(part, quantity, warehouse, warehouse address)`  
is not in **2NF** because nonprime attribute `warehouse-address` depends  
on a part (`warehouse`) of a key ( $\text{part}, \text{warehouse}$ )

A functional dependency that **violates 2NF** is  $\text{warehouse} \rightarrow \text{warehouse- address}$

If all **minimal keys** in a relational schema consist of only one attribute  
(single attribute keys) then such schema is always in **2NF**

This is because any **nonprime attribute** in the schema does not depend  
on a part of a key because each key consists of one attribute only

## Second Normal Form (2NF)

A relational schema

`INVENTORY(part, quantity, warehouse, warehouse address)`

must be decomposed into the relational schemas

`INVENTORY(part, quantity, warehouse)`

`WAREHOUSE(warehouse, warehouse-address)`

The following functional dependencies are valid in a relational schema

`INVENTORY(part, quantity, warehouse)`

`part, warehouse → quantity`

Hence `(part, warehouse)` is a minimal key

A relational schema `INVENTORY(part, quantity, warehouse)`

is in **2NF** because a nonprime attribute `quantity` does not depend on a part of key `(part, warehouse)`

## Second Normal Form (2NF)

The following functional dependencies are valid in a relational schema

**WAREHOUSE(warehouse, warehouse-address)**

$\text{warehouse} \rightarrow \text{warehouse-address}$

Hence **(warehouse)** is a minimal key

A relational schema **WAREHOUSE(warehouse, warehouse-address)** is in **2NF** because a nonprime attribute **warehouse-address** does not depend on a part of key **(warehouse)**

# Database normalization

## Outline

[First Normal Form \(1NF\)](#)

[Keys](#)

[Functional dependencies and keys](#)

[Attributes](#)

[Full and partial functional dependencies](#)

[Second Normal Form \(2NF\)](#)

[Transitive functional dependencies](#)

[Third Normal Form \(3NF\)](#)

[Boyce-Codd Normal Form \(BCNF\)](#)

[Normalization of relational schemas](#)

# Transitive functional dependencies

A functional dependency  $X \rightarrow Y$  valid in a relational schema  $R$  is a **transitive functional dependency** if there exists a nonempty subset  $Z$  of  $R$ , that is not a subset of any key in  $R$  and such that the functional dependencies  $X \rightarrow Z$  and  $Z \rightarrow Y$  are valid in  $R$

For example, if the functional dependencies  
 $\text{employee\#} \rightarrow \text{project-title}$  and  $\text{project-title} \rightarrow \text{department-name}$   
are valid in a relational schema

`DEPARTMENT(department-name, project-title, employee#)`  
then a functional dependency  $\text{employee\#} \rightarrow \text{department-name}$  is a **transitive functional dependency**

For example, if the functional dependencies  
 $\text{licence\#} \rightarrow \text{employee\#}$  and  $\text{employee\#} \rightarrow \text{first-name}$   
are valid in a relational schema

`DRIVER(licence#, employee#, first-name, last-name)`  
then a functional dependency  $\text{licence\#} \rightarrow \text{first-name}$  is **not a transitive functional dependency**

It is because (`employee#`) is a key in a relational schema `DRIVER`

[TOP](#)

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

20/43

# Transitive functional dependencies

We say that  $Y$  is transitively dependent on  $X$  in schema  $R$  if  $X \rightarrow Y$  is valid in  $R$  and  $X \rightarrow Y$  is a transitive functional dependency

For example, if the functional dependencies

$\text{trip\#} \rightarrow \text{licence\#}$ ,  $\text{licence\#} \rightarrow \text{employee\#}$ , and  $\text{employee\#} \rightarrow \text{licence\#}$  are valid in a relational schema  $\text{TRIP}(\text{trip\#}, \text{licence\#}, \text{employee\#})$  then an attribute  $\text{employee\#}$  is transitively dependent on an attribute  $\text{trip\#}$  and ...

... an attribute  $\text{licence\#}$  is transitively dependent on an attribute  $\text{trip\#}$

Then, information about a  $\text{licence\#}$  of a driver with a given  $\text{employee\#}$  is listed as many times as the total number of trips performed by the driver

For example, if a driver performed 100 trips then his/her  $\text{licence\#}$  is listed together with his/her  $\text{employee\#}$  100 times.

# Transitive functional dependencies

For example, if the functional dependencies  
 $\text{textbook} \rightarrow \text{subject}$  and  $\text{subject} \rightarrow \text{lecturer}$  are valid in a relational schema `OUTLINE(lecturer, subject, textbook)`  
then an attribute `lecturer` is **transitively dependent** on an attribute `textbook`

Then, information about a `lecturer` assigned to a `subject` is repeated as many times as many `textbooks` are listed for the `subject`

For example, if a `subject` has 2 `textbooks` then information about a `lecturer` assigned to a `subject` is listed twice

# Database normalization

## Outline

[First Normal Form \(1NF\)](#)

[Keys](#)

[Functional dependencies and keys](#)

[Attributes](#)

[Full and partial functional dependencies](#)

[Second Normal Form \(2NF\)](#)

[Transitive functional dependencies](#)

[Third Normal Form \(3NF\)](#)

[Boyce-Codd Normal Form \(BCNF\)](#)

[Normalization of relational schemas](#)

## Third Normal Form (3NF)

A relational schema  $R$  is in the **Third Normal Form (3NF)** if it is in **2NF** and no **nonprime attribute** of  $R$  is **transitively dependent** on the primary key

A relational table based on a relational schema

`SUPPLIER(s#, sname, company-name, city)`

contains information about suppliers working for a company located in a given city

The following functional dependencies are valid in a relational schema

`SUPPLIER(s#, sname, company-name, city)`

$s\# \rightarrow sname$

$s\# \rightarrow company-name$

$s\# \rightarrow city$

$company-name \rightarrow city$

A primary key in a relational schema `SUPPLIER` is (`s#`) because

$s\# \rightarrow sname, company-name, city$

## Third Normal Form (3NF)

A relational schema **SUPPLIER** is **not** in the **Third Normal Form (3NF)** because an attribute **city** is transitively dependent on a primary key (**s#**)

An attribute **city** is transitively dependent on a primary key (**s#**) because

$s\# \rightarrow \text{company-name}$  and  $\text{company-name} \rightarrow \text{city}$

A relational schema **SUPPLIER** is in the **Second Normal Form (2NF)** because each nonprime attribute **sname**, **company**, and **city** is fully functionally dependent on a primary key (**s#**)

$s\# \rightarrow \text{sname}$

$s\# \rightarrow \text{company-name}$

$s\# \rightarrow \text{city}$

## Third Normal Form (3NF)

A relational schema `SUPPLIER(s#, sname, company-name, city)` should be decomposed into the relational schemas

`SUPPLIER(s#, sname, company-name)`

`COMPANY(company-name, city)`

A relational schema `SUPPLIER(s#, sname, company-name)` is in **3NF** because no attribute is transitively dependent on a primary key (`s#`)

$s\# \rightarrow sname$

$s\# \rightarrow company-name$

A relational schema `COMPANY(company-name, city)`

is in **3NF** because no nonprime attribute is transitively dependent on a primary key (`company-name`)

$company-name \rightarrow city$

## Third Normal Form (3NF)

Any relational schema that consists of at most 2 attributes is always in 3NF

Let  $R(a,b)$  be a relational schema such that no notrivial functional dependencies are valid in  $R$

Then  $(a, b)$  is a primary key in  $R$  and ...

... no nonprime attribute is transitively dependent on a primary key  $(a, b)$

Let  $R(a,b)$  be a relational schema such that a functional dependency  $a \rightarrow b$  is valid in  $R$

Then  $(a)$  is a primary key in  $R$  and ...

... no nonprime attribute is transitively dependent on a primary key  $(a)$

## Third Normal Form (3NF)

Let  $R(a,b)$  be a relational schema such that the functional dependencies  $a \rightarrow b$  and  $b \rightarrow a$  are valid in  $R$

Then either ( $a$ ) is a primary key in  $R$  or ( $b$ ) is a primary key in  $R$  and ...

... no nonprime attribute is transitively dependent on either ( $a$ ) or ( $b$ )

# Third Normal Form (3NF)

Alternative definition of the **Third Normal Form**

A relational schema  $R$  is in the **Third Normal Form (3NF)** if whenever a functional dependency  $X \rightarrow A$  is valid in  $R$  then either ...

- $X$  is a superkey in  $R$  or
- $A$  is a prime attribute in  $R$

For example, a relational schema

`SUPPLIER(s#, sname, company-name, city)`

$s\# \rightarrow sname$

$s\# \rightarrow company-name$

$s\# \rightarrow city$

$company-name \rightarrow city$

is **not** in **3NF** because ...

... if we consider a functional dependency  $company-name \rightarrow city$  then ...

- an attribute `company-name` is not a superkey and
- an attribute `city` is not a prime attribute

## Third Normal Form (3NF)

For example, if the functional dependencies

$\text{city}, \text{street} \rightarrow \text{zipcode}$

$\text{zipcode} \rightarrow \text{city}$

are valid in a relational schema  $\text{LOCATION}(\text{city}, \text{street}, \text{zipcode})$   
then the schema has two minimal keys

$(\text{city}, \text{street})$

Directly implied by a functional dependency  $\text{city}, \text{street} \rightarrow \text{zipcode}$

$(\text{zipcode}, \text{street})$

If  $\text{zipcode} \rightarrow \text{city}$  then  $\text{zipcode}, \text{street} \rightarrow \text{city}, \text{street}$

A relational schema  $\text{LOCATION}(\text{city}, \text{street}, \text{zipcode})$  is in **3NF**  
because

- the left hand side of  $\text{city}, \text{street} \rightarrow \text{zipcode}$  is a superkey and
- the left side of  $\text{zipcode} \rightarrow \text{city}$  is not a superkey but ... the right hand side ( $\text{city}$ )  
of  $\text{zipcode} \rightarrow \text{city}$  is a prime attribute

# Third Normal Form (3NF)

The following relational table created over a relational schema  
**LOCATION(city, street, zipcode)** is redundant

city	street	zipcode
NY	55	484
NY	56	484
LA	55	473
LA	56	473
LA	57	474

because the repetitions of | LA ... 473 | and | NY ... 484 | are forced by a functional dependency  $\text{zip-code} \rightarrow \text{city}$

It means that **3NF** is not the highest normal form required !

# Database normalization

## Outline

[First Normal Form \(1NF\)](#)

[Keys](#)

[Functional dependencies and keys](#)

[Attributes](#)

[Full and partial functional dependencies](#)

[Second Normal Form \(2NF\)](#)

[Transitive functional dependencies](#)

[Third Normal Form \(3NF\)](#)

[Boyce-Codd Normal Form \(BCNF\)](#)

[Normalization of relational schemas](#)

# Boyce-Codd Normal Form (BCNF)

A relational schema  $R$  is in the **Boyce-Codd Normal Form (BCNF)** if whenever a functional dependency  $X \rightarrow A$  is valid in  $R$  then

- $X$  is a superkey in  $R$

**Boyce-Codd Normal Form** is more restrictive because its definition does not give the "second chance"

- $A$  is a prime attribute in  $R$

For example, if the functional dependencies

$\text{city}, \text{street} \rightarrow \text{zipcode}$

$\text{zipcode} \rightarrow \text{city}$

are valid in a relational schema  $\text{LOCATION}(\text{city}, \text{street}, \text{zipcode})$

then the schema has two minimal keys

$(\text{city}, \text{street})$  and  $(\text{zipcode}, \text{street})$

Then, a relational schema  $\text{LOCATION}$  is not in **BCNF** because

- the left side of  $\text{zipcode} \rightarrow \text{city}$  is not a superkey

# Boyce-Codd Normal Form (BCNF)

A relational schema  $\text{LOCATION}(\text{city}, \text{street}, \text{zipcode})$  should be decomposed into the relational schemas

$\text{SZ}(\text{street}, \text{zipcode})$

$\text{CZ}(\text{city}, \text{zipcode})$

A relational schema  $\text{SZ}(\text{street}, \text{zipcode})$  has no valid nontrivial functional dependencies

A minimal key in a relational schema  $\text{SZ}(\text{street}, \text{zipcode})$  is  $(\text{street}, \text{zipcode})$

A relational schema  $\text{SZ}(\text{street}, \text{zipcode})$  is in **BCNF** because does not exist a functional dependency whose left hand side is not a superkey

# Boyce-Codd Normal Form (BCNF)

A functional dependency  $\text{zipcode} \rightarrow \text{city}$  is valid in a relational schema  $\text{CZ}(\text{city}, \text{ zipcode})$

A minimal key in a relational schema  $\text{CZ}(\text{city}, \text{ zipcode})$  is  $(\text{zipcode})$

A relational schema  $\text{CZ}(\text{city}, \text{ zipcode})$  is in BCNF because the left hand of functional dependency  $\text{zipcode} \rightarrow \text{city}$  is a superkey ( $\text{zipcode}$ )

Every relational schema, which consists of at most 2 attributes is always in BCNF

Normalization to BCNF "costs" a functional dependency  $\text{city}, \text{ street} \rightarrow \text{zipcode}$

It means that it is impossible to enforce the functional dependency  $\text{city}, \text{ street} \rightarrow \text{zipcode}$

with a primary key or candidate key constraints of `CREATE TABLE` statement

# Boyce-Codd Normal Form (BCNF)

The following relational tables are created through decomposition of a relational schema **LOCATION(city, street, zipcode)** into the relational schemas **SZ(street, zipcode)** and **CZ(city, zipcode)**

LOCATION			SZ		CZ		Decomposition into BCNF tables	
city	street	zipcode	street	zipcode	city	zipcode		
NY	55	484						
NY	56	484	55	484	NY	484		
LA	55	473	56	484	LA	473		
LA	56	473	55	473	LA	474		
LA	57	474	56	473				
			57	474				

A functional dependency  $\text{city}, \text{street} \rightarrow \text{zipcode}$  cannot be enforced in the relational tables **SZ** and **CZ**

# Database normalization

## Outline

[First Normal Form \(1NF\)](#)

[Keys](#)

[Functional dependencies and keys](#)

[Attributes](#)

[Full and partial functional dependencies](#)

[Second Normal Form \(2NF\)](#)

[Transitive functional dependencies](#)

[Third Normal Form \(3NF\)](#)

[Boyce-Codd Normal Form \(BCNF\)](#)

[Normalization of relational schemas](#)

# Normalization of relational schemas

Let  $R = (A_1, \dots, A_n)$  be a relational schema (a header of relational table)

Normalization of a relational schema  $R$  is performed in the following way

- Identify all functional dependencies valid in a relational schema  $R$
- Use the functional dependencies to derive all minimal keys
- Use the functional dependencies and minimal keys to identify the highest normal form satisfied by a relational schema  $R$
- Decompose a relational schema  $R$  into the relational schemas in **BCNF (3NF)**

# Normalization of relational schemas

For example, consider a relational schema

**SHIPMENT(s#, city, status, p#, quantity)**

The following functional dependencies are valid in the schema

$s\# \rightarrow city$

$s\# \rightarrow status$

$city \rightarrow status$

$s\#, p\# \rightarrow quantity$

$s\#, p\# \rightarrow city$

$s\#, p\# \rightarrow status$

A minimal key is (**s#, p#**)

A relational schema **SHIPMENT(s#, city, status, p#, quantity)** is **not** in **2NF** because a nonprime attribute **city** depends on a subset of a minimal key (**s#, p#**),  $s\# \rightarrow city$

# Normalization of relational schemas

A relational schema  $\text{SHIPMENT}(s\#, \text{city}, \text{status}, p\#, \text{quantity})$  should be decomposed into the relational schemas

$\text{SP}(s\#, p\#, \text{quantity})$  with a minimal key  $(s\#, p\#)$

$\text{SUPPLIER}(s\#, \text{city}, \text{status})$  with a minimal key  $(s\#)$

A relational schema  $\text{SP}(s\#, p\#, \text{quantity})$  is in BCNF because  $s\#, p\# \rightarrow \text{quantity}$ , i.e. left hand side of the functional dependency is a superkey

A relational schema  $\text{SUPPLIER}(s\#, \text{city}, \text{status})$  is not in 3NF because an attribute  $\text{status}$  is transitively dependent on an attribute  $s\#$ , i.e.  $s\# \rightarrow \text{city}$  and  
 $s\# \rightarrow \text{status}$

A relational schema  $\text{SUPPLIER}(s\#, \text{city}, \text{status})$  is not in 3NF because left hand side of functional dependency  $\text{city} \rightarrow \text{status}$  is not a superkey and right hand side is not prime attribute

# Normalization of relational schemas

A relational schema **SUPPLIER(s#, city, status)** should be decomposed into the relational schemas

**SUPPLIERCITY(s#, city)** with a minimal key (**s#**)

**LOCATION(city, status)** with a minimal key (**city**)

A relational schema **SUPPLIERCITY(s#, city)** is in **BCNF** because  $s\# \rightarrow city$ , i.e. left hand side of the functional dependency is a superkey

A relational schema **LOCATION(city, status)** is in **BCNF** because  $city \rightarrow status$ , i.e. left hand side of the functional dependency is a superkey

# Normalization of relational schemas

A relational schema  $\text{SUPPLIER}(s\#, \text{city}, \text{status})$  can be alternatively decomposed into the relational schemas  
 $\text{SUPPLIERCITY}(s\#, \text{city})$  with a minimal key ( $s\#$ )  
 $\text{SUPPLIERSTAT}(s\#, \text{status})$  with a minimal key ( $s\#$ )

A relational schema  $\text{SUPPLIERCITY}(s\#, \text{city})$  is in **BCNF** because  $s\# \rightarrow \text{city}$ , i.e. left hand side of the functional dependency is a superkey

A relational schema  $\text{SUPPLIERSTAT}(s\#, \text{status})$  is in **BCNF** because  $s\# \rightarrow \text{status}$ , i.e. left hand side of the functional dependency is a superkey

# References

T. Connolly, C. Begg, Database Systems, A Practical Approach to Design, Implementation, and Management, Chapter 14.5 The Process of Normalization, Chapter 14.6 First Normal Form (1NF), Chapter 14.7 Second Normal Form (2NF), Chapter 14.8 Third Normal Form (3NF), Chapter 14.9 General definitions of 2NF and 3NF, Chapter 15.2 Boyce-Codd Normal Form (BCNF) Pearson Education Ltd, 2015

# CSCI235 Database Systems

## Normalization in Practice

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Example 1

A relational schema  $R = (A, B, C)$

Functional dependencies:  $AB \rightarrow C$

Keys ?

If  $AB \rightarrow C$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A, B$ )

No other minimal keys

Normal form ?

Left hand side of  $AB \rightarrow C$  is a minimal key  $K = (A, B)$

BCNF

## Example 2

A relational schema  $R = (A, B, C)$

Functional dependencies:  $AB \rightarrow C, C \rightarrow B$

Keys ?

If  $AB \rightarrow C$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A, B$ )

If  $C \rightarrow B$  then through **augmentation rule**  $AC \rightarrow AB$

If  $AC \rightarrow AB$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A, C$ )

Normal form ?

Not BCNF because left hand side of  $C \rightarrow B$  is not a minimal key

3NF because right hand side of  $C \rightarrow B$  is a prime attribute

Decomposition into BCNF ?

$R1 = (C, B), R2 = (A, B)$

## Example 3

A relational schema  $R = (A, B, C)$

Functional dependencies:  $AB \rightarrow C, C \rightarrow B, C \rightarrow A$

Keys ?

If  $AB \rightarrow C$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A, B$ )

If  $C \rightarrow B$  and  $C \rightarrow A$  then through union rule  $C \rightarrow AB$

If  $C \rightarrow AB$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $C$ )

Normal form ?

**BCNF** because left hand side of each functional dependency is a minimal key

## Example 4

A relational schema  $R = (A, B, C)$

Functional dependencies:  $A \rightarrow B$

Keys ?

If  $A \rightarrow B$  is valid in  $R$  then through augmentation rule  $AC \rightarrow BC$

If  $AC \rightarrow BC$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A, C$ )

Normal form ?

not 2NF because a nonprime attribute  $B$  functionally depends ( $A \rightarrow B$ ) on a subset of primary key ( $A, C$ )

Decomposition into BCNF ?

$R1 = (A, B), R2 = (A, C)$  or

$R1 = (A, B), R2 = (B, C)$

## Example 5

A relational schema  $R = (A, B, C)$

Functional dependencies:  $A \rightarrow B, B \rightarrow A$

Keys ?

If  $A \rightarrow B$  then through augmentation rule  $AC \rightarrow BC$

If  $AC \rightarrow BC$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A, C$ )

If  $B \rightarrow A$  then through augmentation rule  $BC \rightarrow AC$

If  $BC \rightarrow AC$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $B, C$ )

## Example 5

Normal form ?

Not BCNF because left hand side of  $A \rightarrow B$  is not a minimal key

3NF because right hand side of  $A \rightarrow B$  is a prime attribute and right hand side of  $B \rightarrow A$  is a prime attribute

Decomposition into BCNF ?

$R1 = (A, B)$ ,  $R2 = (A, C)$  or

$R1 = (A, B)$ ,  $R2 = (B, C)$

## Example 6

A relational schema  $R = (A, B, C)$

Functional dependencies:  $A \rightarrow B, B \rightarrow C$

Keys ?

If  $A \rightarrow B$  and  $B \rightarrow C$  then through **transitivity rule**  $A \rightarrow C$

If  $A \rightarrow B$  and  $A \rightarrow C$  then through **union rule**  $A \rightarrow BC$

If  $A \rightarrow BC$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A$ )

Normal form ?

**Not 3NF** because a non prime attribute  $C$  is transitively dependent on primary key  $A$

**2NF** because no nonprime attribute depends on a part of primary key

## Example 6

Decomposition into BCNF ?

$R1 = (A, B)$ ,  $R2 = (B, C)$  or

$R1 = (A, B)$ ,  $R2 = (A, C)$

## Example 7

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

Functional dependencies:  $A \rightarrow B, A \rightarrow C, B \rightarrow D$

Keys ?

If  $A \rightarrow B$  and  $A \rightarrow C$  then through **union rule**  $A \rightarrow BC$

If  $A \rightarrow B$  and  $B \rightarrow D$  then through **transitivity rule**  $A \rightarrow D$

If  $A \rightarrow BC$  and  $A \rightarrow D$  then through **union rule**  $A \rightarrow BCD$

If  $A \rightarrow BCD$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A$ )

Normal form ?

**Not 3NF** because a non prime attribute  $D$  is transitively dependent on primary key  $A$

**2NF** because no nonprime attribute depends on a part of primary key

## Example 7

Decomposition into BCNF ?

$R1 = (A, B, C), R2 = (B, D)$

## Example 8

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

Functional dependencies:  $A \rightarrow B$ ,  $B \rightarrow D$ ,  $C \rightarrow B$

Keys ?

If  $A \rightarrow B$  and  $B \rightarrow D$  then through **transitivity rule**  $A \rightarrow D$

If  $A \rightarrow D$  and  $A \rightarrow B$  then through **union rule**  $A \rightarrow BD$

If  $A \rightarrow BD$  then through **augmentation rule**  $AC \rightarrow BCD$

If  $AC \rightarrow BCD$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A, C$ )

If  $C \rightarrow B$  and  $B \rightarrow D$  then through **transitivity rule**  $C \rightarrow D$

If  $C \rightarrow D$  and  $C \rightarrow B$  then through **union rule**  $C \rightarrow BD$

If  $C \rightarrow BD$  then through **augmentation rule**  $AC \rightarrow ABD$

If  $AC \rightarrow BCD$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A, C$ )

## Example 8

Normal form ?

Not 2NF because a nonprime attribute **B** depends on a part of a primary key

(**A, C**)

Decomposition into BCNF ?

$R1 = (A, B)$ ,  $R2 = (B, C)$ ,  $R3 = (B, D)$

## Example 9

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

Functional dependencies:  $A \rightarrow B, A \rightarrow C, B \rightarrow A, B \rightarrow C$

Keys ?

If  $A \rightarrow B$  and  $A \rightarrow C$  then through **union rule**  $A \rightarrow BC$

If  $A \rightarrow BC$  then through **augmentation rule**  $AD \rightarrow BCD$

If  $AD \rightarrow BCD$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A, D$ )

If  $B \rightarrow A$  and  $B \rightarrow C$  then through **union rule**  $B \rightarrow AC$

If  $B \rightarrow AC$  then through **augmentation rule**  $BD \rightarrow ACD$

If  $BD \rightarrow ACD$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $B, D$ )

## Example 9

Normal form ?

Not 2NF because a nonprime attribute **C** depends on a part of a primary key

(**B**, **D**)

Decomposition into BCNF ?

$R_1 = (A, B)$ ,  $R_2 = (B, C)$ ,  $R_3 = (A, D)$

## Example 10

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

Functional dependencies:  $AB \rightarrow C, C \rightarrow D, D \rightarrow A, D \rightarrow B$

Keys ?

If  $AB \rightarrow C$  and  $C \rightarrow D$  then through **transitivity rule**  $AB \rightarrow D$

If  $AB \rightarrow D$  and  $AB \rightarrow C$  then through **union rule**  $AB \rightarrow CD$

If  $AB \rightarrow CD$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A, B$ )

If  $D \rightarrow A$  and  $D \rightarrow B$  then through **union rule**  $D \rightarrow AB$

If  $D \rightarrow AB$  and  $AB \rightarrow C$  then through **transitivity rule**  $D \rightarrow C$

If  $D \rightarrow C$  and  $D \rightarrow AB$  then  $D \rightarrow ABC$

If  $D \rightarrow ABC$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $D$ )

## Example 10

If  $C \rightarrow D$  and  $D \rightarrow AB$  then through **transitivity rule**  $C \rightarrow AB$

If  $C \rightarrow D$  and  $C \rightarrow AB$  then through **union rule**  $C \rightarrow ABD$

If  $C \rightarrow ABD$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $C$ )

Normal form ?

**BCNF** because left hand side of each functional dependency is a superkey

## Example 11

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

Functional dependencies:  $A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow A$

Keys ?

If  $A \rightarrow B$  and  $B \rightarrow C$  then through **transitivity rule**  $A \rightarrow C$

If  $A \rightarrow C$  and  $C \rightarrow D$  then through **transitivity rule**  $A \rightarrow D$

If  $A \rightarrow B$  and  $A \rightarrow C$  and  $A \rightarrow D$  then through **union rule**  $A \rightarrow BCD$

If  $A \rightarrow BCD$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $A$ )

If  $B \rightarrow C$  and  $C \rightarrow D$  then through **transitivity rule**  $B \rightarrow D$

If  $B \rightarrow D$  and  $D \rightarrow A$  then through **transitivity rule**  $B \rightarrow A$

If  $B \rightarrow C$  and  $B \rightarrow D$  and  $B \rightarrow A$  then through **union rule**  $B \rightarrow ACD$

If  $B \rightarrow ACD$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $B$ )

## Example 11

If  $C \rightarrow D$  and  $D \rightarrow A$  then through transitivity rule  $C \rightarrow A$

If  $C \rightarrow A$  and  $A \rightarrow B$  then through transitivity rule  $C \rightarrow B$

If  $C \rightarrow A$  and  $C \rightarrow B$  and  $C \rightarrow D$  then through union rule  $C \rightarrow ABD$

If  $C \rightarrow ABD$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $C$ )

If  $D \rightarrow A$  and  $A \rightarrow B$  then through transitivity rule  $D \rightarrow B$

If  $D \rightarrow B$  and  $B \rightarrow C$  then through transitivity rule  $D \rightarrow C$

If  $D \rightarrow A$  and  $D \rightarrow B$  and  $D \rightarrow C$  then through union rule  $D \rightarrow ABC$

If  $D \rightarrow ABC$  is valid in  $R$  and it covers entire relational schema then its left hand side is a minimal key ( $D$ )

Normal form ?

**BCNF** because left hand side of each functional dependency is a superkey

# References

T. Connolly, C. Begg, Database Systems, A Practical Approach to Design, Implementation, and Management, Chapter 14.5 The Process of Normalization, Chapter 14.6 First Normal Form (1NF), Chapter 14.7 Second Normal Form (2NF), Chapter 14.8 Third Normal Form (3NF), Chapter 14.9 General definitions of 2NF and 3NF, Chapter 15.2 Boyce-Codd Normal Form (BCNF), Chapter 15.3 Review of Normalization Up to BCNF, Pearson Education Ltd, 2015

# CSCI235 Database Systems

## Beyond BCNF

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Beyond BCNF

## Outline

Data explosion problem

Multivalued dependency

4NF

Join dependency

5NF

# Data explosion problem

A relational schema

`EMP(employee-number, programming-language, operating-system)`

has no valid functional dependencies

A relational table `EMPLOYEE` over a relational schema `EMP` contains information about the programming languages and operating systems known by employees

employee-number	programming-language	operating-system	EMPLOYEE
200	Python	Linux	
200	Java	Linux	
200	Scala	Linux	
200	Python	Windows 10	
200	Java	Windows 10	
200	Scala	Windows 10	

# Data explosion problem

employee-number	programming-language	operating-system	EMPLOYEE
200	Python	Linux	
200	Java	Linux	
200	Scala	Linux	
200	Python	Windows 10	
200	Java	Windows 10	
200	Scala	Windows 10	
200	Fortran	Linux	
200	Fortran	Windows 10	
200	Python	Unix	
200	Java	Unix	
200	Scala	Unix	
200	Fortran	Unix	

# Data explosion problem

employee-number	programming-language	operating-system	EMPLOYEE
200	Python	Linux	
200	Java	Linux	
200	Scala	Linux	
200	Python	Windows 10	
200	Java	Windows 10	
200	Scala	Windows 10	
200	Fortran	Linux	
200	Fortran	Windows 10	
200	Python	Unix	
200	Java	Unix	
200	Scala	Unix	
200	Fortran	Unix	

Normal form ?

No valid functional dependencies means that does not exists a functional dependence such that its left hand side is not a superkey

It means that no functional dependencies violate **BCNF**

**BCNF** but ... still a lot of redundancies !

# Beyond BCNF

## Outline

Data explosion problem

Multivalued dependency

4NF

Join dependency

5NF

# Multivalued dependency

Let  $R = (A_1, \dots, A_n)$  be a relational schema and let  $X, Y, Z$  be nonempty subsets of  $R$

We say that multivalued dependency  $X \rightarrow\!\!\!\rightarrow Y|Z$  is valid in relational schema  $R$  if ...

... for any relational table  $r$  created over a relational schema  $R$ , if for any two rows  $v$  and  $w$  in  $r$  such that  $v[X] = w[X]$  there exist a row  $t$  in  $r$  such that ...

...  $v[XY] = t[XY]$  and  $w[XZ] = t[XZ]$

Other notation

X	Y		Multivalued dependency
X		Z	
-----			
X	Y	Z	

It means that if a row  $\textcolor{teal}{X} \textcolor{teal}{Y} \textcolor{brown}{\square}$  is in a relational table and a row  $\textcolor{teal}{X} \textcolor{brown}{\square} \textcolor{teal}{Z}$  is in the same table then a row  $\textcolor{teal}{X} \textcolor{teal}{Y} \textcolor{teal}{Z}$  must be in the same relational table

# Multivalued dependency

## Examples

employee-number →→ programming-language | operating-system

employee-number	programming-language	Multivalued dependency
employee-number		operating-system
employee-number	programming-language	operating-system

A person owns many cars and has many skills

first-name, last-name →→ registration-number | skill

A student has many friends and many hobbies

student-number →→ first-name, last-name | hobby

```
CREATE VIEW XY AS (SELECT X,Y FROM R);
CREATE VIEW XZ AS (SELECT X,Z FROM R);
SELECT XY.X, XY.Y, XZ.Z
FROM XY JOIN XZ ON XY.X = XZ.X
```

Multivalued dependency

The result of **SELECT** is always equal to **R**

# Beyond BCNF

## Outline

Data explosion problem

Multivalued dependency

4NF

Join dependency

5NF

# 4NF

A relational schema  $R$  is in the **Fourth Normal Form (4NF)** if for every nontrivial multivalued dependency  $X \rightarrow\!\!\! \rightarrow Y|Z$  a set of attributes  $X$  is a superkey in a relational schema  $R$

Alternative definition:

A relational schema  $R$  is in **4NF** if no nontrivial multivalued dependencies are valid in a relational schema  $R$

A multivalued dependency

$\text{employee-number} \rightarrow\!\!\! \rightarrow \text{programming-language} | \text{operating-system}$   
is valid in a relational schema

$\text{EMP}(\text{employee-number}, \text{programming-language}, \text{operating-system})$

A relational schema  $\text{EMP}$  is **NOT** in **4NF** because a nontrivial multivalued dependency is valid in  $\text{EMP}$

# 4NF

Decomposition into **4NF** ?

EPGM(*employee-number*, *programming-language*),  
EOPS(*employee-number*, *operating-system*)

# Beyond BCNF

## Outline

Data explosion problem

Multivalued dependency

4NF

Join dependency

5NF

# Join dependency

Let  $R = (A_1, \dots, A_n)$  be a relational schema and let  $X, Y_1, \dots, Y_n$  be nonempty subsets of  $R$

We say that join dependency  $\bowtie(X, Y_1, \dots, Y_n)$  is valid in a relational schema  $R$  if ...

... for any relational table  $r$  with relational schema  $R$ , if for any  $n$  rows  $v_1, \dots, v_n$  in  $r$  such that  $v_1[X] = \dots = v_n[X]$  there exist a row  $t$  in  $r$  such that ...

...  $v_1[XY_1] = t[XY_1]$  and ... and  $v_n[XY_n] = t[XY_n]$

Other notation

		Multivalued dependency
X	Y1	
X	Y2	
X	Y3	
...	...	...
X	...	...
	YN	
<hr/>		
X	Y1	Y2
	Y3	...
	YN	

# Join dependency

## Examples

$\bowtie(\text{employee-number}, \text{programming-language}, \text{operating-system}, \text{hobby})$

employee-number	programming-language	Join dependency	
employee-number		operating-system	
employee-number		hobby	
<hr/>			
employee-number	programming-language	operating-system	hobby

A person owns many cars and has many skills and has many employers

$\bowtie((\text{first-name}, \text{last-name}), \text{registration-number}, \text{skill}, \text{employer})$

CREATE VIEW XY1 AS (SELECT X,Y1 FROM R);	Join dependency
CREATE VIEW XY2 AS (SELECT X,Y2 FROM R);	
.....	
CREATE VIEW XYN AS (SELECT X,YN FROM R);	
SELECT XY1.X, XY1.Y1, XY2.Y2, ... XYN.YN FROM XY1 JOIN XY2 ON XY1.X = XY2.X JOIN ... JOIN XYN ON XY1.X = XYN.X	

The result of **SELECT** is always equal to **R**

# Beyond BCNF

## Outline

Data explosion problem

Multivalued dependency

4NF

Join dependency

5NF

# 5NF

A relational schema  $R$  is in the **Fifth Normal Form (5NF)** if for every nontrivial join dependency  $\bowtie(X, Y_1, \dots, Y_n)$  a set of attributes  $X$  is a superkey in  $R$

Alternative definition:

A relational schema  $R$  is in **5NF** if no nontrivial join dependencies are valid in schema  $R$

A join dependency  $\bowtie(\text{employee-number}, \text{programming-language}, \text{operating-system}, \text{hobby})$  is valid in a relational schema  
 $\text{EMP}(\text{employee-number}, \text{programming-language}, \text{operating-system}, \text{hobby})$

A relational schema  $\text{EMP}$  is **NOT** in **5NF** because a nontrivial join dependency is valid in  $\text{EMP}$

# 5NF

Decomposition into **5NF** ?

EPGM(*employee-number*, *programming-language*),  
EOPS(*employee-number*, *operating-system*),  
EHOB(*employee-number*, *hobby*)

# References

T. Connolly, C. Begg, *Database Systems, A Practical Approach to Design, Implementation, and Management*, Chapter 15.4 Fourth Normal Form (4NF), Chapter 15.5 Fifth Normal Form (5NF), Pearson Education Ltd, 2015

# CSCI235 Database Systems

## Introduction to Indexing

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Introduction to Indexing

## Outline

Index ? What is it ?

Index versus indexed file organization

Primary (unique) index

Secondary (nonunique) index

Clustered index

B\*-tree index implementation

Traversals of B\*-tree index

Examples

# Index? What is it?

An **index** is a data structure that organizes data records on disk to optimize certain kinds of retrieval operations

An **index** is used to efficiently retrieve all records that satisfy a search condition on the search key fields of the index

An index is a function  $f : K \rightarrow \wp(id_R)$  where  $K$  is set of keys and

$\wp(id_R)$  is a powerset (a set of all sets) of identifiers (addresses)  $id_R$  of the records in a set  $R$

Let  $EMP$  be a relational table over a relational schema

$Employee(enumber, name, department)$

Then,  $F_{department} : domain(department) \rightarrow \wp(id_{EMP})$  is a function that maps the names of departments in  $domain(DEPARTMENT)$  into the sets of identifiers of rows  $\wp(id_{EMP})$  in relational table  $EMP$

$F_{department}(d)$  returns the identifiers of all rows where a value of attribute  $department$  is equal to  $d$

# Introduction to Indexing

## Outline

[Index ? What is it ?](#)

[Index versus indexed file organization](#)

[Primary \(unique\) index](#)

[Secondary \(nonunique\) index](#)

[Clustered index](#)

[B\\*-tree index implementation](#)

[Traversals of B\\*-tree index](#)

[Examples](#)

# Index versus indexed file organization

An **indexed file organization (index organized file)** is a function  $f : K \rightarrow \wp(R)$  where  $K$  is a set of keys and  $\wp(R)$  is a powerset (a set of sets) of records  $R$

An **index** maps a **value** into **set of row identifiers**

An **index organized file** maps a **value** into a **set of records**

A **relational table** can be **indexed** or it can be **index organized**

An **indexed relational table** consists of several **index(es)** created separately from implementation of a **relational table** itself

An **index organized relational table** consists only of implementation of one **index** where an **index key** is the same as a **relational schema** of an **index organized table**

Indexing in database systems is **transparent to data manipulation and data retrieval operations**

It means that a database system automatically **modifies an index** and automatically decides whether an **index is used for search**

# Introduction to Indexing

## Outline

[Index ? What is it ?](#)

[Index versus indexed file organization](#)

[Primary \(unique\) index](#)

[Secondary \(nonunique\) index](#)

[Clustered index](#)

[B\\*-tree index implementation](#)

[Traversals of B\\*-tree index](#)

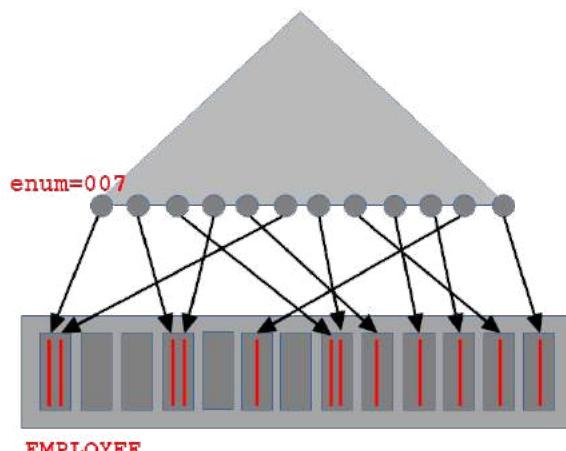
[Examples](#)

# Primary (unique) index

A **primary (unique) index** is an index on a set of attributes equal to **primary** or **candidate key**

A **primary index** is a function  $f : K \rightarrow id_R$  where  $K$  is a set of **key values** and  $id_R$  is a set of identifiers (physical addresses) of rows in a relational table  $R$

A **primary index** maps an **index key** into a **single row identifier** (physical address of a row)



$F_{enum} : \text{domain}(enum) \rightarrow id_{EMPLOYEE}$

[TOP](#)

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

7/30

# Primary (unique) index

A **primary key** in a relational table is always **automatically indexed** by a database system

For example, a relational table **EMPLOYEE** created over a relational schema **Employee(enum, name, department)** where **enum** is a **primary key** has an index automatically created on an attribute (**enum**)

For example, a relational table **ENROLMENT** created over a relational schema **Enrolment(snumber,code,edate)** where **(snumber,code)** is a **primary key** has an index automatically created on a set of attributes **(snumber,code)**

An index on **(snumber,code)** is a **composite index**

# Introduction to Indexing

## Outline

Index ? What is it ?

Index versus indexed file organization

Primary (unique) index

Secondary (nonunique) index

Clustered index

B\*-tree index implementation

Traversals of B\*-tree index

Examples

# Secondary (nonunique) index

A **secondary index** is an index which is not **primary**

A **secondary index** is a function  $f : K \rightarrow \wp(R)$  where **K** is a set of **key values** and  $\text{id}_R$  is a set of identifiers (physical addresses) of rows in a relational table **R**

A **secondary index** maps and **index key** into **a set of row identifiers** (a set of physical addressess of the rows)



$$F_{\text{department}}: \text{domain}(\text{department}) \rightarrow \wp(\text{id}_{\text{EMPLOYEE}})$$

## Secondary (nonunique) index

For example, an index on an attribute (`name`) in a relational table `EMPLOYEE` created over a relational schema `Employee (enum, name, department)` is a **secondary (nonunique) index**

For example, an index on a set of attributes (`name, department`) in a relational table `EMPLOYEE` created over a relational schema `Employee (enum, name, department)` is a **secondary index**

For example, an index on an attribute (`snumber`) in a relational table `ENROLMENT` created over a relational schema `Enrolment (snumber, code, edate)` is a **secondary index**

An index on a set of attributes (`enum, name`) in a relational table `EMPLOYEE` created over a relational schema `Employee (enum, name, department)` is still a **primary index** because (`enum, name`) is a **superkey**

# Introduction to Indexing

## Outline

Index ? What is it ?

Index versus indexed file organization

Primary (unique) index

Secondary (nonunique) index

Clustered index

B\*-tree index implementation

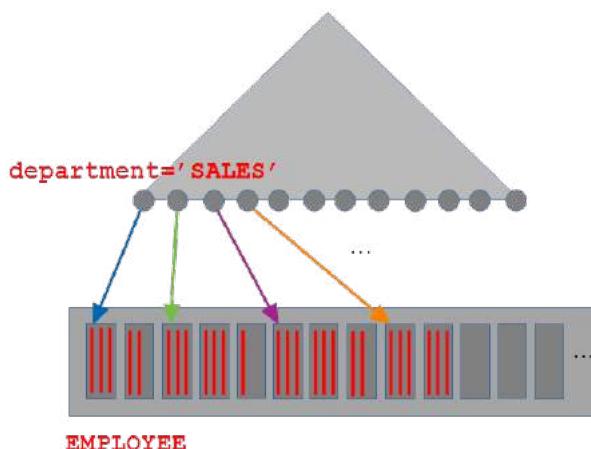
Traversals of B\*-tree index

Examples

# Clustered index

A **clustered index** is an index organized such that the ordering of rows is the same as ordering of keys in the index

A clustered index is a function  $f: K \rightarrow id_R$  where  $K$  is a set of keys and  $id_R$  is a set of row identifiers (addresses) in a relational table  $R$  such that  $f(v)$  returns row identifier (address) of the first row in a sequence of rows such that a value of attribute  $K$  is equal to  $v$



$f_{\text{department}}: \text{domain}(\text{department}) \rightarrow id_{\text{EMPLOYEE}}$

# Clustered index

Every primary index is clustered

Clustered index provides faster access to data than nonclustered secondary index

Clustered index has a very negative impact on performance of **INSERT** and **UPDATE** SQL statements

Therefore, clustered indexing should be applied to mainly to read-only data

# Introduction to Indexing

## Outline

Index ? What is it ?

Index versus indexed file organization

Primary (unique) index

Secondary (nonunique) index

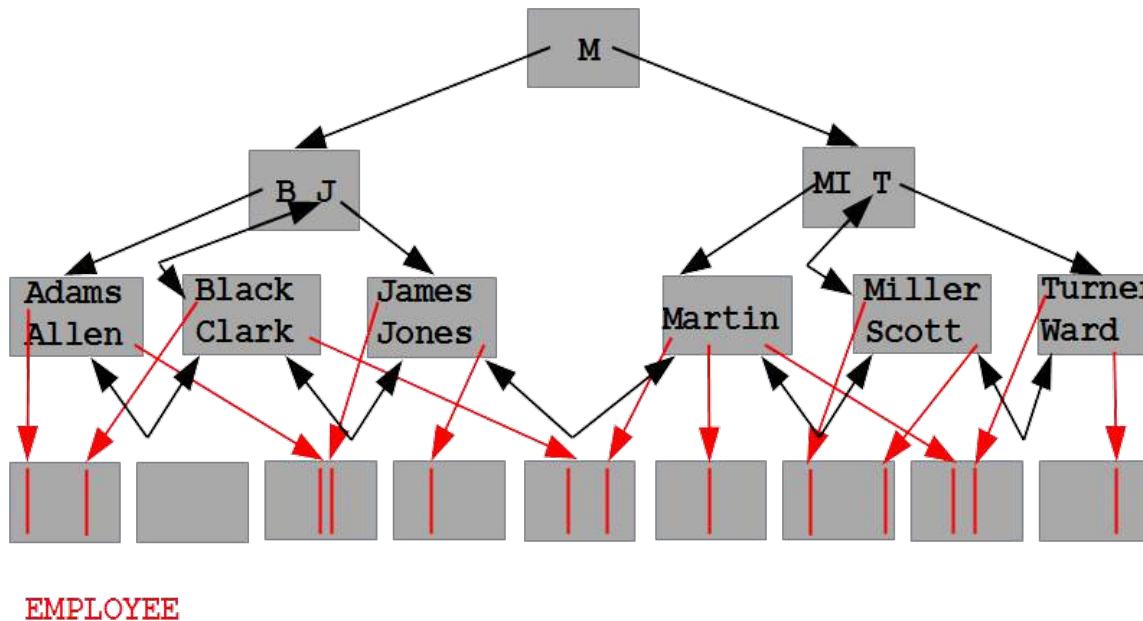
Clustered index

B\*-tree index implementation

Traversals of B\*-tree index

Examples

# B\*-tree index implementation



B\*-tree can be traversed either:

- vertically from root to leaf level of a tree
- horizontally either from left corner of leaf level to right corner of leaf level or the opposite
- vertically and later on horizontally either towards left lower corner or right lower corner of leaf level

# Introduction to Indexing

## Outline

Index ? What is it ?

Index versus indexed file organization

Primary (unique) index

Secondary (nonunique) index

Clustered index

B\*-tree index implementation

Traversals of B\*-tree index

Examples

# Traversals of B\*-tree index

An index on a primary key (`enum`) in a relational table `EMPLOYEE` created over a relational schema

`Employee(enum, name, department, salary)`  
is always built automatically by a database system

A name of an index is the same as a name of primary key constraint in a relational table `EMPLOYEE`

The following queries are processed through a **vertical traversal** of an index on (`enum`)

```
SELECT *
FROM EMPLOYEE
WHERE enum = 007;
```

SQL

```
SELECT *
FROM EMPLOYEE
WHERE enum = 007 AND department = 'MI6';
```

SQL

```
SELECT enum
FROM EMPLOYEE
WHERE enum = 007;
```

SQL

# Traversals of B\*-tree index

The following queries are processed through a **horizontal traversal** of leaf level of an index on (**enum**)

```
SELECT COUNT(*)  
FROM EMPLOYEE;
```

SQL

```
SELECT COUNT(enum)  
FROM EMPLOYEE;
```

SQL

```
SELECT COUNT(name) /* Only if name IS NOT NULL */  
FROM EMPLOYEE;
```

SQL

```
SELECT enum  
FROM EMPLOYEE;
```

SQL

```
SELECT enum, COUNT(*)  
FROM EMPLOYEE  
GROUP BY enum;
```

SQL

```
SELECT enum  
FROM EMPLOYEE  
ORDER BY enum;
```

SQL

# Traversals of B\*-tree index

Assume that we created an index on attribute (`name`) in a relational table

`EMPLOYEE` created over a relational schema

`Employee(enum, name, department, salary)`

The following queries will be processed through a **vertical traversal** of an index on (`name`)

```
SELECT *
FROM EMPLOYEE
WHERE name = 'James';
```

SQL

```
SELECT *
FROM EMPLOYEE
WHERE name = 'James' and department = 'MI6';
```

SQL

```
SELECT count(*)
FROM EMPLOYEE
WHERE name = 'James'
```

SQL

# Traversals of B\*-tree index

Assume that we created an index on the attributes

(`name`, `department`)

in a relational table `EMPLOYEE` created over a relational schema

`Employee(enum, name, department, salary)`

The following queries are processed through a **vertical traversal** of an index on (`name`, `department`)

```
SELECT *
FROM EMPLOYEE
WHERE name = 'James' and department = 'MI6';
```

SQL

```
SELECT count(*)
FROM EMPLOYEE
WHERE name = 'James' and department = 'MI6';
```

SQL

```
SELECT *
FROM EMPLOYEE
WHERE name = 'James' and department = 'MI6' and salary > 1000;
```

SQL

# Traversals of B\*-tree index

The following queries can be processed through a **vertical traversal** and later on **horizontal traversal** of an index on (**enum**)

```
SELECT *
FROM EMPLOYEE
WHERE enum > 300;
```

SQL

```
SELECT count(*)
FROM EMPLOYEE
WHERE enum < 007;
```

SQL

```
SELECT *
FROM EMPLOYEE
WHERE enum > 300 and salary > 1000;
```

SQL

# Traversals of B\*-tree index

Assume that we created an index on the attributes

(`name, department`)

in a relational table `EMPLOYEE` created over a relational schema

`Employee(enum, name, department, salary)`

The following queries can be processed through a vertical traversal and later on horizontal traversal of an index on (`name, department`)

```
SELECT *
FROM EMPLOYEE
WHERE name > 'James';
```

SQL

```
SELECT count(*)
FROM EMPLOYEE
WHERE name <= 'James';
```

SQL

```
SELECT *
FROM EMPLOYEE
WHERE name = 'James' and department > 'MI6';
```

SQL

# Traversals of B\*-tree index

Assume that we created an index on the attributes

(`name, department`)

in a relational table `EMPLOYEE` created over a relational schema

`Employee(enum, name, department, salary)`

The following queries can be processed through a vertical traversal and later on horizontal traversal of an index on (`name, department`)

```
SELECT *
FROM EMPLOYEE
WHERE name > 'James' and salary > 1000;
```

SQL

```
SELECT name, count(*)
FROM EMPLOYEE
WHERE name > 'James' and salary > 1000
GROUP BY name;
```

SQL

```
SELECT *
FROM EMPLOYEE
WHERE name > 'James' and salary > 1000
ORDER BY name;
```

SQL

# Traversals of B\*-tree index

Assume that we created an index on the attributes

(`name, department`)

in a relational table `EMPLOYEE` created over a relational schema

`Employee(enum, name, department, salary)`

The following queries can be processed through a horizontal traversal of an index on (`name, department`)

```
SELECT *
FROM EMPLOYEE
WHERE department = 'MI6;
```

SQL

```
SELECT *
FROM EMPLOYEE
WHERE department > 'MI6;
```

SQL

```
SELECT name, department
FROM EMPLOYEE;
```

SQL

```
SELECT name, department, count(*)
FROM EMPLOYEE
GROUP BY name, department,
```

SQL

# Introduction to Indexing

## Outline

Index ? What is it ?

Index versus indexed file organization

Primary (unique) index

Secondary (nonunique) index

Clustered index

B\*-tree index implementation

Traversals of B\*-tree index

Examples

# Examples

What index should be created on a relational table **DEPARTMENT** created over a relational schema

**Department (dname, chairperson, budget)**  
to speed up the following queries ?

```
SELECT *
  FROM DEPARTMENT
 WHERE dname = 'MI6';
```

SQL

There is no need for any new index because an attribute **dname** is a **primary key** and it is automatically indexed

```
SELECT *
  FROM DEPARTMENT
 WHERE dname = 'MI6' AND budget > 10000;
```

SQL

There is no need for any new index because an attribute **dname** is a **primary key** and it is automatically indexed

# Examples

What index should be created on a relational table **DEPARTMENT** created over a relational schema

**Department (dname, chairperson, budget)**  
to speed up the following queries ?

```
SELECT *  
FROM DEPARTMENT  
WHERE budget = 10000;
```

SQL

```
CREATE INDEX DEPT_IDX_BUDGET ON DEPARTMENT(budget);
```

SQL

```
SELECT *  
FROM DEPARTMENT  
WHERE budget = 10000 and chairperson = 'James';
```

SQL

```
CREATE INDEX DEPT_IDX_BC ON DEPARTMENT(budget,chairperson);
```

SQL

```
SELECT DISTINCT chairperson  
FROM DEPARTMENT;
```

SQL

```
CREATE INDEX DEPT_IDX_CHAIR ON DEPARTMENT(chairperson);
```

SQL

# Examples

What index should be created on a relational table **DEPARTMENT** created over a relational schema

**Department (dname, chairperson, budget)**  
to speed up the following queries ?

```
SELECT *  
FROM DEPARTMENT  
ORDER BY budget;
```

SQL

```
CREATE INDEX DEPT_IDX_BUDGET ON DEPARTMENT(budget);
```

SQL

```
SELECT chairperson, budget, count(*)  
FROM DEPARTMENT  
GROUP BY budget, chairperson;
```

SQL

```
CREATE INDEX DEPT_IDX_BC ON DEPARTMENT(budget,chairperson);
```

SQL

```
SELECT chairperson, budget, count(*)  
FROM DEPARTMENT  
GROUP BY chairperson, budget;
```

SQL

```
CREATE INDEX DEPT_IDX_CB ON DEPARTMENT(chairperson,budget);
```

SQL

# References

Elmasri R. and Navathe S. B., Fundamentals of Database Systems,  
Chapter 17 Indexing Structures for Files and Physical Database Design,  
7th ed., The Person Education Ltd, 2017

# CSCI235 Database Systems

## Stored PL/SQL

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Stored PL/SQL

## Outline

Stored PL/SQL ? What is it ?

Applications

CREATE OR REPLACE PROCEDURE statement

CREATE OR REPLACE FUNCTION statement

GRANT statement revisited

# Stored PL/SQL ? What is it

Stored PL/SQL means **PL/SQL procedures** and **PL/SQL functions** pre-compiled and stored in a **data dictionary** ready to be processed

**Stored procedures** and **functions** can be referenced or called any number of times by multiple applications processing the relational tables

**Stored procedures** and **functions** can accept parameters when processed (called)

**Stored procedures** can be processed (called) with **EXECUTE** statement

**Stored functions** can be processed (called) in SQL statement wherever a function can be used, e.g. as row functions in **SELECT** statement

**Stored procedures** and **stored functions** can be used to extend the functionality of data retrieval and data manipulation statements of SQL (**extensibility**) and to eliminate duplication of code in the database applications (**re-useability**)

# Stored PL/SQL ? What is it

Stored procedures and functions are created with `CREATE OR REPLACE PROCEDURE` and `CREATE OR REPLACE FUNCTION` SQL statements

# Stored PL/SQL

## Outline

Stored PL/SQL ? What is it ?

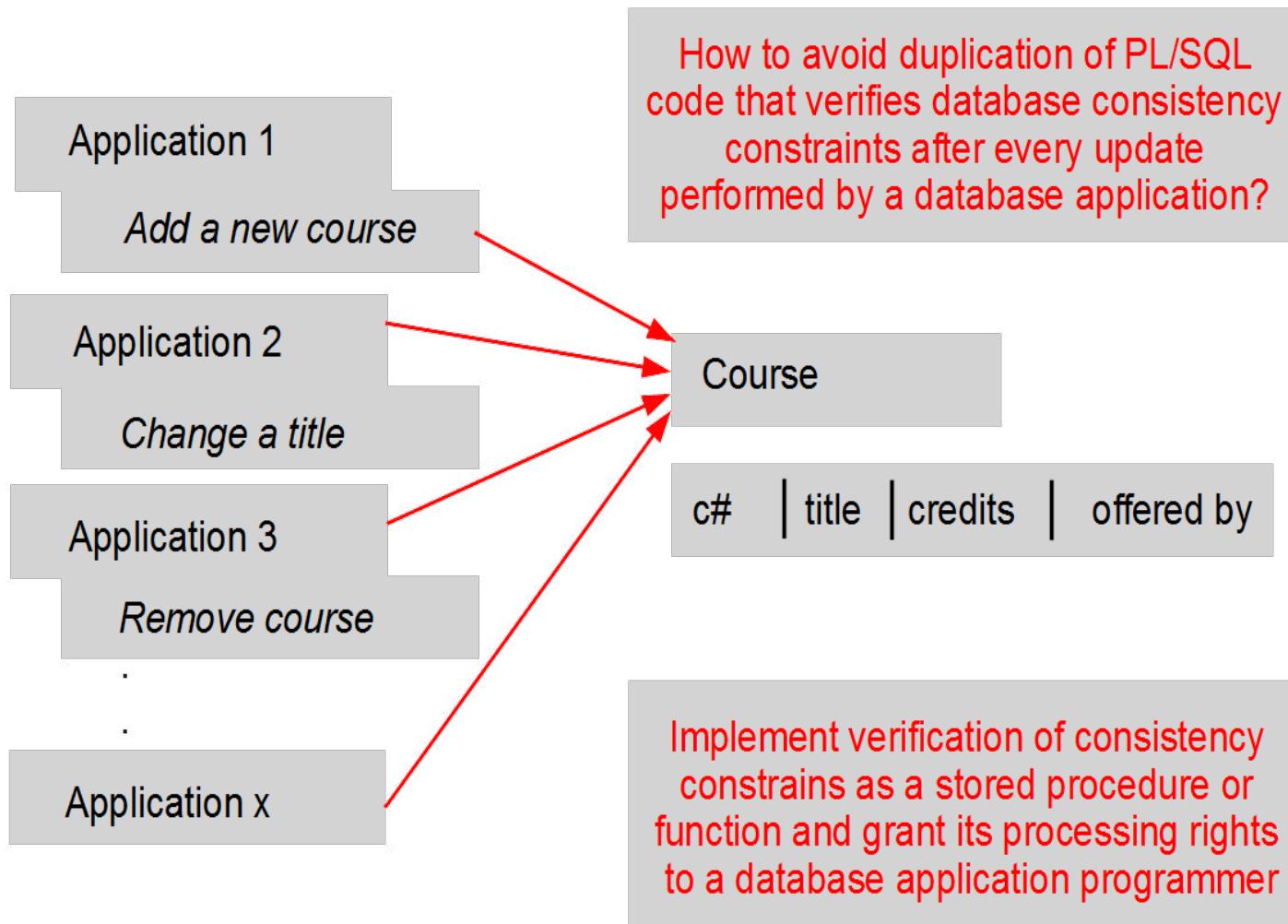
Applications

CREATE OR REPLACE PROCEDURE statement

CREATE OR REPLACE FUNCTION statement

GRANT statement revisited

# Applications - reusability



# Applications - extensibility

Find the names of all departments together with a list of courses offered by each department, display the results in the following form:

DEPARTMENT NAME	LIST OF COURSES OFFERED
Math	Calculus Topology Logic Algebra
Comp Sci	Python Java Databases
Biol	
Phys	Relativity Mechanics
Astro	Astrology

Sample results

Implement a function `LCOURSES( dept_name )` that returns a list of courses offered by a department whose name is a value of a parameter `dept_name`

Use a function `LCOURSES` as a row function in `SELECT` statement

```
SELECT dname, LCOURSES( dname )
FROM DEPARTMENT;
```

Application of stored function

A function `LCOURSES` is called for every row retrieved from a relational table `DEPARTMENT` like any standard row function, e.g. `UPPER` function

[TOP](#)

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

7/15

# Stored PL/SQL

## Outline

Stored PL/SQL ? What is it ?

Applications

CREATE OR REPLACE PROCEDURE statement

CREATE OR REPLACE FUNCTION statement

GRANT statement revisited

# CREATE OR REPLACE PROCEDURE statement

`CREATE OR REPLACE PROCEDURE` statement compiles and stores PL/SQL **procedure** in a data dictionary

The following **stored procedure** `INSERT_COURSE` converts the values of string parameters to upper case and inserts a row into a relational table `COURSE`

```
Header of stored procedure
CREATE OR REPLACE PROCEDURE INSERT_COURSE( cnumber IN NUMBER,
                                         ctitle IN VARCHAR,
                                         ccredits IN NUMBER,
                                         coffer IN VARCHAR) IS

Body of stored procedure
BEGIN
  INSERT INTO COURSE VALUES( cnumber, UPPER(ctitle), ccredits, UPPER(coffer) );
  COMMIT;
END INSERT_COURSE;
```

`EXECUTE` statement is used to process a procedure `INSERT_COURSE`

```
Application of stored procedure
EXECUTE INSERT_COURSE(666, 'Java for kids', 6, 'Comp Sci');
```

# Stored PL/SQL

## Outline

Stored PL/SQL ? What is it ?

Applications

CREATE OR REPLACE PROCEDURE statement

CREATE OR REPLACE FUNCTION statement

GRANT statement revisited

# CREATE OR REPLACE FUNCTION statement

CREATE OR REPLACE FUNCTION statement compiles and stores PL/SQL function in a data dictionary

The following stored function LCOURSES lists the names of departments together with the titles of courses offered by each department

```
Header of stored function
CREATE OR REPLACE FUNCTION LCOURSES( dept_name VARCHAR ) RETURN VARCHAR IS
    Declaration and initialization of local variable
    course_list VARCHAR(300);
BEGIN
    course_list = '';
    Iterations over cursor
    FOR course_cur_rec IN (SELECT title FROM COURSE WHERE offered_by = dept_name);
        Body of cursor
        LOOP
            course_list := course_list || course_cur_rec.title || ' ';
        END LOOP;
    RETURN course_list;
    End LCOURSES;
```

Returning a result

# CREATE OR REPLACE FUNCTION statement

A **stored function** LCOURSES is called as a **row function** in **SELECT** statement

```
SELECT dname, LCOURSES( dname )
FROM COURSE;
```

Application of stored function

# Stored PL/SQL

## Outline

Stored PL/SQL ? What is it ?

Applications

CREATE OR REPLACE PROCEDURE statement

CREATE OR REPLACE FUNCTION statement

GRANT statement revisited

# GRANT statement revisited

In addition to **read** and **write** access rights it is possible to grant **EXECUTE** rights on **stored procedures** and **functions**

For example, a user **scott** grants execution rights on **INSERT\_COURSE** to a user **janusz**

Granting a processing right on a stored procedure

```
GRANT EXECUTE ON INSERT_COURSE TO janusz;
```

Now, a user **janusz** executes a **stored procedure** **INSERT\_COURSE**

Application of stored procedure

```
EXECUTE scott.INSERT_COURSE(958, 'Multimedia Databases', 6, 'Comp Sci');
```

# References

[Database PL/SQL Language Reference](#)

[Database SQL Language Reference, CREATE PROCEDURE](#)

[Database SQL Language Reference, CREATE FUNCTION](#)

[Database SQL Language Reference, GRANT](#)

T. Connolly, C. Begg, Database Systems, A Practical Approach to Design, Implementation, and Management, Chapter 8 Advanced SQL, Pearson Education Ltd, 2015

# CSCI235 Database Systems

## Database Triggers

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Database Triggers

## Outline

Database trigger ? What is it ?

Active database system

CREATE OR REPLACE TRIGGER statement

Statement database triggers

Row database triggers

Problems with database triggers

# Database trigger ? What is it ?

Database trigger is a piece of code stored in a data dictionary and automatically processed whenever a pre-defined event happens and pre-defined condition is satisfied

For example, we would like to automatically increase job level for all employees whose salary is above 100000

```
ON UPDATE OF EMPLOYEE.salary
  IF :NEW.salary > 100000 THEN
    IncreaseJobLevel(:NEW.enumber, :NEW.salary);
  END IF;
```

Database trigger

For example, we would like to implement a data security rule saying that a salary cannot be updated over a weekend

```
ON UPDATE OF EMPLOYEE.salary
  IF TO_CHAR(SYSDATE, 'Day') IN ('Saturday', 'Sunday') THEN
    RAISE_APPLICATION_ERROR(-20001, 'Salary cannot be updated over a weekend !');
  END IF;
```

Database trigger

# Database trigger ? What is it ?

For example, we would like to enforce a consistency constraint saying that a department cannot have more than 100 employees

```
ON INSERT INTO EMPLOYEE
  SELECT COUNT(*)
    INTO total_employees
   FROM EMPLOYEE
 WHERE dname = :NEW.dname;
 IF total_employees = 100 THEN
   RAISE_APPLICATION_ERROR(-20002, 'Too many employees in ' || :NEW.dname);
 END IF;
```

Database trigger

In the example above we assume that a trigger **fires** and it is processed **before** **INSERT** statement

Sometimes it is more convenient to **fire** a trigger that verifies a consistency constraint **after** modification of a relational table and **before** **COMMIT** statement

This is why we have two temporal options for triggers: **BEFORE** and **AFTER**

[TOP](#)

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

4/37

# Database trigger ? What is it ?

What do we need database triggers for ?

- To verify the consistency constraints
- To enforce the sophisticated database access controls
- To implement transparent event logging
- To generate the values of derived attributes
- To maintain replicated data in a distributed database
- To update the relational views

**Active Database Systems** provide functionalities for implementation of database triggers

# Database Triggers

## Outline

Database trigger ? What is it ?

Active database system

CREATE OR REPLACE TRIGGER statement

Statement database triggers

Row database triggers

Problems with database triggers

# Active database system

**Active database system** is a system which is able to detect the **events** that have happened in a certain period of time and in the response to these **events** it is able to execute the **actions** when the **pre-defined conditions** are met

A logic of active database system is implemented as a collection of **Event-Condition-Action (ECA)rules**

In SQL **ECA** rule can be created with **CREATE TRIGGER** statement and it can be deleted with **DROP TRIGGER** statement

Syntax of **ECA rule**:

- (**EVENT, CONDITION, ACTION**)

Semantics of **ECA rule**:

- Whenever an **EVENT** happens and a **CONDITION** is satisfied then a database system performs an **ACTION**

# Active database system

A sample **event**

ON UPDATE OF EMPLOYEE.salary

Trigger

A sample **condition**

IF :NEW.salary > 100000

Trigger

A sample **action**

IncreaseJobLevel(:NEW.enumer, :NEW.salary);

Trigger

**CREATE OR REPLACE TRIGGER** statement implements **ECA** rule

# Database Triggers

## Outline

Database trigger ? What is it ?

Active database system

CREATE OR REPLACE TRIGGER statement

Statement database triggers

Row database triggers

Problems with database triggers

# CREATE OR REPLACE TRIGGER statement

A sample CREATE OR REPLACE TRIGGER statement

CREATE OR REPLACE TRIGGER CheckBudget

Trigger name

Temporal option

BEFORE

Temporal option specification

Event

UPDATE OF budget ON DEPARTMENT

Event specification

Type of trigger, either statement or row trigger

FOR EACH ROW

Row trigger

-- FOR EACH ROW means that it is a row trigger

Condition

WHEN NEW.name = 'Math'

-- NEW is a so called pseudorecord

Trigger condition

# CREATE OR REPLACE TRIGGER statement

Beginning of trigger's body

```
BEGIN
```

Start of trigger's body

Pseudorecords :OLD and :NEW that represents a row before modification or deletion and a row after modification or insertion

```
IF NOT ( :NEW.budget BETWEEN 1 AND 7000 ) THEN
```

Application of correlation variables in a row trigger

Abnormal termination of a trigger together with a transaction that fired a trigger

```
RAISE_APPLICATION_ERROR(-200001, 'Budget of department ' || :NEW.name ||  
' cannot be equal to ' || :NEW.budget );
```

Abnormal termination of a trigger

End of trigger's body

```
END IF;  
END;
```

End of trigger's body

# CREATE OR REPLACE TRIGGER statement

A complete CREATE OR REPLACE TRIGGER statement

A sample row trigger

```
CREATE OR REPLACE TRIGGER CheckBudget
BEFORE UPDATE OF budget ON DEPARTMENT
FOR EACH ROW
WHEN NEW.name = 'Math'
BEGIN
    IF NOT ( :NEW.budget BETWEEN 1 AND 7000 ) THEN
        RAISE_APPLICATION_ERROR(-200001, 'Budget of department ' || :NEW.name ||
                                ' cannot be equal to ' || :NEW.budget );
    END IF;
END;
```

# CREATE OR REPLACE TRIGGER statement

The following **temporal options** are available

- **BEFORE** - a trigger fires before a triggering event
- **AFTER** - a trigger fires after a triggering event
- **INSTEAD OF** - a trigger fires instead of a triggering event, it is typically used to correctly implement **view update** operation i.e. a correct modification of **base relational tables** through an update performed on a **relational view**

Sample applications of **temporal options**

Fire a trigger before **UPDATE** operation on a column **budget** in a relational table **DEPARTMENT**

BEFORE UPDATE OF budget ON DEPARTMENT

A sample temporal option

Fire a trigger after any **DELETE** or **UPDATE** operation performed on **DEPARTMENT** table

AFTER DELETE OR UPDATE ON DEPARTMENT

A sample temporal option

# CREATE OR REPLACE TRIGGER statement

Fire a trigger instead of **UPDATE** operation on a relational view **EMPVIEW**

**INSTEAD OF INSERT ON EMPVIEW**

A sample temporal option

# CREATE OR REPLACE TRIGGER statement

The following events can fire a trigger

- Data Manipulation event - any `INSERT` or `UPDATE` or `DELETE` statement
- Data Definition event - any `CREATE` or `ALTER` or `DROP` statement
- Database events - the events such as a database server error, startup/shutdown of a database server, logon/logoff of a user, etc

Sample applications of DML events

BEFORE UPDATE OF attribute, attribute,... ON table	A sample DML event
AFTER INSERT ON table	A sample DML event
BEFORE DELETE ON table	A sample DML event
AFTER DELETE OR INSERT OR UPDATE ON table	A sample DML event

# CREATE OR REPLACE TRIGGER statement

## Sample applications of DDL events

AFTER ALTER database object	A sample DDL event
BEFORE CREATE database object	A sample DDL event
AFTER DROP database object	A sample DDL event
AFTER GRANT database object	A sample DDL event
BEFORE ANALYZE database object	A sample DDL event
AFTER GRANT system privilege	A sample DDL event

# CREATE OR REPLACE TRIGGER statement

## Sample applications of Database events

AFTER SERVERERROR ON SCHEMA	A sample database event
BEFORE LOGON	A sample database event
BEFORE LOGOFF	A sample database event
AFTER STARTUP	A sample database event
BEFORE SHUTDOWN	A sample database event

# CREATE OR REPLACE TRIGGER statement

Condition determines whether a trigger processes its body after it has been fired

Sample applications of condition

WHEN (condition)	A sample condition
WHEN ( <code>OLD.status = 'BUSY' AND NEW.status = 'AVAILABLE'</code> );	A sample condition
WHEN ( <code>NEW.amount &gt; 1000</code> );	A sample condition
WHEN ( <code>OLD.credits IN (6, 12)</code> );	A sample condition

`OLD` and `NEW` are so called pseudorecords such that for

- `INSERT` triggering operation `OLD` contains no values and `NEW` contains the new values
- `UPDATE` triggering operation `OLD` contains the old values and `NEW` contains the new values
- `DELETE` triggering operation `OLD` contains the old values and `NEW` contains no values

[TOP](#)

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

18/37

# Database Triggers

## Outline

Database trigger ? What is it ?

Active database system

CREATE OR REPLACE TRIGGER statement

Statement database triggers

Row database triggers

Problems with database triggers

# Statement database triggers

A **statement trigger** fires once either before or after a triggering event

A sample **statement** trigger

```
Trigger name  
CREATE OR REPLACE TRIGGER ModifyDepartment  
Temporal option and event specification  
AFTER DELETE OR UPDATE ON DEPARTMENT  
Start of statement trigger's body  
BEGIN      -- Statement triggers have no FOR EACH ROW clause!  
Trigger condition  
IF DELETING THEN  
Trigger's body  
    INSERT INTO DEPTAUDIT VALUES('DELETE', SYSDATE);  
Trigger's body  
ELSIF UPDATING THEN  
Trigger's body  
    INSERT INTO DEPTAUDIT VALUES('UPDATE', SYSDATE);  
End of trigger's body  
END IF;  
End of trigger's body  
END;
```

# Statement database trigger

Assume that the following **UPDATE** statement has been processed and not **COMMIT**ed yet

```
UPDATE DEPARTMENT  
SET budget = budget + 1000  
WHERE budget < 5000;
```

UPDATE statement

3 row updated

Feedback message

The following body of a trigger **ModifyDepartment** has been processed immediately **after** processing of **UPDATE** statement

```
BEGIN  
  IF DELETING THEN  
    INSERT INTO DEPTAUDIT VALUES( 'DELETE' , SYSDATE );  
  ELSIF UPDATING THEN  
    INSERT INTO DEPTAUDIT VALUES( 'UPDATE' , SYSDATE );  
  END IF;  
END;
```

A body of statement trigger

# Database Triggers

## Outline

Database trigger ? What is it ?

Active database system

CREATE OR REPLACE TRIGGER statement

Statement database triggers

Row database triggers

Problems with database triggers

# Row database triggers

A **row trigger** fires either after or before a triggering event affects a row in a relational table

- When a **temporal option BEFORE** is used a trigger fires **once before** a triggering event affects a row in a relational table
- When a **temporal option AFTER** is used a trigger fires **once after** a triggering event affects a row in a relational table

For example, if a **temporal option** and **event** are

**BEFORE INSERT ON DEPARTMENT**

A temporal option and event

then a trigger fires before each insertion into a relational table (it is possible to have many insertions when a multirow **INSERT** statement is processed)

For example, if a **temporal option** and **event** are

**AFTER UPDATE ON EMPLOYEE**

A temporal option and event

then a trigger fires after a row is updated in a relational table, if a triggering event updates **n** rows then a trigger fires **n** times

# Row database triggers

For example, if a **temporal option** and **event** are

AFTER DELETE ON PROJECT

A temporal option and event

Then a trigger fires after a row is deleted from a relational table, if a triggering event deletes **n** rows then a trigger fires **n** times

# Row database triggers

A sample **row** trigger

```
CREATE OR REPLACE TRIGGER UpdateDepartment
AFTER UPDATE ON DEPARTMENT
FOR EACH ROW          -- Row trigger must have FOR EACH ROW clause !
WHEN (NEW.city = 'Boston')-- Only for row triggers!
BEGIN
  INSERT INTO DEPTTRACE VALUES
    ('UPDATE', SYSDATE, :NEW.name, :NEW.budget, :NEW.city,
     :OLD.name, :OLD.budget, :OLD.city );
END;
```

Trigger name

Temporal option and event

Row trigger

Trigger condition

Start of trigger's body

Trigger's body

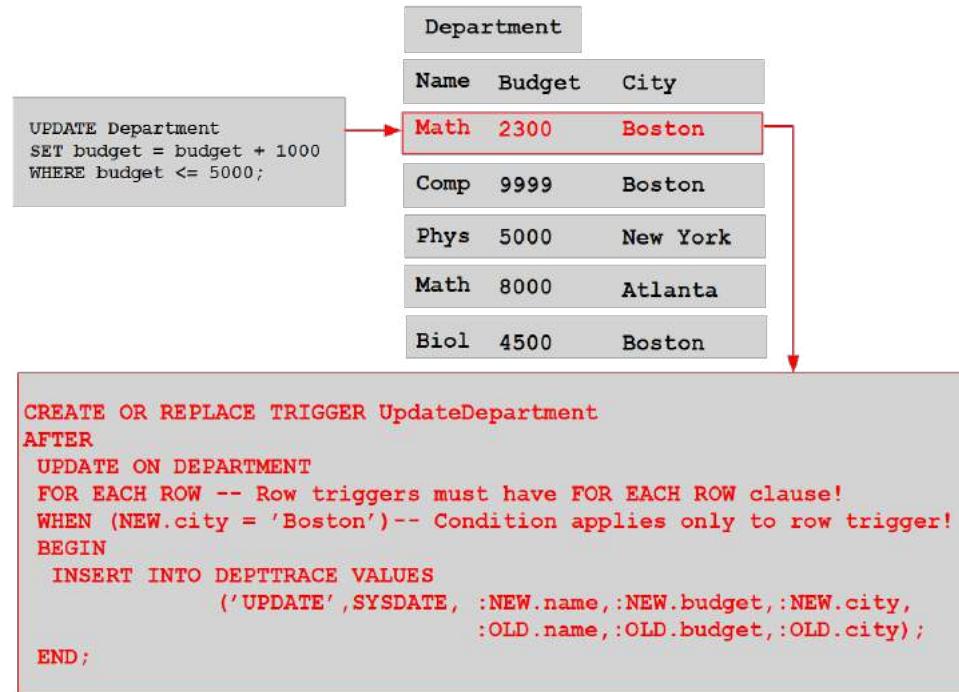
Trigger's body

Trigger's body

End of trigger's body

# Row database triggers

A sample processing of a row database trigger

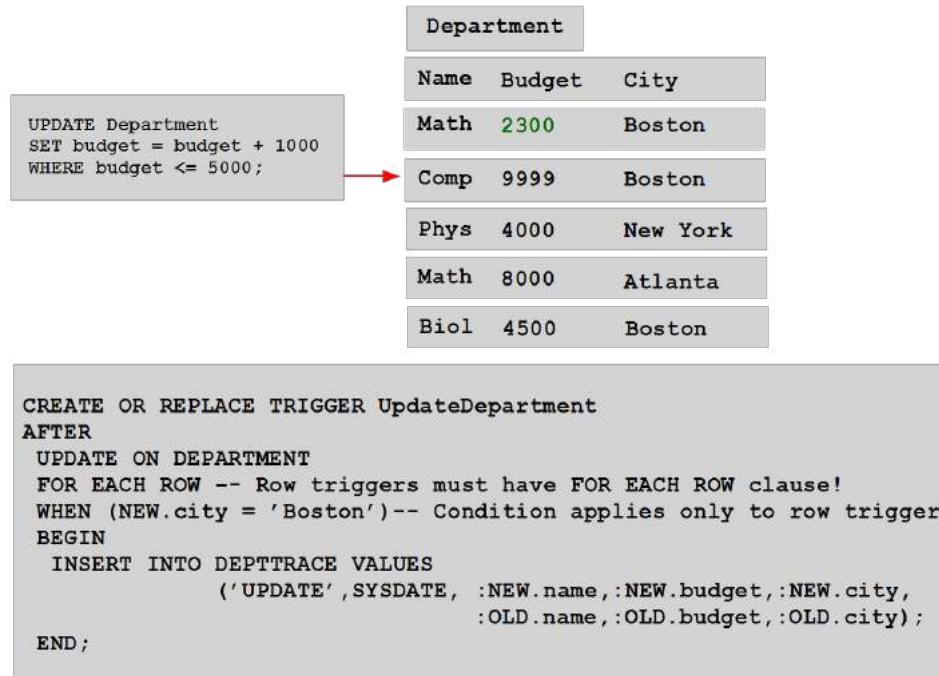


A trigger fires after **UPDATE** of a row **[Math 2300 Boston]**

**WHEN** condition is satisfied and a trigger processes its body

# Row database triggers

A sample processing of a row database trigger

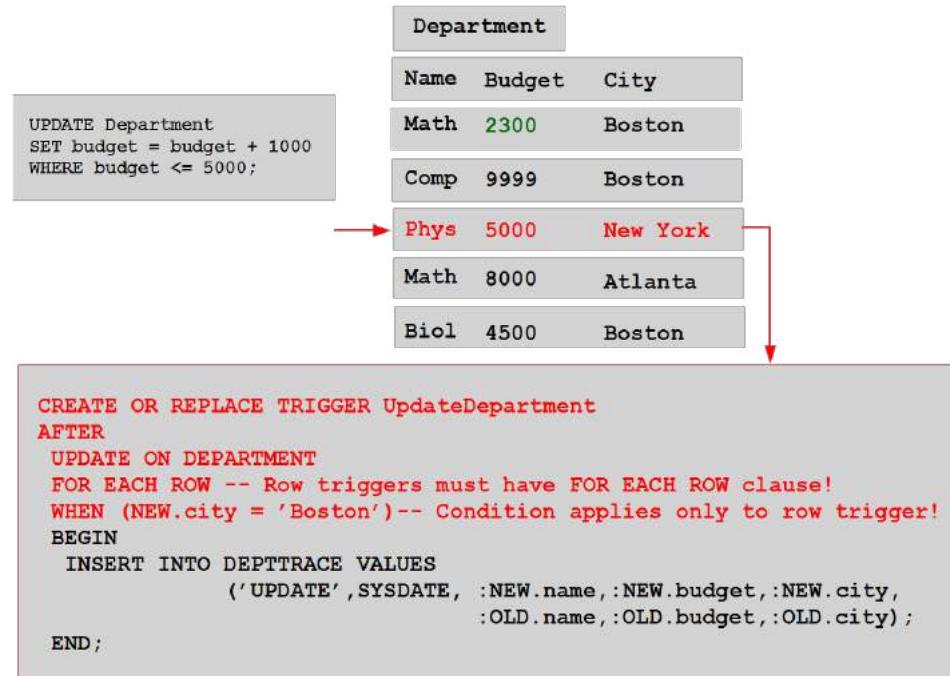


A row **[Comp 9999 Boston]** does not satisfy a condition in `WHERE` clause and it is not `UPDATE`ed

A trigger does not fire

# Row database triggers

A sample processing of a row database trigger

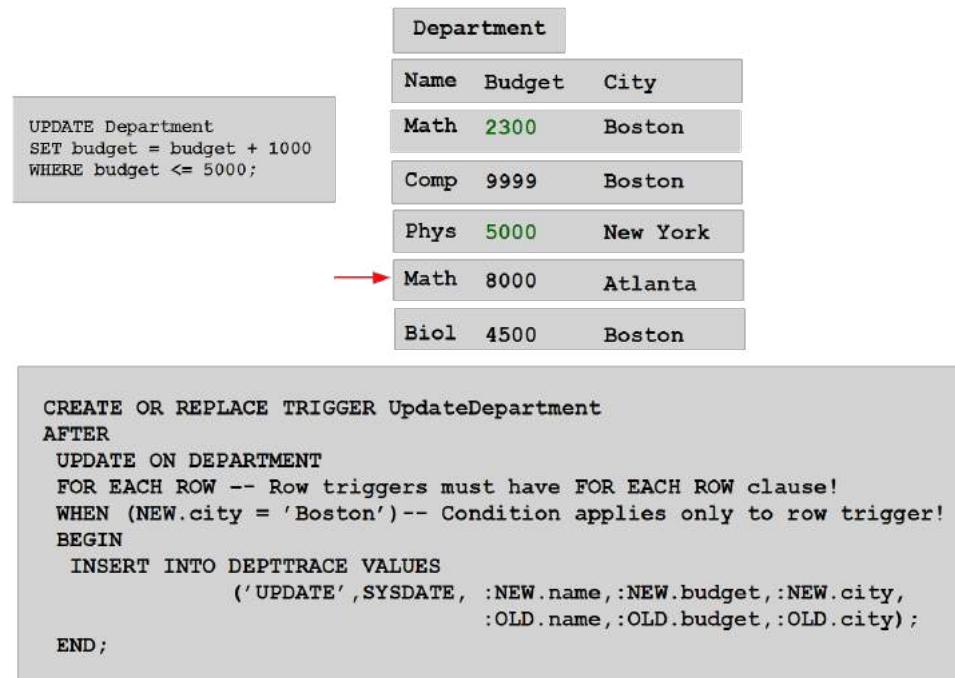


A trigger fires after **UPDATE** of a row [**Phys 5000 New York**]

**WHEN** condition is not satisfied and a trigger does not process its body

# Row database triggers

A sample processing of a row database trigger

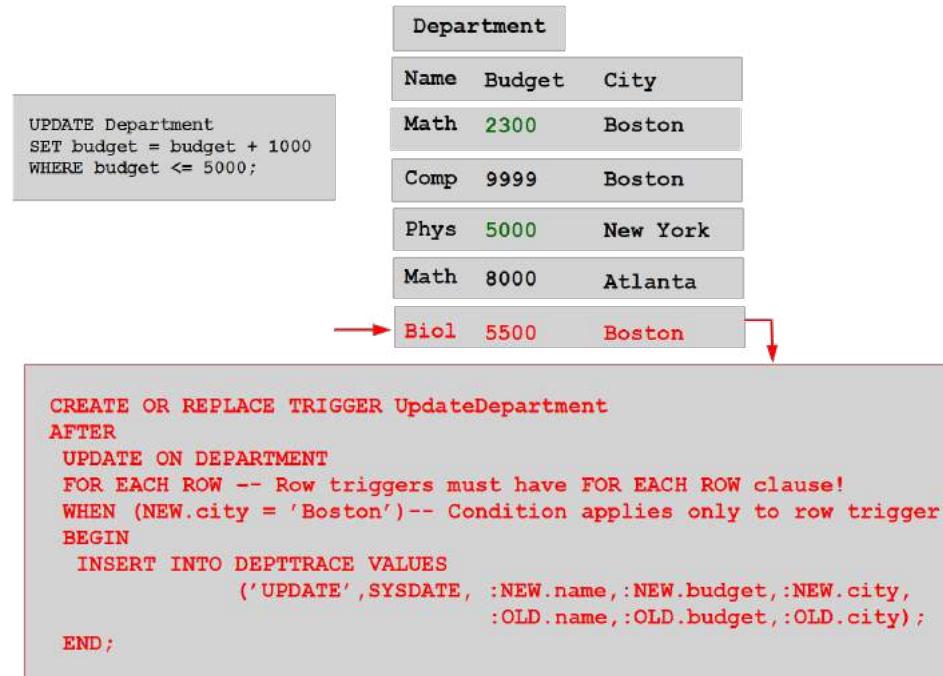


A row [**Math 8000 Atlanta**] does not satisfy a condition in **WHERE** clause and it is not **UPDATE**ed

A trigger does not fire

# Row database triggers

A sample processing of a row database trigger



A trigger fires after **UPDATE** of a row [**Biol 5500 Boston**]

**WHEN** condition is satisfied and a trigger processes its body

# Row database triggers

A sample processing of a row database trigger is completed

Department		
Name	Budget	City
Math	2300	Boston
Comp	9999	Boston
Phys	5000	New York
Math	8000	Atlanta
Biol	5500	Boston

```
CREATE OR REPLACE TRIGGER UpdateDepartment
AFTER
    UPDATE ON DEPARTMENT
    FOR EACH ROW -- Row triggers must have FOR EACH ROW clause!
    WHEN (NEW.city = 'Boston')-- Condition applies only to row trigger!
    BEGIN
        INSERT INTO DEPTTRACE VALUES
            ('UPDATE',SYSDATE, :NEW.name,:NEW.budget,:NEW.city,
             :OLD.name,:OLD.budget,:OLD.city);
    END;
```

# Row database triggers

Assume that while processing a rows trigger it attempts to access a relational table affected by a triggering event

For example, a trigger attempts to count the total number of rows in **UPDATE**ed realtional table

The diagram illustrates a database operation. On the left, a grey box contains an **UPDATE** statement:

```
UPDATE Department
SET budget = budget + 1000
WHERE budget <= 5000;
```

An arrow points from this statement to the **Department** table on the right. The table has columns **Name**, **Budget**, and **City**. It shows the following data after the update:

Department		
Name	Budget	City
Math	2300	Boston
Comp	9999	Boston
Phys	4000	New York
Math	8000	Atlanta
Biol	4500	Boston

Below the table is a code block for a trigger:

```
CREATE OR REPLACE TRIGGER UpdateDepartment
AFTER UPDATE ON DEPARTMENT
FOR EACH ROW -- Row triggers must have FOR EACH ROW clause!
WHEN (NEW.city = 'Boston')-- Condition applies only to row trigger!
BEGIN
    INSERT INTO DEPTTRACE VALUES
        ('UPDATE',SYSDATE, :NEW.name,:NEW.budget,:NEW.city,
         :OLD.name,:OLD.budget,:OLD.city);
    SELECT SUM(budget) FROM DEPARTMENT;
END;
```

What is a correct of summation over a column **budget** ?

[TOP](#)

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

32/37

# Row database triggers

It is impossible to provide a correct result of summation over a column **budget** while an **UPDATE** statement changes the values in the column

An outcome is a **mutating table** error when processing a row trigger

The screenshot shows a SQL developer interface with three main sections:

- Error Message:** A red box containing the following text:

```
ERROR at line 1:
ORA-04091: table SCOTT. DEPARTMENT is
Mutating, trigger/function may not
See it
ORA-06512: at
"SCOTT. UPDATEDEPARTMENT" , line 2 ORA-
04088: error during execution of
Trigger 'SCOTT. UPDATEDEPARTMENT'
```
- Department Table:** A table with columns Name, Budget, and City. One row for 'Math' with a budget of 2300 is highlighted with a red border.
- Trigger Creation Script:** A grey box containing the PL/SQL code for the trigger:

```
CREATE OR REPLACE TRIGGER UpdateDepartment
AFTER UPDATE ON DEPARTMENT
FOR EACH ROW -- Row triggers must have FOR EACH ROW clause!
WHEN (NEW.city = 'Boston')-- Condition applies only to row trigger!
BEGIN
    INSERT INTO DEPTTRACE VALUES
        ('UPDATE',SYSDATE, :NEW.name,:NEW.budget,:NEW.city,
         :OLD.name,:OLD.budget,:OLD.city);
    SELECT SUM(budget) FROM DEPARTMENT;
END;
```

# Row database triggers

The solution to a **mutating table** error problem

- If a trigger fires on `INSERT` then use `BEFORE INSERT` temporal option
- Rewrite a trigger as a statement trigger
- Run a trigger as an **autonomous transaction**
- Record the modifications in a temporary table and fire a row trigger that reapplies the modifications as a statement trigger

# Database Triggers

## Outline

Database trigger ? What is it ?

Active database system

CREATE OR REPLACE TRIGGER statement

Statement database triggers

Row database triggers

Problems with database triggers

# Other problems with triggers

## Infinite chains of trigger invocations

- What to do when a trigger **A** while processing its body fires a trigger **B** and a trigger **B** while processing its body fires a trigger **A** ?

## Indeterministic trigger invocations

- It may happen that due to a database transaction serialization mechanisms the same chain of trigger invocations will be processed (serialized) in many different way by a transaction scheduler, e.g. if two triggers **A** and **B** fire in more or less the same moment in time then sometimes **A** will be processed before **B** and sometimes **B** will be processed before **A**

## Lack of external control

- Long chains of trigger invocations contribute to very serious data security risks, e.g. it is possible to "hide" malicious code at the end of long chains of trigger invocations

## Lack of design methodology

- The ad hoc uncontrolled and not well planned additions of new triggers lead to a situation where after addition or modification of a trigger there is no certainty that the chains of trigger invocations do not corrupt a database

[TOP](#)

# References

T. Connolly, C. Begg, Database Systems, A Practical Approach to Design, Implementation, and Management, Chapter 8.3 Triggers, Pearson Education Ltd, 2015

[Database SQL Language Reference, CREATE\\_TRIGGER](#)

[Database PL/SQL Language Reference, 9 PL/SQL Triggers](#)

# CSCI235 Database Systems

## Introduction to Transaction Processing (1)

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Introduction to Transaction Processing

## Outline

An interesting experiment

Where is a problem ?

Principles of transaction processing

Update synchronisation

ACID properties

Protocols

# An interesting experiment

Use SQLcl to create two simultaneous connections to the same user account

```
$sqlcl jrg
```

SQLcl

```
$sqlcl jrg
```

SQLcl

Next, process the same **SELECT** statement in both connections

```
SQL> SELECT COUNT(*) FROM SKILL;
```

SQL

```
COUNT(*)
```

```
-----
```

```
19
```

```
SQL> SELECT COUNT(*) FROM SKILL;
```

SQL

```
COUNT(*)
```

```
-----
```

```
19
```

Obviously, the results are the same

# An interesting experiment

Now, **INSERT** a row into a relational table **SKILLS** through one of the connections

```
SQL> INSERT INTO SKILL VALUES('singing');  
1 row created.
```

SQL

And now repeat the same **SELECT** statements

```
SQL> SELECT COUNT(*) FROM SKILL;
```

SQL

```
COUNT(*)  
-----  
20
```

```
SQL> SELECT COUNT(*) FROM SKILL;
```

SQL

```
COUNT(*)  
-----  
19
```

Surprise, surprise, the results are different ! Why ?

# Introduction to Transaction Processing

## Outline

An interesting experiment

Where is a problem ?

Principles of transaction processing

Update synchronisation

ACID properties

Protocols

# Where is a problem ?

Why a modification performed by the first user is not visible to the second user ?

Is it correct that the second user must see all modifications performed by the first user ?

What if a modification performed by the first user is immediately visible to the second user and after that the first user rolls back the modification ?

Then, the second user is left with incorrect data !

Hence, only committed data can be revealed to the other users

Is such conclusion always true ?

## Problem statement

- Given a multiuser database system
- Find the most efficient synchronisation method for a set of concurrent processes accessing the shared database resources

# Introduction to Transaction Processing

## Outline

An interesting experiment

Where is a problem ?

Principles of transaction processing

Update synchronisation

ACID properties

Protocols

# Principles of transaction processing

A partially ordered set of **read, write** operations on the database items is called as a **transaction**

Users interact with a database by executing programs

Execution of a program is equivalent to execution of a partially ordered set of **read, write** operations

A database is visible to **transactions** as a collection of data items

Concurrently running **transactions** interleave their operations

**Transactions** have no impact on execution of their operations

Each **transaction** terminates by either **commit** or **abort** operation

Each **transaction** arrives at a consistent database state and must leave a database in a consistent state as well

# Principles of transaction processing

A sample concurrent processing of database transactions

Concurrent processing of database transactions		
T1	T2	x: \$100
a=read(x)		x: \$100 a: \$100
	b=read(x)	x: \$100 b: \$100
write(x,a-10)		x: \$90 a: \$100
	write(x,b+20)	x: \$120 b: \$100
	commit	x: \$120
commit		x: \$120

If a state of a bank account is **\$100** then withdrawal of **\$10** and deposit of **\$20** cannot change a state of bank account to **\$120**

Uncontrolled concurrent processing of database transactions may **corrupt** a database

# Introduction to Transaction Processing

## Outline

An interesting experiment

Where is a problem ?

Principles of transaction processing

Update synchronisation

ACID properties

Protocols

# Update synchronisation

Database transaction can perform update in two different ways:

- A transaction immediately writes uncommitted values into a database - **update-in-place**
- A transaction does not modify a database until the time it commits itself - **deferred-update**

In the last example the transactions applied **update-in-place** to modify a database

A way how the transactions perform an update has no impact on the final outcomes, e.g. when deferred-update is applied a database maybe still corrupted (see the next example)

# Principles of transaction processing

A sample concurrent processing of database transactions when **deferred-update** is applied

Concurrent processing of database transactions		
T1	T2	x: \$100
a=read(x)		x: \$100 a: \$100
	b=read(x)	x: \$100 b: \$100
write(x,a-10)		x: \$100 log T1:\$90
	write(x,b+20)	x: \$100 log T2:\$120
	commit	x: \$120
commit		x: \$90

If a state of a bank account is **\$100** then withdrawal of **\$10** and deposit of **\$20** cannot change a state of bank account to **\$90**

**Deferred-update** does not solve the problem

# Introduction to Transaction Processing

## Outline

An interesting experiment

Where is a problem ?

Principles of transaction processing

Update synchronisation

ACID properties

Protocols

# ACID properties

Processing of database transactions must satisfy **ACID** properties

## Atomicity

- Each database operation is treated as a single unit (all-or-nothing)

## Consistency

- A transaction takes a database from one consistent state to another

## Isolation

- Transactions do not directly communicate one with each other and they do not read the intermediate results of the other transactions

## Durability

- The results of committed transactions must be permanent in a database in spite of failures

# Introduction to Transaction Processing

## Outline

An interesting experiment

Where is a problem ?

Principles of transaction processing

Update synchronisation

ACID properties

Protocols

# Protocols

An **execution atomicity protocol** ensures **Consistency** property

A **failure atomicity protocol** ensures **Atomicity, Isolation** and **Durability** properties

A sample incorrect **execution atomicity protocol**

Concurrent processing of database transactions		
T1	T2	x: \$100
a=read(x)		x: \$100 a: \$100
	b=read(x)	x: \$100 b: \$100
write(x,a-10)		x: \$90 a: \$100
	write(x,b+20)	x: \$120 b: \$100
	commit	x: \$120
commit		x: \$120

# Protocols

## A sample incorrect failure atomicity protocol

Concurrent processing of database transactions		
T1	T2	x: \$100
a=read(x)		x: \$100 a: \$100
write(x,a-10)		x: \$90 a: \$100
	b=read(x)	x: \$90 b: \$90
	write(x,b+20)	x: \$110 b: \$90
	commit	x: \$110
abort		x: \$100

If a state of a bank account is **\$100** then withdrawal of **\$10** and deposit of **\$20** cannot change a state of bank account to **\$100**

# Protocols

Execution atomicity protocol = Concurrency control protocol

Failure atomicity protocol = Recovery protocol

Lost update problem

Concurrent processing of database transactions		
T1	T2	
		x: \$100
a=read(x)		x: \$100 a: \$100
	b=read(x)	x: \$100 b: \$100
write(x,a-10)		x: \$90 a: \$100
	write(x,b+20)	x: \$120 b: \$100
	commit	x: \$120
commit		x: \$120

# Protocols

## Inconsistent retrieval problem

		Concurrent processing of database transactions				
T1	T2	x	y	a	b	c
	a=read(x)	100	50	100		
	b=read(y)	100	50	100	50	
	write(x,a-10)	90	50	100	50	
	c=read(x)	90	50	100	50	90
	write(y,a-30)	90	70	100	50	90
	print(b+c)	90	70	100	50	90

# References

T. Connolly, C. Begg, *Database Systems, A Practical Approach to Design, Implementation, and Management*, Chapter 22.1 Transaction Support, Chapter 22.2 Concurrency Control, Pearson Education Ltd, 2015

# CSCI235 Database Systems

## Introduction to Transaction Processing (2)

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Introduction to Transaction Processing

## Outline

### Correctness

Conflict serializability versus view serializability

Order preserving conflict serializability

Recoverable executions

Cascadeless executions

Strict executions

# Correctness

What makes concurrent execution of database transaction **incorrect** ?

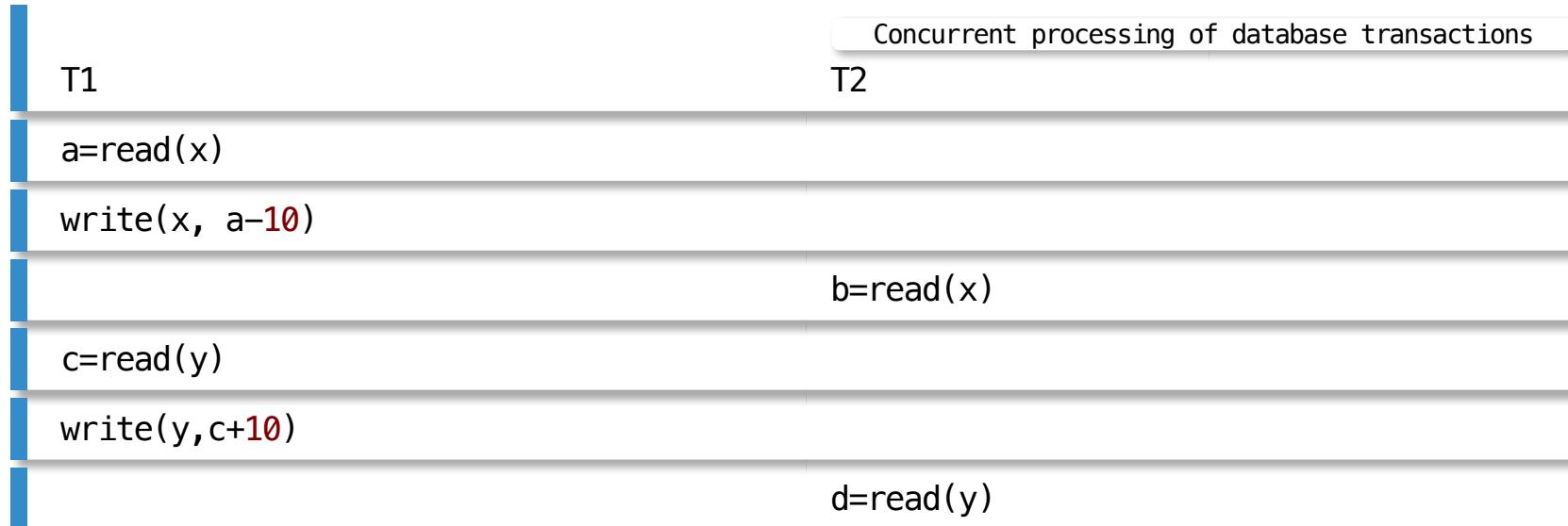
How do we define a **correct** concurrent execution of database transactions ?

Concurrent execution of database transactions is **view serializable** if there **exists a possible serial execution of the same set of transactions** such that in both executions each **transaction reads the same values and the final states of the database are the same**

A concurrent execution of database transactions is **correct** when it is **view serializable**

# Correctness

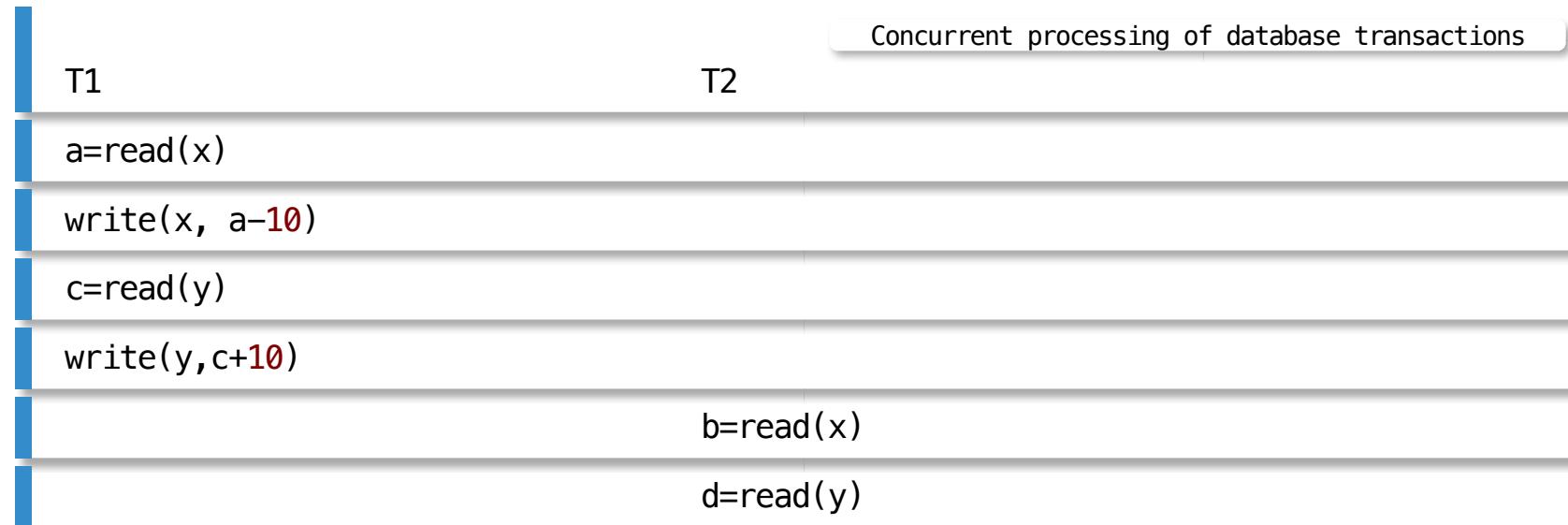
A sample **view serializable** execution of database transactions



The execution of database transactions above is **view serializable** because there exists a **serial execution** of the same transactions such that in both executions the transactions **read the same values and the final states of the database are the same** (see next slide)

# Correctness

A sample serial execution equivalent to a concurrent execution from the previous slide



# Correctness

## A problem with **view serializability**

- Verification that concurrent execution of Database transactions is **view serializable** is **NP-complete**
- It means that it takes too much time to check whether execution of a database operation violates **view serializability** correctness criterion

## A more practical correctness criterion is **conflict serializability**

- Concurrent execution of database transactions is **conflict serializable** if there exists a possible serial execution of the same set of transactions such that in both executions **the order of conflicting operations** is the same

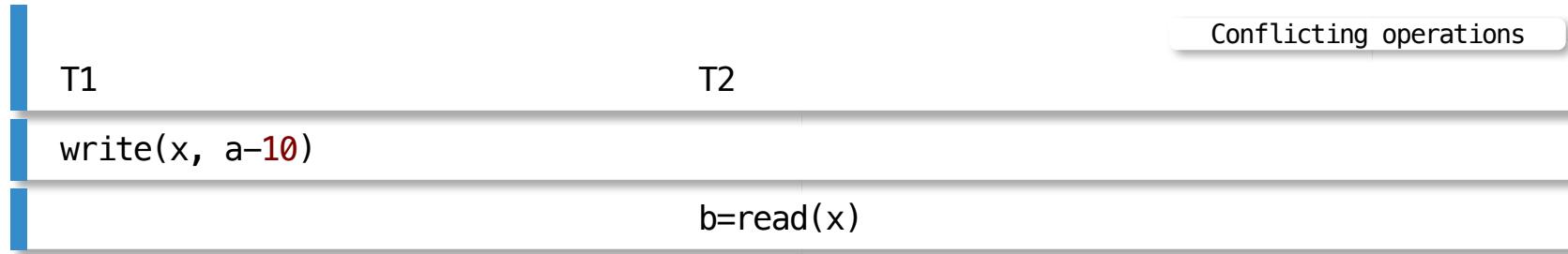
## Which operations are **conflicting operations** ?

Two operations are **conflicting operations** if both **access the same data item** and one or both of them is **write** operation

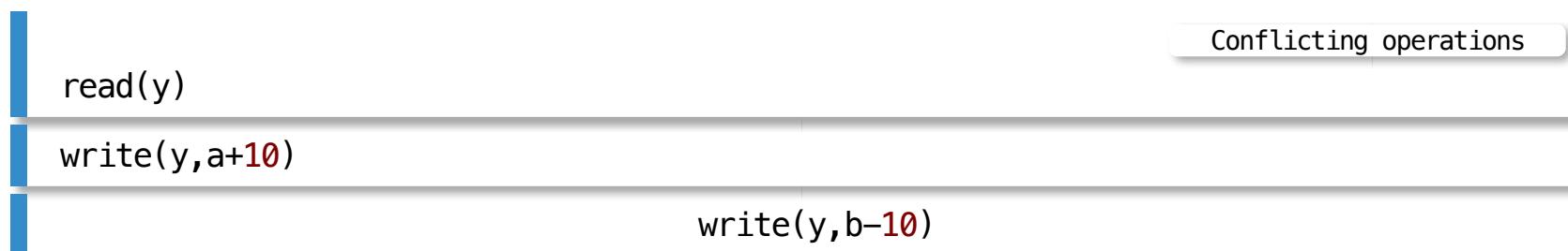
	read	write	Conflicting operations
read	NO	YES	
write	YES	YES	

# Correctness

Conflicting operations in a sample concurrent execution of transactions



The operations `write(x, a-10)` and `b=read(x)` are conflicting operations because both access the same data item `x` and one of them is `write`

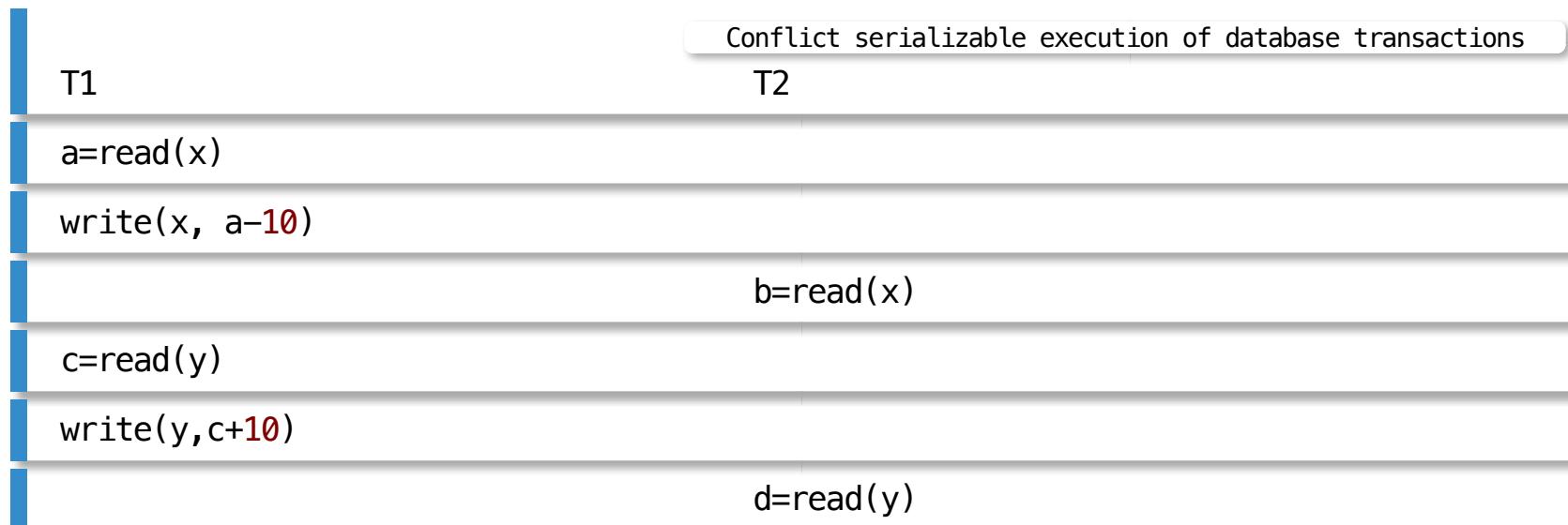


The operations `write(y,a+10)` and `write(y,b-10)` are conflicting operations because both access the same data item `y` and both of them are `write`

# Correctness

Concurrent execution of database transactions is **conflict serializable** if there exists a possible serial execution of the same set of transactions such that in both executions the order of conflicting operations is the same

A sample **conflict serializable** execution of database transactions



Order of conflicting operations: T1 before T2

# Correctness

A sample **not conflict serializable** execution of database transactions

Not conflict serializable execution of database transactions		
T1	T2	
		x: \$100
a=read(x)		x: \$100 a: \$100
	b=read(x)	x: \$100 a: \$100 b: \$100
write(x,a-10)		x: \$90 a: \$100 b: \$100
	write(x,b+20)	x: \$120 a: \$100 b: \$100
	commit	x: \$120 a: \$100 b: \$100
commit		x: \$120 a: \$100 b: \$100

Order of conflicting operations: **T1 before T2** and **T2 before T1** means that it is **impossible to serialize** the concurrent execution of **T1** and **T2**

It means that the concurrent execution of database transactions **T1** and **T2** is **incorrect**

# Introduction to Transaction Processing

## Outline

Correctness

Conflict serializability versus view serializability

Order preserving conflict serializability

Recoverable executions

Cascadeless executions

Strict executions

# Conflict serializability versus view serializability

Every **conflict serializable** execution is **view serializable**

A **view serializable** execution **may not be conflict serializable**

An execution below is **view serializable** because there exists equivalent **serial execution** where each transaction reads and writes the same data items, see next slide

View serializable execution of database transactions		
T1	T2	T3
write(x, 10)		x: 10
	write(x, 20)	x: 20
		write(x, 30) x: 30
	write(y, 10)	x: 30 y: 10
a=read(y)		x: 30 y: 10 a: 10

# Conflict serializability versus view serializability

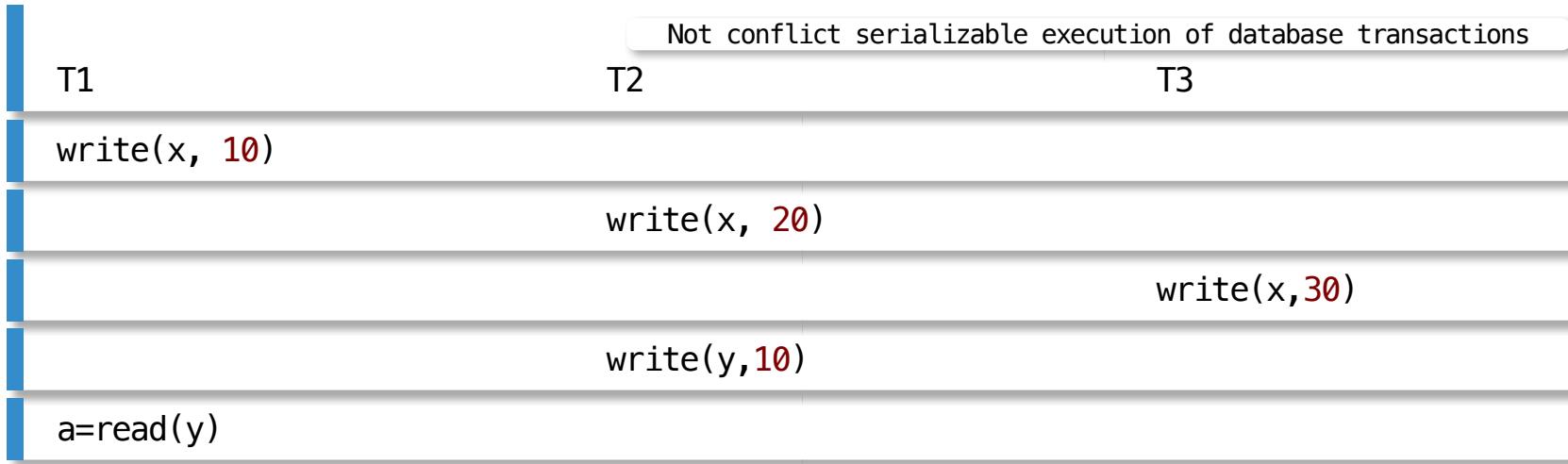
Equivalent **serial execution**

T1	T2	T3	Serial execution of database transactions
	write(x, 20)		x: 20
	write(y, 10)		x: 20 y: 10
write(x, 10)			x: 10 y: 10
a=read(y)			x: 10 y: 10 a: 10
		write(x, 30)	x: 30 y: 10 a: 10

Hence, the original execution is **view serializable**

# Conflict serializability versus view serializability

However, the original execution is **not conflict serializable**



This is because **T1** processes a conflicting operation `write(x, 10)` before **T2** processes `write(x, 20)` and **T2** processes a conflicting operation `write(y, 10)` before **T1** processes `a=read(y)`

Hence, the execution is **not conflict serializable**

# Introduction to Transaction Processing

## Outline

Correctness

Conflict serializability versus view serializability

Order preserving conflict serializability

Recoverable executions

Cascadeless executions

Strict executions

# Order preserving serializability

Concurrent execution of database transactions is **order-preserving conflict serializable** if it is

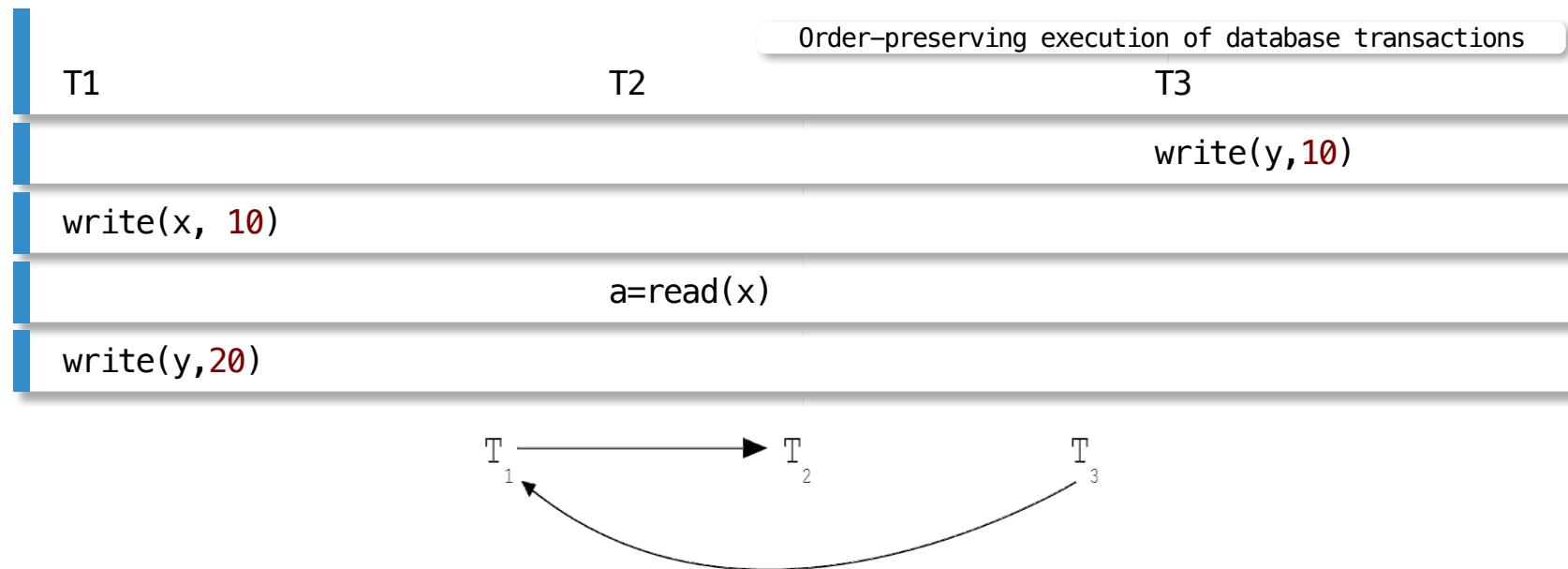
- conflict serializable and
- all non-interleaved transactions have the same order in both original execution and some corresponding serial execution

Every **order-preserving conflict serializable** execution is **conflict serializable**

A **conflict serializable execution** may **not be order-preserving conflict serializable**

# Order preserving serializability

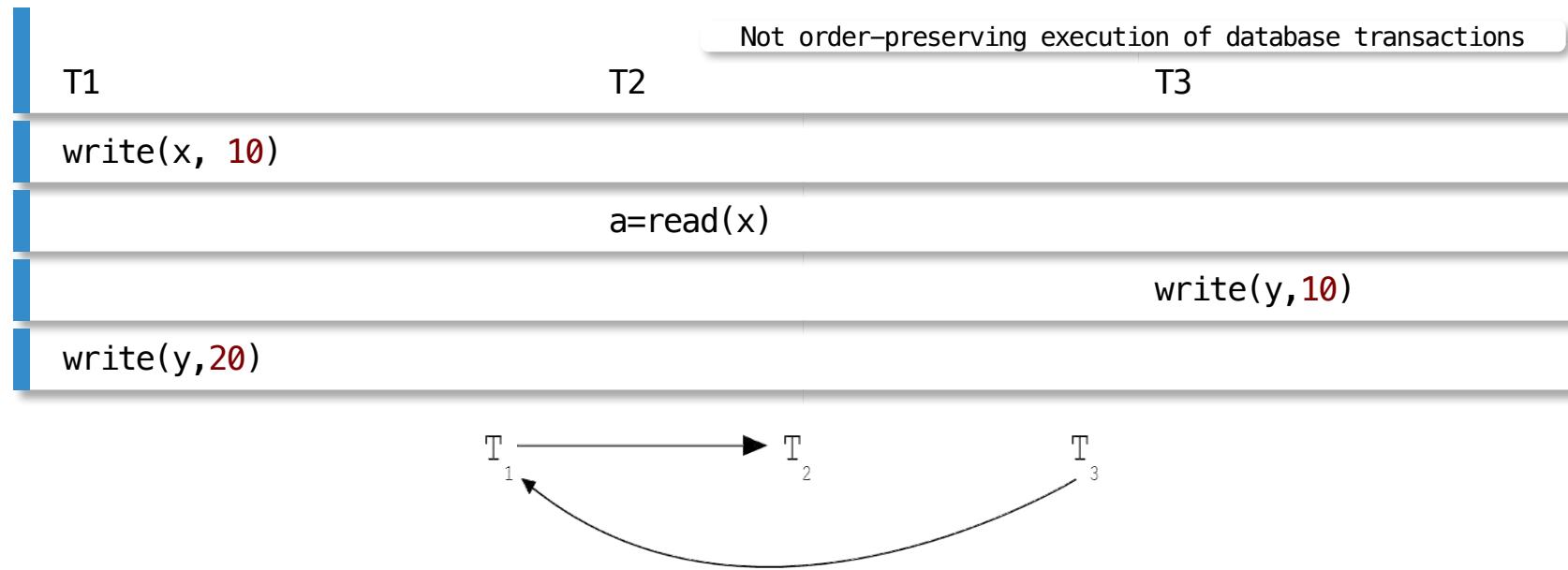
A sample **conflict serializable** and **order-preserving serializable** execution



Order of transactions indicated by their start timestamps is the same as serialization order (**T3** before **T1** and **T1** before **T2**)

# Order preserving serializability

A sample **conflict serializable** and **not order-preserving serializable** execution



The execution is **not order-preserving serializable** because order of transactions indicated by their start timestamps (**T1** before **T2** and **T2** before **T3**) is different from serialization order (**T3** before **T1** and **T1** before **T2**)

# Introduction to Transaction Processing

## Outline

Correctness

Conflict serializability versus view serializability

Order preserving conflict serializability

Recoverable executions

Cascadeless executions

Strict executions

# Recoverable executions

Transaction manager must provide the all-or-nothing property of transactions even in the presence of **various types of failures**

Execution is **recoverable** if every transaction  $T$  that reads a data item written by another transaction  $T'$  commits after  $T'$  is committed

A sample **not recoverable** execution

Not recoverable execution of database transactions		
T1	T2	
		x: \$100
a=read(x)		x: \$100 a: \$100
write(x,a-10)		x: \$90 a: \$100
	b=read(x)	x: \$90 a: \$100 b: \$90
	write(x,b+20)	x: \$110 a: \$100 b: \$90
	commit	x: \$110 a: \$100 b: \$90
abort		x: \$100 a: \$100 b: \$90

# Introduction to Transaction Processing

## Outline

Correctness

Conflict serializability versus view serializability

Order preserving conflict serializability

Recoverable executions

Cascadeless executions

Strict executions

# Cascadeless executions

Execution is **cascadeless** if none of the transactions reads data item written by any other transaction that is not committed or aborted

A sample **cascadeless** execution

Cascadeless execution of database transactions		
T1	T2	
		x: \$100
a=read(x)		x: \$100 a: \$100
write(x,a-10)		x: \$90 a: \$100
	b=read(x) wait	x: \$90 a: \$100
abort		x: \$100 a: \$100
	b=read(x)	x: \$100 a: \$100 b: \$100
	write(x,b+20)	x: \$120 a: \$100 b: \$100

# Cascadeless executions

A sample **not cascadeless** execution

Not cascadeless execution of database transactions		
T1	T2	
		x: \$100
a=read(x)		x: \$100 a: \$100
write(x,a-10)		x: \$90 a: \$100
	b=read(x)	x: \$90 a: \$100 b: \$90
	write(x,b+20)	x: \$110 a: \$100 b: \$90
abort		x: \$100 b: \$90
	forced abort	x: \$100

# Introduction to Transaction Processing

## Outline

Correctness

Conflict serializability versus view serializability

Order preserving conflict serializability

Recoverable executions

Cascadeless executions

Strict executions

# Strict executions

Execution is **strict** if any data item **d** is written by transaction **T** then any other transaction cannot either read or write data item **d** until **T** is committed or aborted

A sample **strict** execution

Strict execution of database transactions		
T1	T2	
		x: \$100
a=read(x)		x: \$100 a: \$100
write(x, a+10)		x: \$110 a: \$100
	write(x, 20) wait	x: \$110 a: \$100
abort		x: \$100
	write(x, 20)	x: \$20
	commit	x: \$20

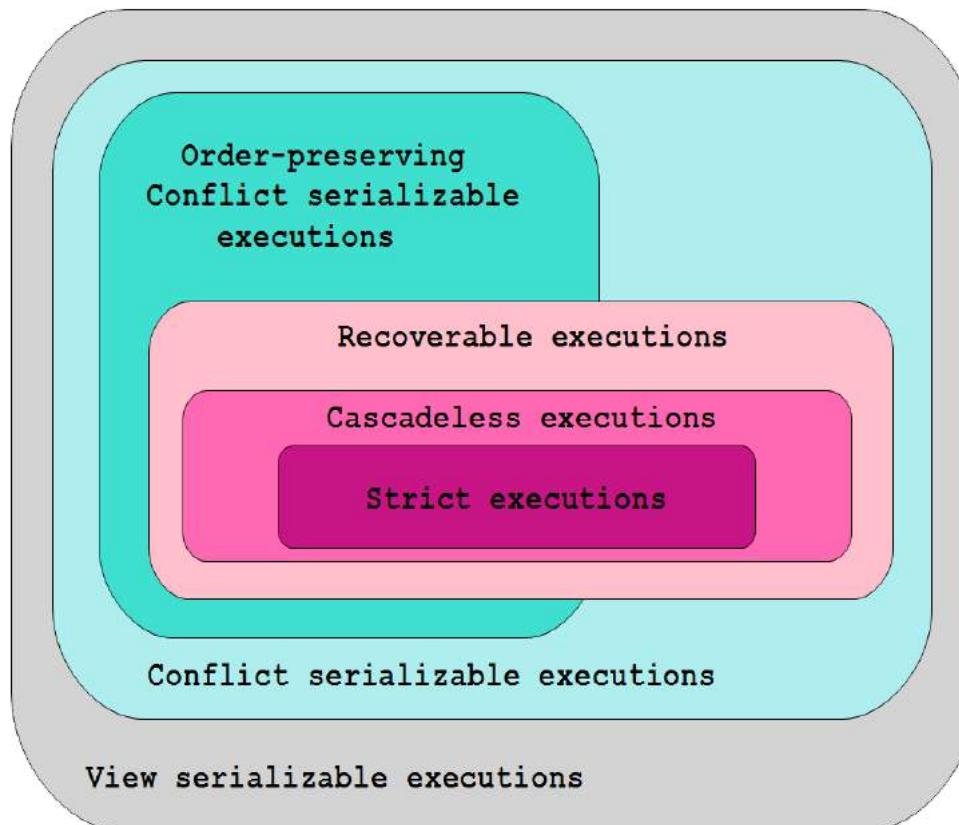
# Strict executions

A sample **not strict** execution

Not strict execution of database transactions		
T1	T2	
		x: \$100
a=read(x)		x: \$100 a: \$100
write(x, a+10)		x: \$110 a: \$100
	write(x,20)	x: \$20 a: \$100
	commit	x: \$20
abort		x \$100

Rollback of T1 destroys committed T2 :`write(x,20)`

# Summary



# References

T. Connolly, C. Begg, *Database Systems, A Practical Approach to Design, Implementation, and Management*, Chapter 22.1 Transaction Support, Chapter 22.2 Concurrency Control, Pearson Education Ltd, 2015

# CSCI235 Database Systems

## Introduction to Transaction Processing (3)

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Introduction to Transaction Processing

## Outline

[Serialization graph](#)

[Serialization graph testing protocol](#)

[Two phase locking protocol \(2PL\)](#)

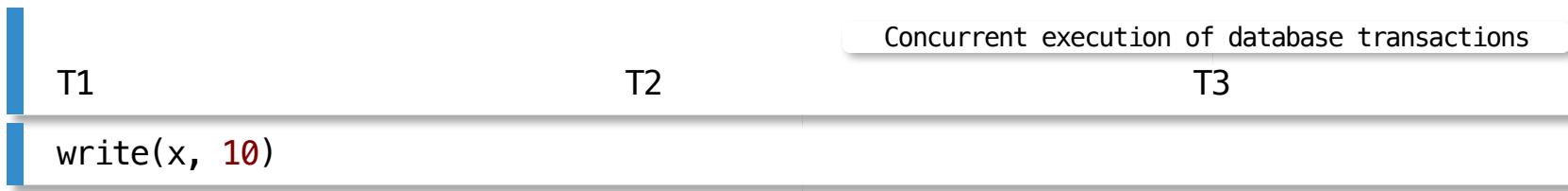
[Timestamp ordering protocol](#)

# Serialization graph

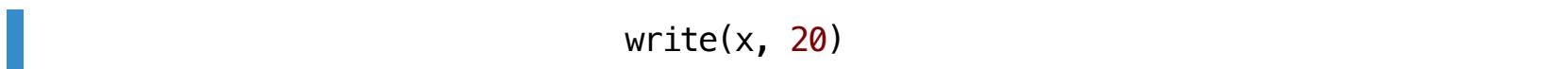
Serialization graph is constructed in the following way

- If a transaction  $T$  participates in a concurrent execution then we add a node labeled with  $T$  to a serialization graph
- If the transactions  $T_i$  and  $T_j$  process conflicting operations such that  $T_i$  processes its operation first then we add an edge directed from  $T_i$  to  $T_j$

Sample construction of a serialization graph



Create a node  $T_1$



Create a node  $T_2$  and add an edge from  $T_1$  to  $T_2$

# Serialization graph

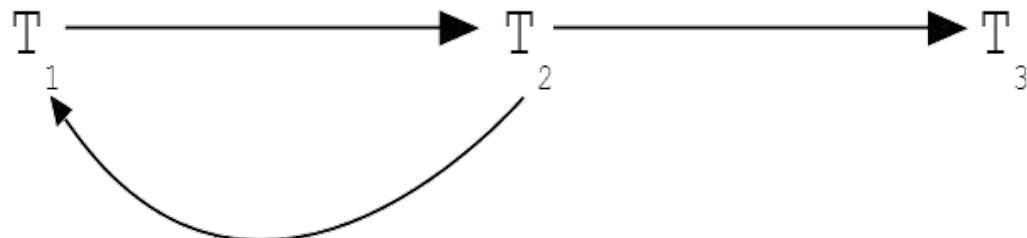
```
write(x,30)
```

Create a node **T3** and add the edges from **T1** to **T3** and from **T2** to **T3**

```
write(y,10)
```

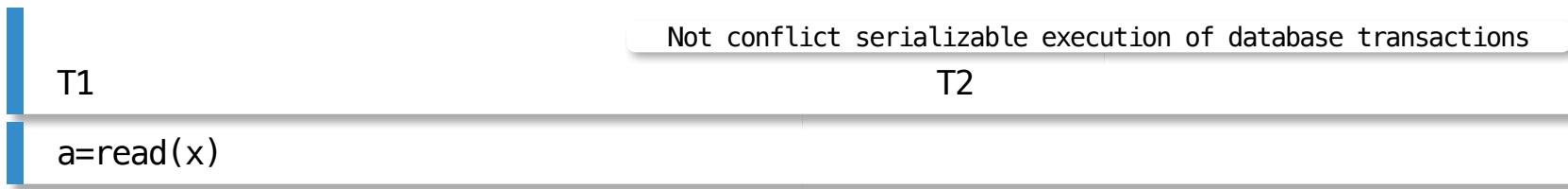
```
a=read(y)
```

Add an edge from **T2** to **T1**



# Serialization graph

A serialization graph of **not conflict serializable** execution of database transactions



Create a node **T1**



Create a node **T2**



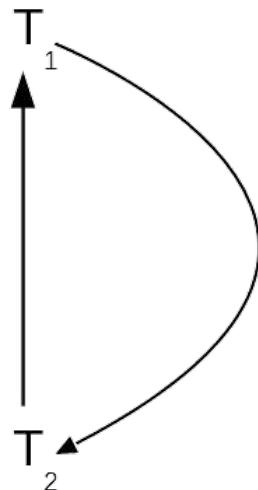
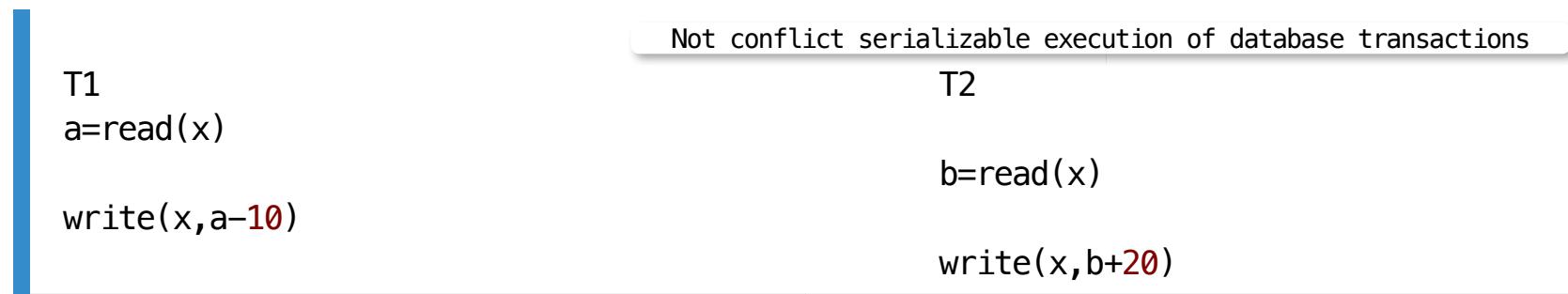
Add an edge from **T2** to **T1**



Add an edge from **T1** to **T2**

# Serialization graph

A serialization graph for **not conflict serializable** execution of database transactions



# Introduction to Transaction Processing

## Outline

[Serialization graph](#)

[Serialization graph testing protocol](#)

[Two phase locking protocol \(2PL\)](#)

[Timestamp ordering protocol](#)

# Serialization graph testing protocol (SGT)

## Principles

- Scheduler maintains and tests **serialization graph**
- If an operation issued by a transaction violates conflict serializability, i.e. if it creates a cycle in **serialization graph** then such transaction is aborted

## Problems

- **Cascading aborts**: if a transaction **T** that created a data item **x** is aborted then all transactions that read a new value of **x** must be aborted
- **Performance**: testing acyclicity of serialization graph has  $O(n^2)$  complexity

# Introduction to Transaction Processing

## Outline

[Serialization graph](#)

[Serialization graph testing protocol](#)

[Two phase locking protocol \(2PL\)](#)

[Timestamp ordering protocol](#)

# Two-phase locking (2PL) protocol

## Principles

- Access to data items is controlled by **shared (read)** and **exclusive (write)** locks
- A transaction can **read** a data item only if a data item is **read** or **write locked** by the same transaction or a data item is **read locked** by another transaction
- A transaction can **write** a data item only if a data item is **write locked** by the same transaction
- **Each transaction must acquire all locks before releasing any lock**

Two-phase locking protocol belongs to a class of **pessimistic protocols**

## Problems

- Deadlocks
- **Unnecessary locks** and **delays** when an execution is conflict serializable

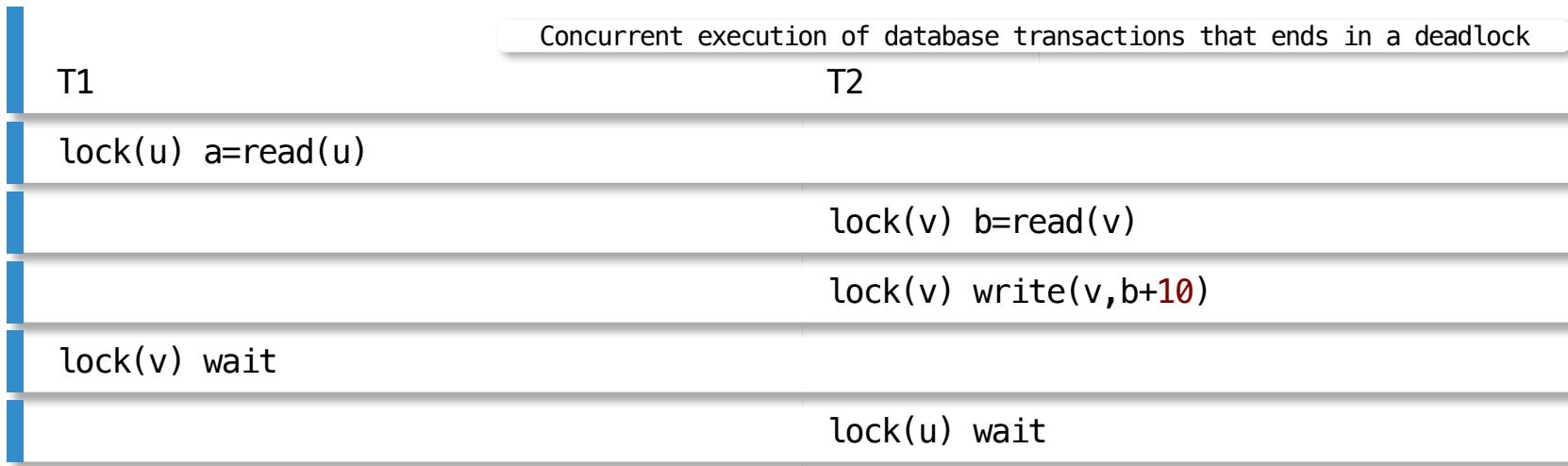
# Two-phase locking (2PL) protocol

A sample execution controlled by **two-phase locking** protocol

Concurrent execution of database transactions controlled by 2PL protocol	
T1	T2
lock(u) a=read(u)	
	lock(v) write(v, <b>10</b> )
write(u,a+2)	
lock(v) wait	
	lock(x) b=read(x)
	unlock(v)
	write(x,b+2)
lock(v)	
	unlock(x)
write(v,a+1)	
unlock(v)	
unlock(u)	

# Two-phase locking (2PL) protocol

A sample execution that ends in a **deadlock**

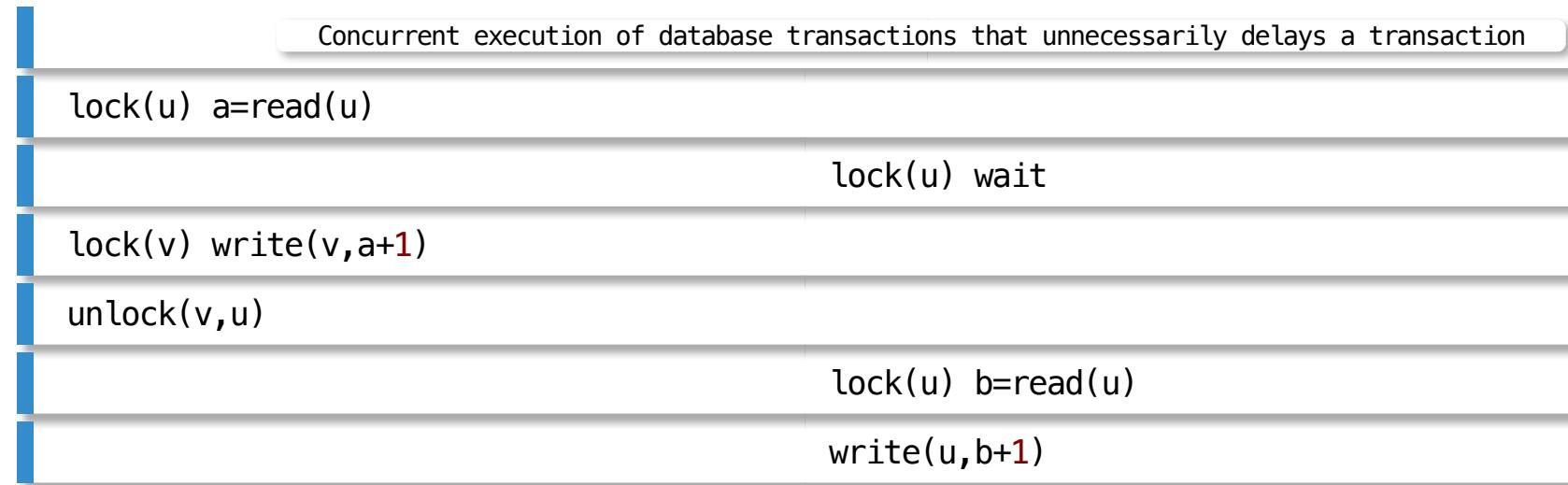


Both transactions are in a **wait** state

In a database system **deadlock** is eliminated through either **wait for graph** or through **timeout**

# Two-phase locking (2PL) protocol

A sample execution that **unnecessarily delays** a transaction



The transactions **T1** and **T2** never get into **not conflict serializable** execution

Therefore, there is no need to delay a transaction **T2**

# Introduction to Transaction Processing

## Outline

[Serialization graph](#)

[Serialization graph testing protocol](#)

[Two phase locking protocol \(2PL\)](#)

[Timestamp ordering protocol](#)

# Timestamp ordering (TO) protocol

## Principles

- Each transaction obtains a **timestamp** at its start point
- Data items are **stamped** each time any transaction accesses data items in a read or write mode
- Access to data items is permitted in an **increasing order** of **timestamps**

Timestamp ordering protocol belongs to a class of **optimistic protocols**

## Problems

- Cascading aborts
- Unnecessary aborts when execution is conflict serializable

# Timestamp ordering (TO) protocol

A sample execution controlled by the timestamp ordering protocol

Concurrent execution of database transactions controlled by TO protocol		
T1	T2	x
timestamp(t1)		
a=read(x)		x:t1
write(x,a-10)		
	timestamp(t2)	
	write(x,10)	x:t1:t2
	b=read(y)	y:t2
write(y,a+1)		y:t2:t1
abort		

# Timestamp ordering (TO) protocol

A sample execution controlled by the **timestamp ordering** protocol with **cascading abort**

Concurrent execution of database transactions controlled by T0 protocol with cascading abort		
T1	T2	x
timestamp(t1)		
a=read(x)		x:t1
write(x,x-10)	timestamp(t2)	
	b=read(x)	x:t1:t2
fail		
	forced abort	

A transaction **T2** is forced to **abort** because it reads a **dirty** data item written by a failed transaction **T1**

# Timestamp ordering (TO) protocol

A sample execution controlled by the timestamp ordering protocol where a transaction is unnecessarily aborted

Concurrent execution of database transactions controlled by T0 protocol with unnecessary abort		
T1	T2	x
	timestamp(t2)	
timestamp(t1)		
a=read(x)		x:t1
write(x,a-10)		
	b=read(x) forced abort	x:t1:t2

A transaction T2 is aborted even the execution is conflict serializable

# References

T. Connolly, C. Begg, *Database Systems, A Practical Approach to Design, Implementation, and Management*, Chapter 22.2 Concurrency Control, Pearson Education Ltd, 2015

# CSCI235 Database Systems

## Transaction Processing in Oracle DBMS

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Transaction Processing in Oracle DBMS

## Outline

Transaction scope

Isolation levels and read consistency levels

Rollback/Undo segments

READ COMMITTED versus SERIALIZABLE levels

Locking

READ COMMITTED isolation level

SERIALIZABLE isolation level

# Transaction scope

Transaction starts from the first executable SQL statement after connection, **COMMIT**, or **ROLLBACK** statement

Transaction ends with either **COMMIT**, **ROLLBACK**, or DDL statement  
**CREATE**, **DROP**, **ALTER**

Transaction also ends with disconnection (**auto-commit** or **auto-rollback** depending on default or set up parameters) or process failure followed by automatic **ROLLBACK** statement

# Transaction Processing in Oracle DBMS

## Outline

[Transaction scope](#)

[Isolation levels and read consistency levels](#)

[Rollback/Undo segments](#)

[READ COMMITTED versus SERIALIZABLE levels](#)

[Locking](#)

[READ COMMITTED isolation level](#)

[SERIALIZABLE isolation level](#)

# Isolation levels and read consistency levels

Oracle DBMS implements three **isolation levels**

**READ COMMITTED** - a transaction may exhibit:

- non-repeatable read phenomenon,
- phantom phenomenon

**SERIALIZABLE** - a transaction may exhibit:

- none of the phenomena

**READ ONLY** - a transaction consists only of **read** operations

At **READ COMMITTED** isolation level all data read by a query (**SELECT** statement) come from a single point in time (**statement-level read consistency**)

At **SERIALIZABLE** isolation level all queries (**SELECT** statements) in a transaction read data that come from a single point in time (**transaction-level read consistency**)

# Transaction Processing in Oracle DBMS

## Outline

[Transaction scope](#)

[Isolation levels and read consistency levels](#)

[Rollback/Undo segments](#)

[READ COMMITTED versus SERIALIZABLE levels](#)

[Locking](#)

[READ COMMITTED isolation level](#)

[SERIALIZABLE isolation level](#)

# Rollback/undo segments

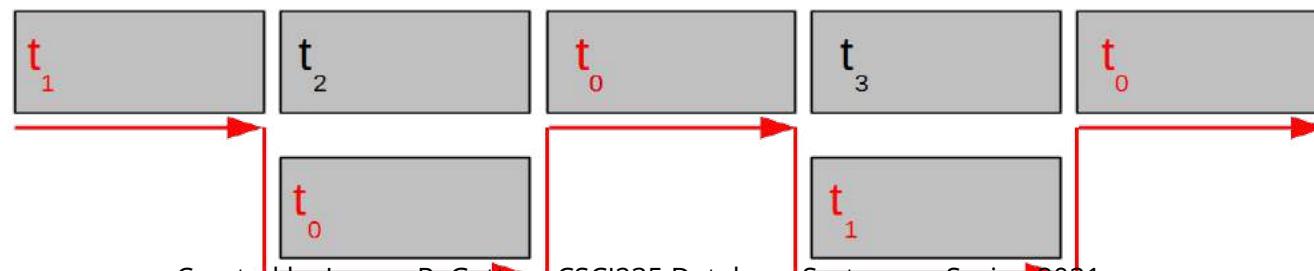
Rollback/undo segments consist of data blocks that contain the old values of data that have been changed by the uncommitted or recently committed transactions

Each time a row in a data block is changed a new version of the block together with a timestamp is added to rollback/undo segment

A new version of a data block obtains a timestamp higher than the previous version

A data block may have many versions updated in the different moments in time

See below the data blocks read by a transaction  $T$  with a timestamp  $t_i$  such that  $t_2, t_3 > t_i > t_0, t_1$



# Transaction Processing in Oracle DBMS

## Outline

[Transaction scope](#)

[Isolation levels and read consistency levels](#)

[Rollback/Undo segments](#)

[READ COMMITTED versus SERIALIZABLE levels](#)

[Locking](#)

[READ COMMITTED isolation level](#)

[SERIALIZABLE isolation level](#)

# READ COMMITTED versus SERIALIZABLE levels

At **READ COMMITTED** isolation level each query executes with respect to its own materialized view time, thereby permitting nonrepeatable reads and phantoms for multiple executions of a query

**READ COMMITTED** isolation level is recommended when few transactions are likely to conflict

If a transaction **T** running at **SERIALIZABLE** isolation level tries to update or delete data modified by a transaction that commits after the serializable transaction **T** began then the system aborts transaction **T**

If a serializable transaction fails then it is possible to:

- commit the work executed to that point,
- execute additional (but different) statements,
- rollback the entire transaction

# Transaction Processing in Oracle DBMS

## Outline

Transaction scope

Isolation levels and read consistency levels

Rollback/Undo segments

READ COMMITTED versus SERIALIZABLE levels

Locking

READ COMMITTED isolation level

SERIALIZABLE isolation level

# Locking

**Locking** is performed automatically by the system

It is also possible to lock data items manually

There are two types of locks:

- **shared** (read locks),
- **exclusive** (write locks)

A transaction holds **exclusive row locks** for all rows **inserted, updated, or deleted** within the transaction

The system releases all locks acquired by a transaction when the transaction either **commits** or **rollbacks**

The system automatically converts a lock of **lower restrictiveness** to one of **higher restrictiveness** when it is appropriate

For example, **SELECT** statement with **FOR UPDATE** clause initially locks the rows in **shared mode**, then when **UPDATE** is performed the locks are upgraded to **exclusive locks**

# Locking

Both **READ COMMITTED** and **SERIALIZABLE** transactions use row-level locking to ensure database consistency

A transaction must wait if it tries to change a row updated by an uncommitted transaction

If a transaction rollbacks then the waiting transactions regardless of its isolation mode can proceed to change the previously locked row

If a transaction commits than any other transaction at **READ COMMITTED** level waiting for a locked row can proceed to change the previously locked row

If a transaction commits than any other transaction at **SERIALIZABLE** level waiting for a locked row fails with the error "Cannot serialize access", because the other transaction has committed a change that was made since the serializable transaction began

# Locking

A **deadlock** occurs when two or more users are waiting for data locked by each other

The system automatically detects a **deadlock** and rolls back one of the statements involved in the deadlock

It is possible to perform **manual locking** of entire relational table with **LOCK TABLE** statement

# Transaction Processing in Oracle DBMS

## Outline

[Transaction scope](#)

[Isolation levels and read consistency levels](#)

[Rollback/Undo segments](#)

[READ COMMITTED versus SERIALIZABLE levels](#)

[Locking](#)

[READ COMMITTED isolation level](#)

[SERIALIZABLE isolation level](#)

# READ COMMITTED isolation level

Setting **READ COMMITTED isolation level** is performed at the beginning of transaction with the following statement

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

At **READ COMMITTED** isolation level each query executes with the respect to its own materialized view time, thereby permitting **nonrepeatable reads** and **phantoms** for multiple processing of the same query

**READ COMMITTED** isolation level is recommended when few transactions are likely to conflict

**READ COMMITTED** is a default isolation level

# READ COMMITTED isolation level

Sample concurrent processing of database transactions at **READ COMMITTED** isolation level that **does not corrupt** a database

Concurrent processing of database transactions at READ COMMITTED isolation level	
Transaction 1	Transaction 2
	<pre>SELECT budget FROM DEPARTMENT WHERE name = 'Sales';</pre>
	2000
<pre>UPDATE DEPARTMENT SET budget = budget + 1000 WHERE NAME = 'Sales';</pre>	
	<pre>SELECT budget FROM DEPARTMENT WHERE name = 'Sales';</pre>
	2000

Transaction 2 cannot read uncommitted modifications

# READ COMMITTED isolation level

Sample concurrent processing of database transactions at **READ COMMITTED** isolation level that **does not corrupt** a database

Concurrent processing of database transactions at READ COMMITTED isolation level	
Transaction 1	Transaction 2
	<code>SELECT budget FROM Department WHERE name = 'Sales';</code>
	<code>2000</code>
<code>UPDATE DEPARTMENT SET budget = budget + 1000 WHERE name = 'Sales';</code>	
<code>COMMIT;</code>	
	<code>SELECT budget FROM Department WHERE name = 'Sales';</code>
	<code>3000</code>

Transaction 2 can only read committed modifications

# READ COMMITTED isolation level

Sample concurrent processing of database transactions at **READ COMMITTED** isolation level that **does not corrupt** a database

## Concurrent processing of database transactions at READ COMMITTED isolation level

Transaction 1

Transaction 2

```
SELECT budget  
FROM DEPARTMENT  
WHERE name = 'Sales';
```

3000

```
UPDATE DEPARTMENT  
SET budget = budget + 10  
WHERE name = 'Sales';
```

```
UPDATE DEPARTMENT  
SET budget = budget + 1000  
WHERE name = 'Sales';
```

Wait

COMMIT;

```
SELECT budget  
FROM DEPARTMENT  
WHERE name = 'Sales'
```

TOP

4010

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

18/26

# READ COMMITTED isolation level

Transaction 1	Transaction 2	Processing at READ COMMITTED isolation level
<pre>SELECT budget FROM DEPARTMENT WHERE name = 'Sales';</pre>		
3000		
	<pre>UPDATE DEPARTMENT SET budget = budget + 10 WHERE name = 'Sales';</pre>	
<pre>UPDATE DEPARTMENT SET budget = budget + 1000 WHERE name = 'Sales';</pre>		
Wait		
	<pre>COMMIT;</pre>	
<pre>SELECT budget FROM DEPARTMENT WHERE name = 'Sales'</pre>		
4010		

Transaction 1 must wait until Transaction 2 either commits or rolls back its update

[TOP](#)

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

19/26

# READ COMMITTED isolation level

Sample concurrent processing of database transactions at **READ COMMITTED** isolation level that **does not corrupt** a database

Transaction 1	Transaction 2	Processing at READ COMMITTED isolation level
<pre>SSELECT budget FROM DEPARTMENT WHERE name = 'Sales'</pre>		
<pre>3000</pre>		
	<pre>UPDATE DEPARTMENT SET budget = budget + 10 WHERE name = 'Sales'</pre>	
<pre>UPDATE DEPARTMENT SET budget = budget + 1000 WHERE name = 'Sales';</pre>		
<pre>Wait</pre>		
	<pre>ROLLBACK;</pre>	
<pre>SELECT budget FROM DEPARTMENT WHERE name = 'Sales'</pre>		
<pre>4000</pre>		

# READ COMMITTED isolation level

Transaction 1	Transaction 2	Processing at READ COMMITTED isolation level
<pre>SSELECT budget FROM DEPARTMENT WHERE name = 'Sales'</pre>		
3000		
	<pre>UPDATE DEPARTMENT SET budget = budget + 10 WHERE name = 'Sales'</pre>	
<pre>UPDATE DEPARTMENT SET budget = budget + 1000 WHERE name = 'Sales';</pre>		
Wait		
	<pre>ROLLBACK;</pre>	
<pre>SELECT budget FROM DEPARTMENT WHERE name = 'Sales'</pre>		
4000		

Transaction 1 must wait until Transaction 2 either commits or rolls back its update

[TOP](#)

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

21/26

# READ COMMITTED isolation level

A sample processing of database transactions at **READ COMMITTED** isolation level that **corrupts** a database

Processing at READ COMMITTED isolation level	
Transaction 1	Transaction 2
<pre>UPDATE DEPARTMENT SET budget = (SELECT budget                FROM DEPARTMENT               WHERE name = 'Sales') WHERE name = 'Finance';</pre>	
	<pre>UPDATE DEPARTMENT SET budget = 500 WHERE name = 'Sales'; COMMIT;</pre>
<pre>UPDATE DEPARTMENT SET budget = budget + (SELECT budget                            FROM DEPARTMENT                           WHERE name = 'Sales') WHERE name = 'Finance'; COMMIT;</pre>	

**Transaction 1** corrupts a database, because a budget of department **Finance** is not equal to  $2 * \text{budget of department Sales}$

# Transaction Processing in Oracle DBMS

## Outline

[Transaction scope](#)

[Isolation levels and read consistency levels](#)

[Rollback/Undo segments](#)

[READ COMMITTED versus SERIALIZABLE levels](#)

[Locking](#)

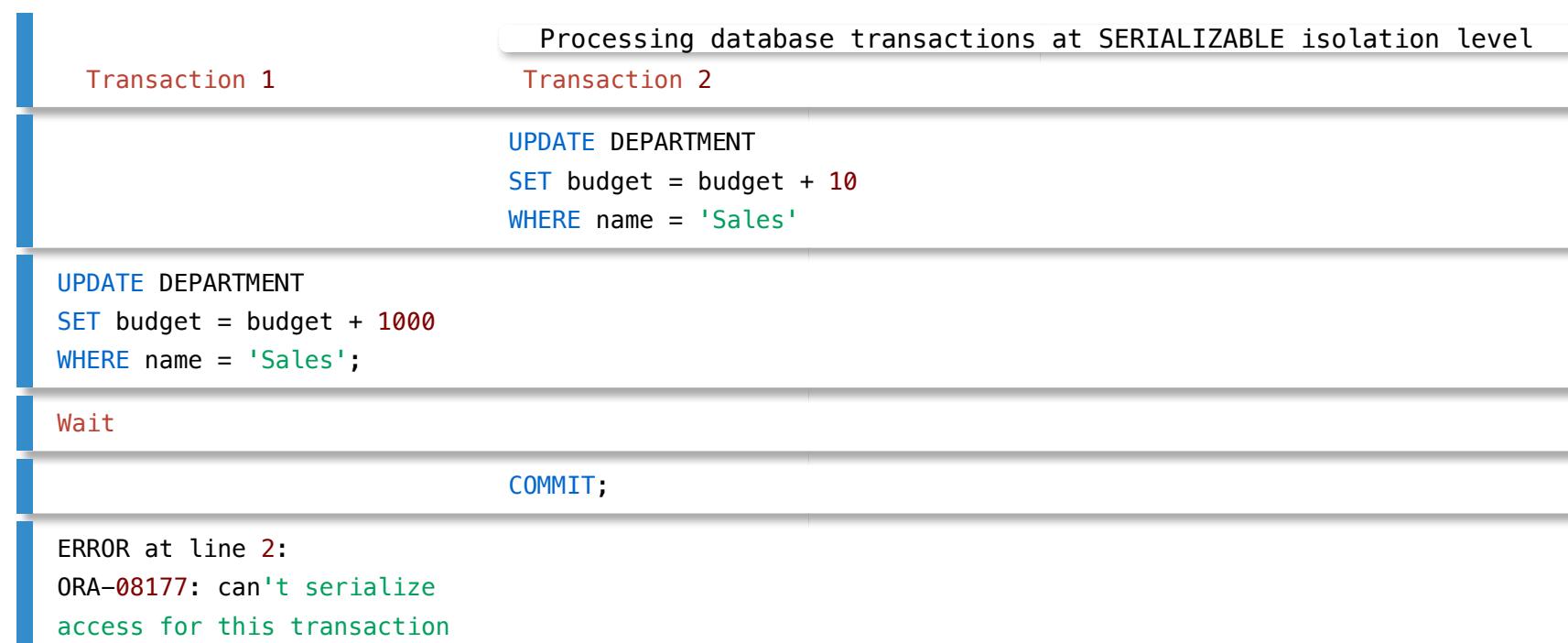
[READ COMMITTED isolation level](#)

[SERIALIZABLE isolation level](#)

# SERIALIZABLE isolation level

If a transaction **T** running at **SERIALIZABLE** isolation level tries to update or delete data modified by a transaction that commits after the serializable transaction **T** began then the system aborts transaction **T**

A sample processing of database transactions at **SERIALIZABLE** isolation level that **fails** one of the transactions



# SERIALIZABLE isolation level

A sample processing of database transactions at **SERIALIZABLE** isolation level that ends **successfully** for both transactions

Processing database transactions at SERIALIZABLE isolation level	
Transaction 1	Transaction 2
	<code>SELECT budget FROM Department WHERE name = 'Sales'</code>
	2000
<code>UPDATE DEPARTMENT SET budget = budget + 1000 WHERE name = 'Sales';</code>	
<code>COMMIT</code>	
	<code>SELECT budget FROM Department WHERE name = 'Sales'</code>
	2000

Transaction 1 creates a new version of a row for **Sales** department

# References

T. Connolly, C. Begg, *Database Systems, A Practical Approach to Design, Implementation, and Management*, Chapter 22.5 Concurrency Control and Recovery in Oracle, Pearson Education Ltd, 2015

# CSCI235 Database Systems

# Distributed Relational Database Systems

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Distributed Relational Database Systems

## Outline

Distributed database ? What is it ?

Centralized database schema model

Mainframe application architecture

Client-server architecture

Distributed client-server architecture

Web based architecture

Distributed Web based architecture

Distributed database schema architecture

Transparency

Autonomy

Advantages and disadvantages of DDBS

Distributed database system Oracle 19c

TOP

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

2/31

# Distributed database ? What is it ?

**Distributed database system** (DDBS) is a collection of multiple logically related databases distributed over a computer network

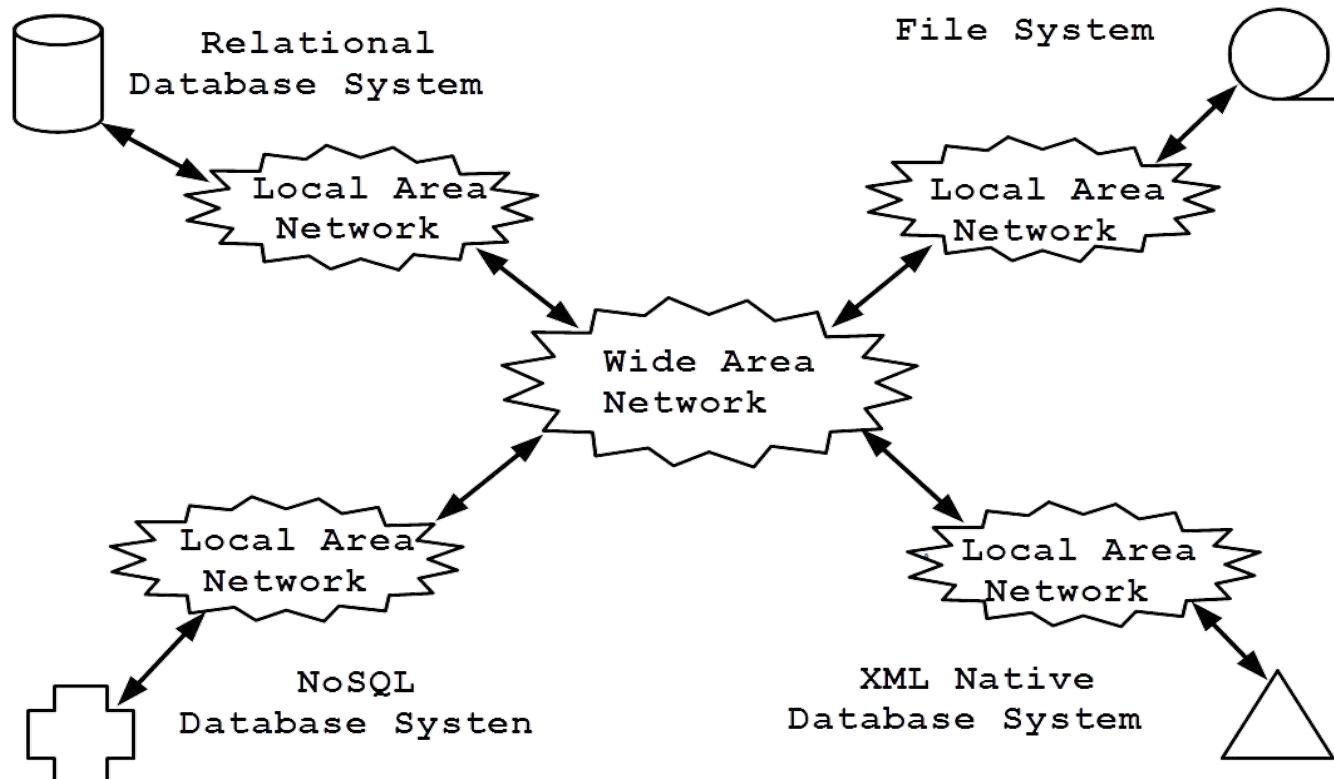
**Distributed database management system** (DDBMS) is a collection of database systems together with software providing a required set of operations on data and management features

**Homogeneous DDBS** is a collection of identical database systems distributed over a computer network, e.g. a collection of Oracle database systems

**Heterogeneous DDBS** is a collection of different database systems distributed over a computer network, e.g. a collection of Oracle + MySQL + DB/2 + MongoDB + XML native database systems + Excel spreadsheets + ... , systems

# Distributed database ? What is it ?

A sample organization of **heterogeneous distributed database system**



# Distributed Relational Database Systems

## Outline

Distributed database ? What is it ?

Centralized database schema model

Mainframe application architecture

Client-server architecture

Distributed client-server architecture

Web based architecture

Distributed Web based architecture

Distributed database schema architecture

Transparency

Autonomy

Advantages and disadvantages of DDBS

Distributed database system Oracle 19c

TOP

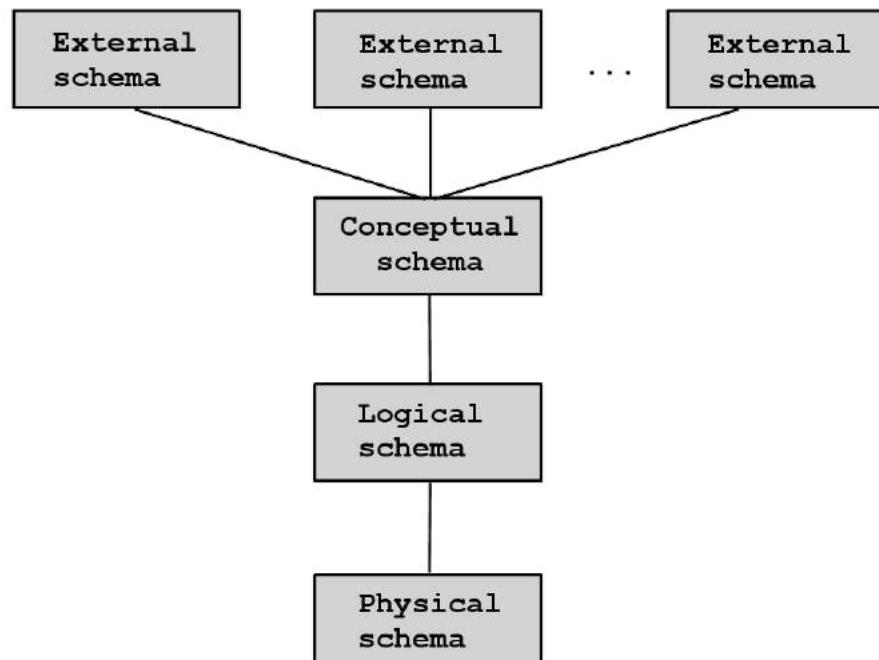
Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

5/31

# Centralized database schema model

In a **centralized database schema model** the users are provided their personal **external schemas** (views) of data

The **external schemas** are integrated into a **single conceptual schema** later on transformed into a **logical schema** and implemented as **physical schema**



# Distributed Relational Database Systems

## Outline

Distributed database ? What is it ?

Centralized database schema model

Mainframe application architecture

Client-server architecture

Distributed client-server architecture

Web based architecture

Distributed Web based architecture

Distributed database schema architecture

Transparency

Autonomy

Advantages and disadvantages of DDBS

Distributed database system Oracle 19c

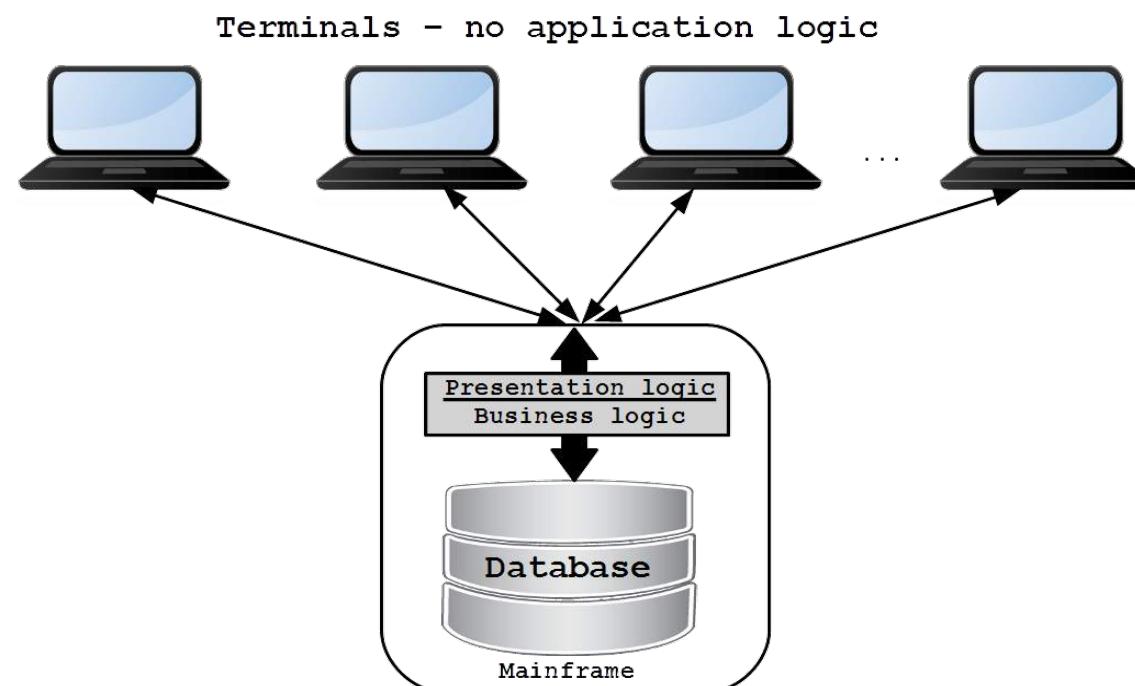
TOP

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

7/31

# Mainframe application architecture

In **mainframe application architecture** "dumb" terminals are connected to a **single database server**



# Distributed Relational Database Systems

## Outline

Distributed database ? What is it ?

Centralized database schema model

Mainframe application architecture

Client-server architecture

Distributed client-server architecture

Web based architecture

Distributed Web based architecture

Distributed database schema architecture

Transparency

Autonomy

Advantages and disadvantages of DDBS

Distributed database system Oracle 19c

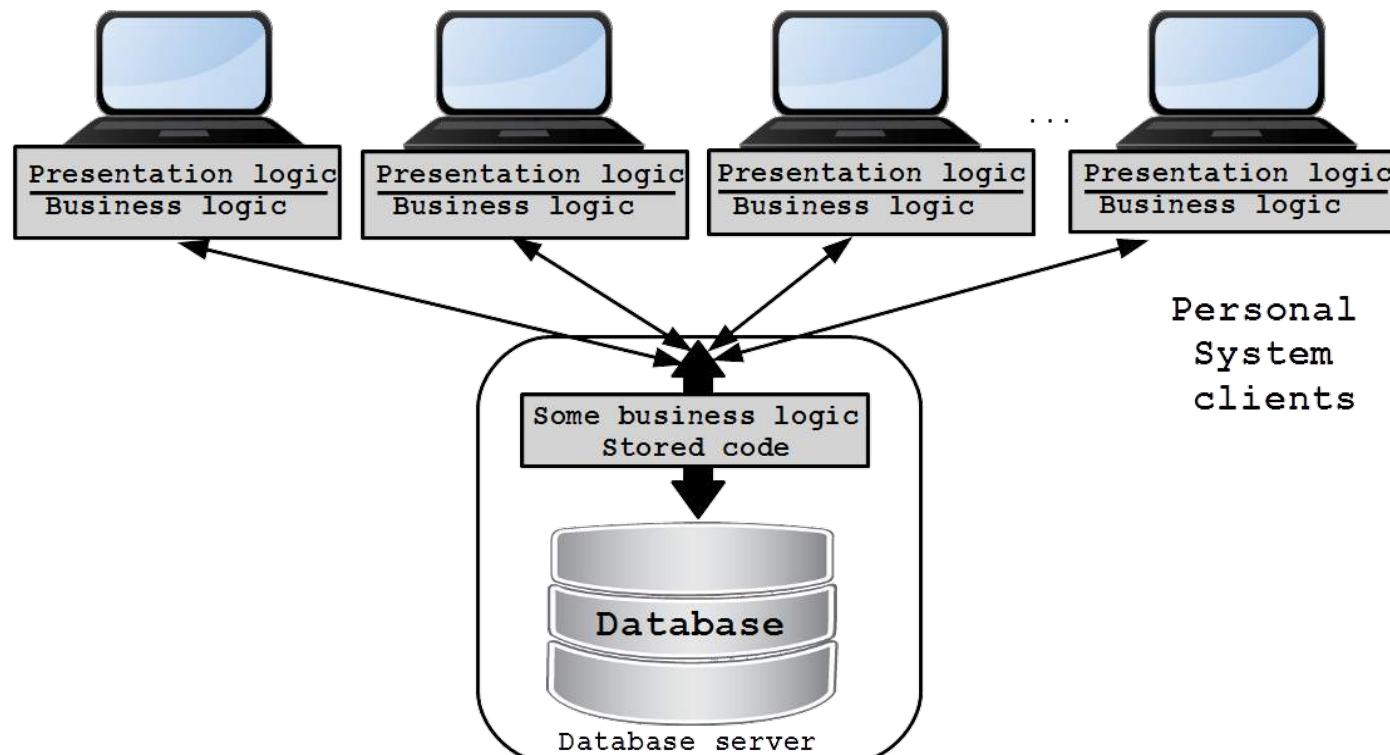
TOP

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

9/31

# Client-server architecture

In **client-server architecture** personal systems communicate with a single database server



# Distributed Relational Database Systems

## Outline

Distributed database ? What is it ?

Centralized database schema model

Mainframe application architecture

Client-server architecture

Distributed client-server architecture

Web based architecture

Distributed Web based architecture

Distributed database schema architecture

Transparency

Autonomy

Advantages and disadvantages of DDBS

Distributed database system Oracle 19c

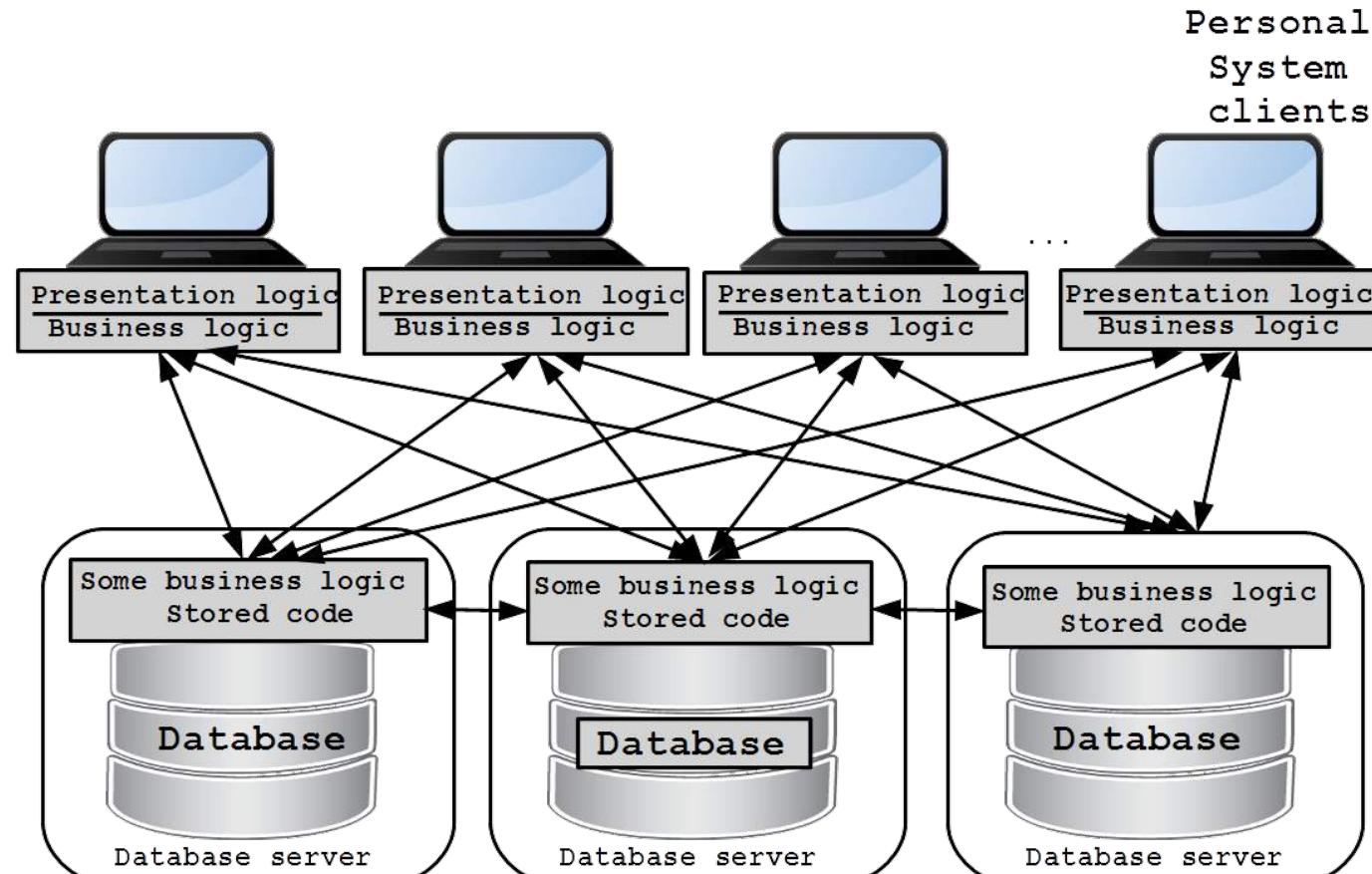
TOP

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

11/31

# Distributed client-server architecture

In **distributed client-server architecture** personal systems communicate with many database servers



# Distributed Relational Database Systems

## Outline

Distributed database ? What is it ?

Centralized database schema model

Mainframe application architecture

Client-server architecture

Distributed client-server architecture

Web based architecture

Distributed Web based architecture

Distributed database schema architecture

Transparency

Autonomy

Advantages and disadvantages of DDBS

Distributed database system Oracle 19c

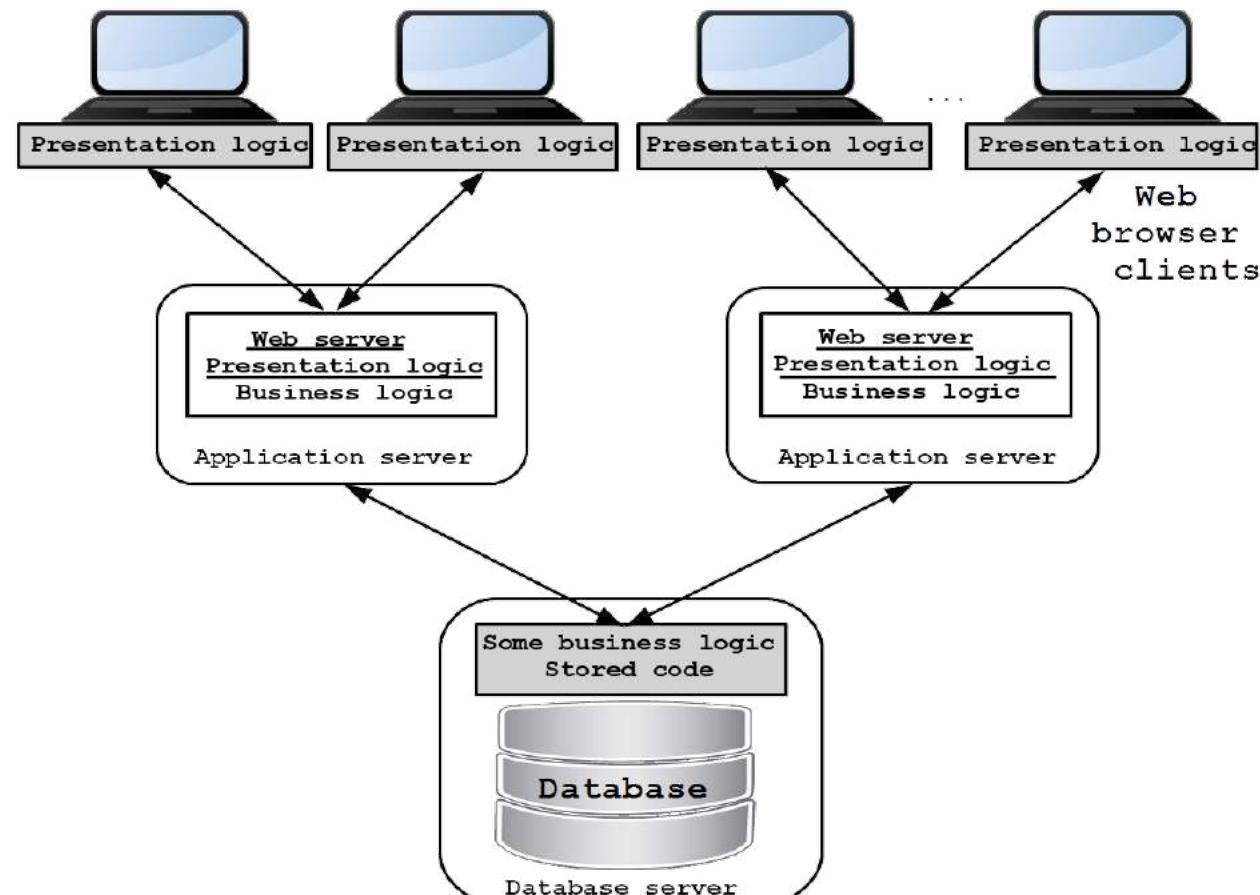
TOP

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

13/31

# Web based architecture

In **Web based architecture** personal systems communicate with the Web servers that communicate with a **single database server**



# Distributed Relational Database Systems

## Outline

Distributed database ? What is it ?

Centralized database schema model

Mainframe application architecture

Client-server architecture

Distributed client-server architecture

Web based architecture

Distributed Web based architecture

Distributed database schema architecture

Transparency

Autonomy

Advantages and disadvantages of DDBS

Distributed database system Oracle 19c

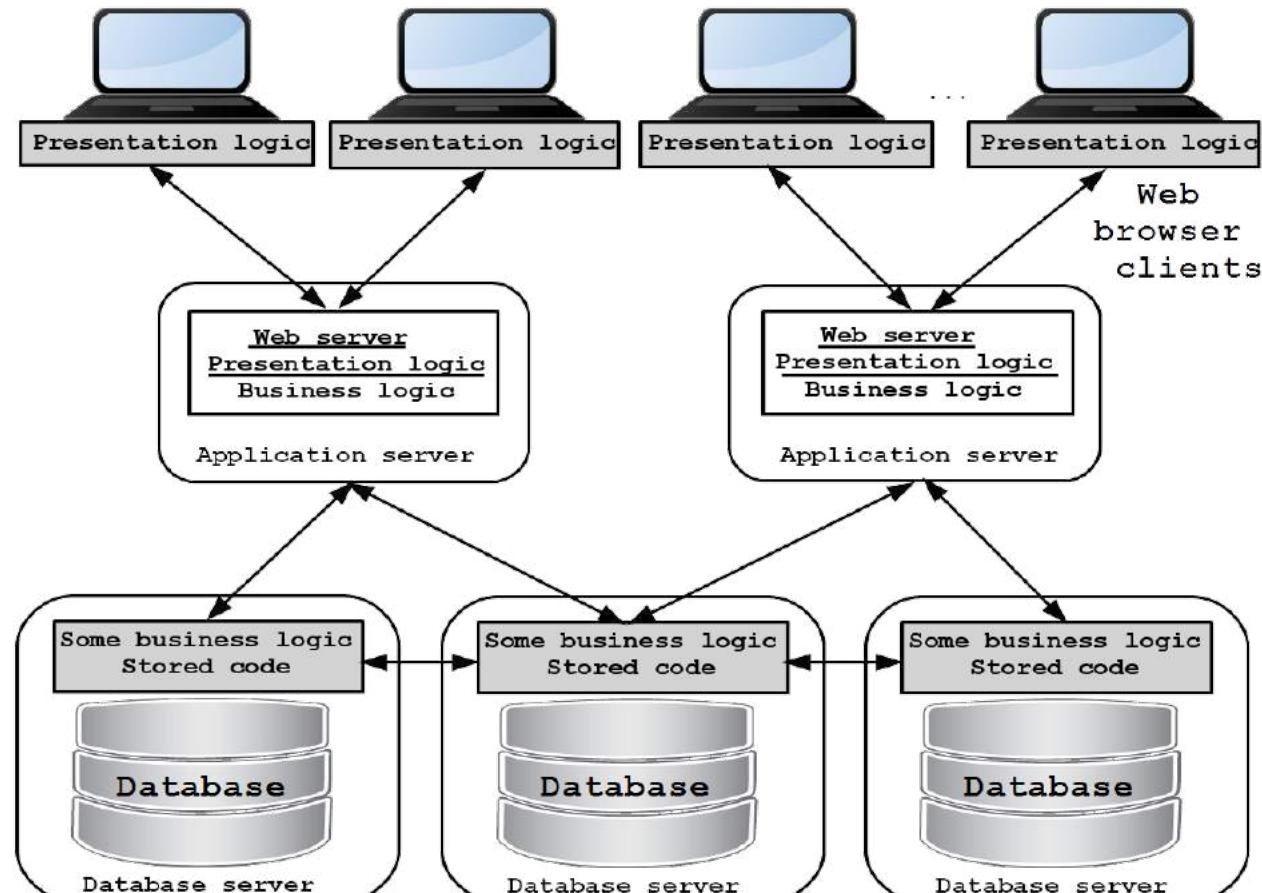
TOP

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

15/31

# Distributed Web based architecture

In **Distributed Web based architecture** personal systems communicate with the Web servers that communicate with **many database servers**



# Distributed Relational Database Systems

## Outline

Distributed database ? What is it ?

Centralized database schema model

Mainframe application architecture

Client-server architecture

Distributed client-server architecture

Web based architecture

Distributed Web based architecture

Distributed database schema architecture

Transparency

Autonomy

Advantages and disadvantages of DDBS

Distributed database system Oracle 19c

TOP

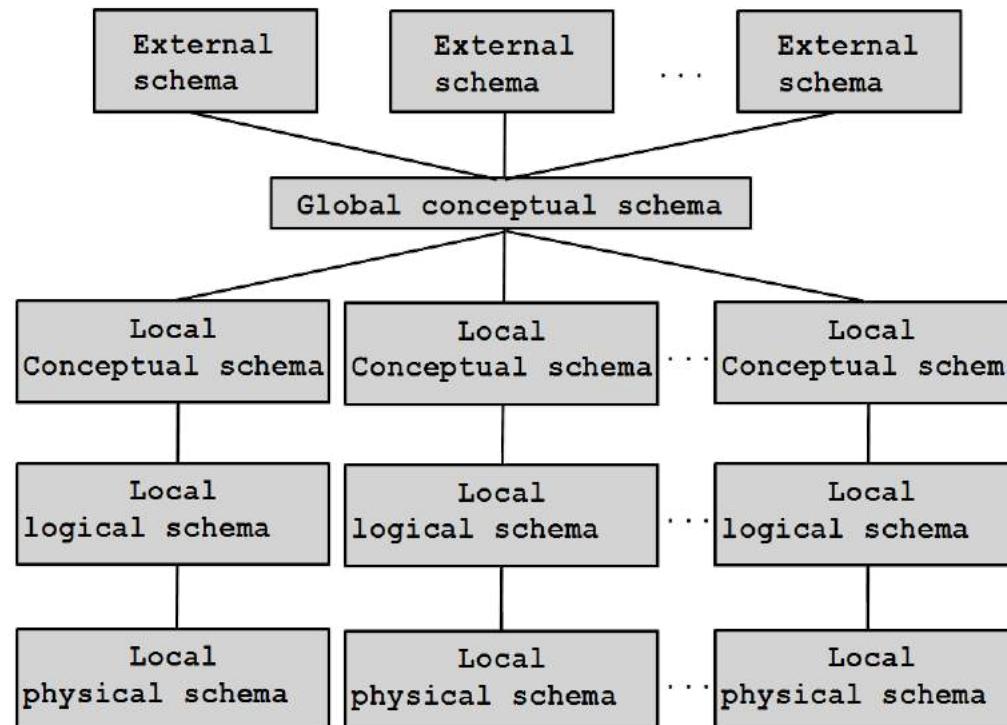
Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

17/31

# Distributed database schema architecture

In **distributed database schema architecture** a **global conceptual schema** hides distribution from the users

The users can see a **distributed database system** as a **single monolithic database system**



# Distributed Relational Database Systems

## Outline

Distributed database ? What is it ?

Centralized database schema model

Mainframe application architecture

Client-server architecture

Distributed client-server architecture

Web based architecture

Distributed Web based architecture

Distributed database schema architecture

Transparency

Autonomy

Advantages and disadvantages of DDBS

Distributed database system Oracle 19c

TOP

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

19/31

# Transparency

Transparency means hiding information from end users

Data organization (distribution or network) transparency means hiding network related information and data placement information; it is either location or naming transparency

Naming transparency allows for global naming of data objects

Location transparency allows the operations to be independent on the locations of data objects

Replication transparency means that users are unaware of the existence of multiple copies of the same data objects

Fragmentation transparency means that users are unaware of data fragmentation over many sites; it includes vertical and horizontal fragmentation

# Transparency

**Design transparency** means that users are unaware of how distributed database was designed

**Execution transparency** means that users are unaware of how database transactions are processed

# Distributed Relational Database Systems

## Outline

Distributed database ? What is it ?

Centralized database schema model

Mainframe application architecture

Client-server architecture

Distributed client-server architecture

Web based architecture

Distributed Web based architecture

Distributed database schema architecture

Transparency

Autonomy

Advantages and disadvantages of DDBS

Distributed database system Oracle 19c

TOP

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

22/31

# Autonomy

**Autonomy** determines a level of independence of individual nodes in distributed database system

**High degree autonomy** is required for flexibility and customized maintenance of distributed database system

**Design autonomy** means a level of independence of data model usage and transaction management technique between the nodes

**Communication autonomy** means a level of independence to which a node can share information with other nodes

**Execution autonomy** means a level of independence to which users act as they please

# Distributed Relational Database Systems

## Outline

Distributed database ? What is it ?

Centralized database schema model

Mainframe application architecture

Client-server architecture

Distributed client-server architecture

Web based architecture

Distributed Web based architecture

Distributed database schema architecture

Transparency

Autonomy

Advantages and disadvantages of DDBS

Distributed database system Oracle 19c

TOP

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

24/31

# Advantages and disadvantages of DDBS

Higher level of **reliability** and **availability**

Improved **ease** and **flexibility** of application development

Improved **performance**

Easier **expansion**

# Advantages and disadvantages of DDBS

- Keeping track of data distribution
- Distributed query processing
- Distributed transaction management
- Replicated data management
- Distributed database recovery
- Security
- Distributed directory (catalog) management

# Distributed Relational Database Systems

## Outline

Distributed database ? What is it ?

Centralized database schema model

Mainframe application architecture

Client-server architecture

Distributed client-server architecture

Web based architecture

Distributed Web based architecture

Distributed database schema architecture

Transparency

Autonomy

Advantages and disadvantages of DDBS

Distributed database system Oracle 19c

TOP

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

27/31

# Distributed Database System Oracle 19c

Global database name uniquely identifies a database in the system

Listing the contents of a system view GLOBAL\_NAME

```
SELECT * FROM GLOBAL_NAME;
```

GLOBAL\_NAME

---

DB.DATA-PC07

It is possible to change a global database name when connected as a database administrator

Renaming global database name

```
ALTER DATABASE RENAME GLOBAL_NAME TO jrg.f8y792s.informatics.uow.edu.au;
```

# Distributed Database System Oracle 19c

Database link is a connection between two physical database servers that allow a user to access them as one logical database

Listing the existing database links

```
SELECT * FROM USER_DB_LINKS;
```

no rows selected

Message

Assume, that we are connected as a user **scott** to a database server **DATA-PC01**. Then, it is possible to create a **database link** to a user **jrg** located at a database server **DATA-PC07** in the following way

Creating a database link

```
CREATE DATABASE LINK "DB.DATA-PC07"  
CONNECT TO jrg  
IDENTIFIED BY rat  
USING 'data-pc07.adeis.uow.edu.au:1521/db';
```

A name of **database link** ( "**DB.DATA-PC07**" ) to a database must be the same as a **global database name** of the database linked to

# Distributed Database System Oracle 19c

A connection string ( '`data-pc07.adeis.uow.edu.au:1521/db`' ) determines a physical location of the database linked to

When a **database link** is created it is possible to access a relational table located in a database system linked to

Accessing a relational table over a database link

```
SELECT *  
FROM EMP@"DB.DATA-PC07";
```

Relational table name used: `EMP`

Database link: "`DB.DATA-PC07`"

A synonym can be used to implement **location transparency**

Creating a synonym

```
CREATE SYNONYM EMP07 FOR EMP@"DB.DATA-PC07";
```

Using a synonym

```
SELECT *  
FROM EMP07;
```

# References

T. Connolly, C. Begg, *Database Systems, A Practical Approach to Design, Implementation, and Management*, Chapter 24 Distributed DBMs - Concepts and Design, Chapter 25.7 Distribution in Oracle, Pearson Education Ltd, 2015

# CSCI235 Database Systems

# NoSQL Database Systems

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# NoSQL Database Systems

## Outline

NoSQL database system ? What is it ?

What is in the name "NoSQL" ?

Why NoSQL database systems ?

Properties of NoSQL database systems

# NoSQL database system ? What is it ?

NoSQL database systems were developed in early 2000s in a response to the demands for processing of vast amounts of data produced by increasing Internet usage and mobile geo-location technologies

Traditional solutions were either too expensive, not scalable, or required too much time to process data

Modern NoSQL database systems "borrowed" some of the solutions from the earlier systems and made significant advances in scalability and efficient processing of diverse types of data such as text audio, video, image, and geo-location

NoSQL database systems include the following types of database systems categorized by a logical view of data provided: key-value stores, document stores, graph stores, column stores

# NoSQL Database Systems

## Outline

NoSQL database system ? What is it ?

What is in the name "NoSQL" ?

Why NoSQL database systems ?

Properties of NoSQL database systems

# What is in the name "NoSQL"?

A term "**NoSQL**" has been used for the first time in late 1990 by C. Strozzi as a name of open-source relational database system that did not use SQL as a query language

The usage of "**NoSQL**" as it is recognized today has been used by J. Oskarsson in 2009 for the projects experimenting with alternate data storage, like BigTable (Google) and Dynamo (Amazon)

"**NoSQL**" database systems do not use SQL, run on the clusters of computers and provide different options for consistency and distribution

Despite its confrontational nature some people say that "**NoSQL**" means "**Not Only SQL**", but ... then it should be written as "**NOSQL**" and it is not

# NoSQL Database Systems

## Outline

NoSQL database system ? What is it ?

What is in the name "NoSQL" ?

Why NoSQL database systems ?

Properties of NoSQL database systems

# Why NoSQL database systems ?

**Data model:** A [tabular view](#) of data is not convenient for [hierarchical](#) and [network](#) database domains

**Impedance mismatch problem:** A [tabular view](#) of persistent data is not consistent with a transient view of data in [object-oriented programming languages](#)

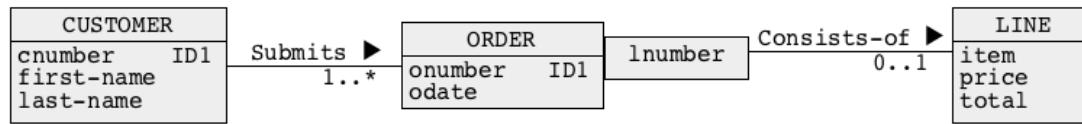
**Application and integration of database systems:** A structure that integrates many applications is more complex than single application requires, for example an index required by one application may cause performance problems for another application

**Clusters of small computers:** Relational databases that operate on the shared persistent storage subsystems are not designed to work well on the clusters of small computers

**Distribution and consistency:** [ACID](#) based transaction protocols are too strict for distributed transaction (see previous presentation)

# Why NoSQL database systems ?

**Data model:** A **tabular view** of data is not convenient for **hierarchical** and **network** database domains



Logical design provides the following relational schemas:

CUSTOMER(cnumber, first-name, **last-name**)  
**PRIMARY KEY** = (cnumber)

Relational schemas

ORDERS(onumber, odate, cnumber) **PRIMARY KEY** = (onumber)  
**FOREIGN KEY** = (cnumber) **REFERENCES** CUSTOMER(cnumber)

LINE(onumber, lnumber, item, price total)  
**PRIMARY KEY** = (onumber, lnumber)  
**FOREIGN KEY** = (onumber) **REFERENCES** ORDERS(onumber)

A query, that finds total amount of money spent by each customer requires join of three relational tables

# Why NoSQL database systems ?

**Impedance mismatch problem:** A **tabular view** of persistent data is not consistent with a transient view of data in **object-oriented programming languages**

Consider the relational tables

DOCUMENT schema of relational table
DOCUMENT(title, <b>class</b> , edate)
ACCESS schema of relational table
ACCESS(user, <b>class</b> )

A query **Find all documents available to a given user** is implemented as the following **SELECT** statement

```
SELECT title  
FROM DOCUMENT JOIN ACCESS  
    ON DOCUMENT.class = ACCESS.class  
WHERE user = ...;
```

SQL query optimizer is able to pick an appropriate algorithm for implementation of **join** operation

# Why NoSQL database systems ?

Object-oriented programming forces **encapsulation of object classes**

**Encapsulation of objects classes** forces **one object at a time** technique of object processing

```
class Document{ String title;  
               String className;  
               Date edate )
```

Document class

```
class Access( String user,  
              String className)
```

Access class

A query find all documents available to a user is implemented as

```
for(a in Access)  
    for( d in Document)  
        if a.user = ... and d.className = a.className then  
            out(d.title);
```

Object oriented implementation of a query

# Why NoSQL database systems ?

In **object-oriented view**, **join** is performed by an application and not by a database server

**One object at a time** technique of object processing forces **nested loop** implementation of **join** operation

When **object-oriented application** is processed remotely from a database systems the relational tables must be **transmitted to a remote site**

Database application programmers should directly access a bulk object (e.g. a collection of the documents) instead of forming the member objects individually and grouping them into a bulk object inside an application code

A query must be **stored at** and it must be **processed by a database server**

# Why NoSQL database systems ?

**Clusters of small computers:** Relational databases that operate on the shared persistent storage subsystems are not designed to work well on the clusters of small computers



# NoSQL Database Systems

## Outline

NoSQL database system ? What is it ?

What is in the name "NoSQL" ?

Why NoSQL database systems ?

Properties of NoSQL database systems

- Semistructured, unstructured data, and schemaless data model
- Specialized distribution models
- Weak consistency
- Relaxing durability
- Versioning

# Semistructured, schemaless data model

**Semistructured data** is a form of structured data that does not conform to the formal structure of data models associated with relational databases or other forms of data tables

**Semistructured data** contains tags or other markers to separate semantic elements and enforce hierarchies of records and fields within the data

**Schemaless data model** means that no particular data structure is used to store data in a database

**Schemaless database** does not require consistency with a rigid schema, e.g. database schema, relational schema, data type, table, etc.

**Schemaless database** does not enforce data type limitations on individual values

**Schemaless database** can **store structured** and **semistructured data**

# NoSQL Database Systems

## Outline

NoSQL database system ? What is it ?

What is in the name ?

Why No SQL database systems ?

### Properties of NoSQL database systems

- Semistructured, unstructured data, and schemaless data model
- Specialized distribution models
- Weak consistency
- Relaxing durability
- Versioning

# Specialized distribution models

Single server: means no distribution at all

Sharding: to support horizontal scalability we put different parts of data onto different servers (**shards**)

Data distribution leads to two **data replication modes**:

- Mater-slave replication: data are replicated across multiple nodes and one node is designated as master node, the others are slaves; all updates are made to the master and later on propagated to slaves
- Peer-to-peer replication: data replicated across multiple nodes; all replicas have the same weight, no master mode; nodes communicate their writes; all nodes read and write all data

Combining sharding and replication: use both master-slave replication and sharding

# NoSQL Database Systems

## Outline

NoSQL database system ? What is it ?

What is in the name ?

Why No SQL database systems ?

### Properties of NoSQL database systems

- Semistructured, unstructured data, and schemaless data model
- Specialized distribution models
- **Weak consistency**
- Relaxing durability
- Versioning

# Weak consistency

A typical **read consistency** principle where update is performed over two or more data items blocks access to all data items affected by the update, for example in **2PL** protocol

NoSQL database systems relax the **strict transactional consistency** to some extent

In a new model of consistency, data items can be left inconsistent over certain period of time called as **inconsistency window**

A concept of **eventual consistency** is used to enforce **replication consistency** over distributed and replicated data items

**Eventual consistency** means that the copies of data items can be inconsistent in **inconsistency window** and all copies will have the same value later on

# Weak consistency

Different domains have different **tolerances for inconsistencies** and we have to take this tolerance into account as make our decisions

Even in traditional relational database systems it is possible to relax consistency from the highest isolation level (**serializable**) to the lowest levels (**read-committed**) to get better performance

**CAP theorem:** Out of three properties of **Consistency**, **Availability**, and **Partition tolerance** you can get only two

**Consistency:** A state of a database satisfies the given consistency constraints at any moment in time

**Availability:** If you can talk to a node in a cluster then it can read and write data

**Partition:** Cluster can survive communication breakages that separate the cluster into multiple partitions unable to communicate with each other (**split brain**)

# Weak consistency

A single server system is **CA** because it has **Consistency** and **Availability** and not **Partition** tolerance

If a cluster must be tolerant of network partitions, then we have to trade consistency for availability

# NoSQL Database Systems

## Outline

NoSQL database system ? What is it ?

What is in the name ?

Why No SQL database systems ?

### Properties of NoSQL database systems

- Semistructured, unstructured data, and schemaless data model
- Specialized distribution models
- Weak consistency
- Relaxing durability
- Versioning

# Relaxing durability

If SQL systems follow **ACID** properties then NoSQL systems follow **BASE** properties: **Basically Available, Soft** state, **Eventually** consistent

**Relaxing durability** means that we trade off durability for higher performance, for example, apply updates to **in-memory representation of a database** and periodically flush changes to disk

Another class of **durability tradeoffs** comes with replicated data, for example, when a node processes an update but fails before that update is replicated to other nodes

# NoSQL Database Systems

## Outline

NoSQL database system ? What is it ?

What is in the name ?

Why No SQL database systems ?

### Properties of NoSQL database systems

- Semistructured, unstructured data, and schemaless data model
- Specialized distribution models
- Weak consistency
- Relaxing durability
- Versioning

# Versioning

A **version** is a particular form of something differing in certain respects from an earlier form or other forms of the same type of thing

**Versioning** in a database systems means that all modifications of data items are stored in a database together with timestamps when such modifications occurred

In practice **versioning** is performed to a predefined **depth**, i.e. a total number of versions of data item is determined when a data item is created

**Versioning** allows for representation of historical information

Numbering of **data versions** through **timestamps** allows to track when a data item has changed and if a new version is available allows to determine specifically which version is the most current one

# References

Harrison G., Next Generation Databases, NoSQL, NewSQL, Big Data, Apress, 2015

# CSCI235 Database Systems

## JSON and BSON Data Models

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# JSON and BSON Data Models

## Outline

JSON ? What is it ?

BSON ? What is it ?

# JSON ? What is it ?

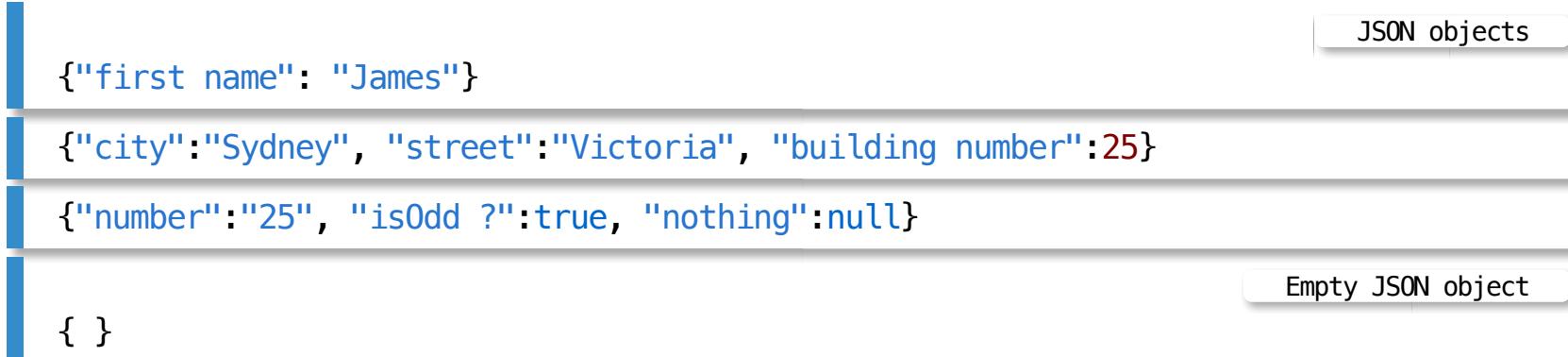
Java Script Object Notation (JSON) is a light-weight data interchange format

It is based on a subset of the JavaScript Programming

Language Standard ECMA-262 3rd Edition - December 1999 JSON is built on two structures:

- A collection of name:value pairs - object
- An ordered list of values - array

Object is an unordered set of name:value pairs



```
{"first name": "James"}  
{"city":"Sydney", "street":"Victoria", "building number":25}  
{"number":"25", "isOdd ?:true, "nothing":null}  
{ }
```

# JSON ? What is it ?

Value is either a string or a number or true or false or null or an object or an array

```
{"single quote": "\'", "back slash": "\\",
"forward slash": "\/", "line feed": "\u000A" }
```

JSON object with the special characters

```
{"fraction": 0.25, "true": true, "false": false, "nothing": null}
```

JSON object with a number, true, false, and nothing values

```
{"full name": {"first name": "James",
"initials": null,
"last name": "Bond"},  
"number": "007"}
```

Nested JSON object

```
{"colours": ["red", "green", "blue"]}
```

JSON object with an array of strings

```
{"numbers": [10, 20, 30, 40, 50]}
```

JSON object with an array of numbers

# JSON ? What is it ?

Array is a sequence of values

```
{"2dim array": [ [11,12,13,14],  
                 [21,22,23,24],  
                 [31,32,33,34] ]}
```

JSON object with two dimensional array

```
{"array of objects": [ {"name": "James Bond"},  
                      {"name": "Harry Potter"},  
                      {"name": "Robin Hood"} ] }
```

JSON object with an array of objects

```
{"array of truth": [true, false, true, true]}
```

JSON object with an array of Boolean values

```
{"array of nothing": [null, null, null, null, null]}
```

JSON object with an array of nulls

```
{"empty array": [ ]}
```

JSON object with an empty array

```
{"array of varieties": [1, "one", true, ["a", "b", "c"]]}
```

JSON object with an array of different values

# JSON and BSON Data Models

## Outline

[JSON ? What is it ?](#)

[BSON ? What is it ?](#)

# BSON ? What is it ?

**Binary JSON (BSON)** is a computer data interchange format used mainly as a data storage and network transfer format in the MongoDB database system

**BSON** is a binary encoded serialization of **JSON**-like documents (objects)

**BSON** includes the extensions not in **JSON** that allow for representation of data types

For example **BSON** has a **Date** type and a **BinData** type

The common data types include: **null**, **boolean**, **number**, **string**, **array**, **embedded document(object)**

The new data types include: **date**, **regular expression**, **object id**, **binary data**, **code**

# BSON ? What is it ?

{ "nothing": null }	BSON object with a null
{ "truth": false }	BSON object with a Boolean value
{ "pi": 3.14 }	BSON object with a floating point value
{ "int": NumberInt("345") }	BSON object with an integer value
{ "long": NumberLong("1234567890987654321") }	Long integer value in BSON object with long integer value
{ "greeting": "Hello !" }	BSON object with a string value
{ "pattern": /(Hi Hello)!/ }	BSON object with a regular expression
{ "date": new Date("2016-10-17") }	BSON object with a date
{ "array of varieties": [1, "one", true, ["a", "b", "c"] ] }	BSON object with an array of different values

# BSON ? What is it ?

<pre>{"_id":ObjectId()}</pre>	BSON object with an object identifier
<pre>{"code":function(){ /* ... */ }}</pre>	BSON object with a functions
<pre>{"Double":124.56}</pre>	BSON object with a timestamp
<pre>{"timestamp":Timestamp(1234567,0)}</pre>	BSON object with a timestamp
<pre>{"timestamp":Timestamp()}</pre>	BSON object with a timestamp
<pre>{"maxkey":{\$maxKey: 1} }</pre>	BSON with the largest possible value
<pre>{"minkey":{\$minKey : 1} }</pre>	BSON with the smallest possible value

# BSON ? What is it ?

Every document (object) in BSON must have an "\_id" key !

"\_id" key's value can be of any type

A default value of "\_id" key is ObjectId()

In a collection of documents (objects) every document (object) must have a unique value for "\_id"

If a document has no "\_id" key then when it is inserted into a collection "\_id" key is automatically added to the document

<pre>{"_id":ObjectId()}</pre>	BSON object with an automatically generated object identifier
<pre>{"_id":ObjectId("507f191e810c19729de860ea") }</pre>	BSON object with an object identifier
<pre>{"_id": "student number": NumberInt(1234567) }</pre>	BSON object with an object identifier
<pre>{"_id": "enrolment": "1234567 CSCI235 01-AUG-2021" }</pre>	BSON object with an object identifier

# References

[Introducing JSON](#)

[BSON](#)

[MongoDB, Document Data Format: BSON](#)

Banker K., Bakkum P., Verch S., Garret D., Hawkins T., MongoDB in Action, 2nd ed., Manning Publishers, 2016

# CSCI235 Database Systems

## MongoDB Databases, Collections, Documents

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# MongoDB Databases, Collections, Documents

## Outline

[Basics](#)

[Architecture](#)

[Server](#)

[Databases](#)

[Collections](#)

[Documents](#)

[Formatting](#)

[Simple DDL and DML](#)

# Basics

MongoDB is a database system that belongs to a class of so called NoSQL database systems based on a data model different from the relational model and data definition, manipulation, retrieval, and administration languages different from SQL

MongoDB data model (BSON) is based on the concept of key:value pairs grouped into documents and arrays

MongoDB database system operates on a number of databases

A MongoDB database is a set of collections

A MongoDB collection is a set of documents

A MongoDB document is a set of key:value pairs

A MongoDB value is either an atomic value or a document or an array

A MongoDB atomic value is one of the types included in the BSON specification like number, string, date, and so on

A MongoDB array is a sequence of values

# Basics

Each MongoDB **key:value** pair must have a unique **key** within **document**

Each MongoDB **document** must have a unique identifier within a **collection**

Each MongoDB **collection** must have a unique name within a **database**

```
{"_id": ObjectId(),
  "full name": {"first name":"James",
                 "initials":null,
                 "last name":"Bond"},
  "employee number":"007",
  "skills":["cooking", "painting", "gardening"],
  "cars owned": [ {"rego":"007-1",
                  "made":"Porsche"}, {"rego":"007-2",
                  "made":"Ferrari"} ],
  "secret codes": [ [1,2,3,4],
                   [9,8,7,5] ],
  "date of birth":new Date("1960-01-01")
}
```

A sample BSON document

# MongoDB Databases, Collections, Documents

## Outline

[Basics](#)

[Architecture](#)

[Server](#)

[Databases](#)

[Collections](#)

[Documents](#)

[Formatting](#)

[Simple DDL and DML](#)

# Architecture

MongoDB **flexible storage architecture** automatically manages the movement of data between storage engine technologies using native **replication**

MongoDB stores data as documents in a **binary representation** called **BSON (Binary JSON)**

MongoDB **query model** is implemented **as methods or functions within the API of a specific programming language**, as opposed to a completely separate language like **SQL**

MongoDB provides **horizontal scale-out** for databases on low cost, commodity hardware or cloud infrastructure using a technique called **sharding**, which is transparent to applications

In-Memory storage engine enables performance advantages of **in-memory computing** for **operational** and **real-time analytics** workloads

MongoDB **Enterprise Advanced** provides extensive capabilities to defend, detect, and control access to data (**data security**)

# Architecture

MongoDB **Ops Manager** makes easy for operation teams to deploy, monitor, backup and scale the system (system management)

MongoDB **Atlas** provides all of the features of **Database as a Service** cloud computing model

# MongoDB Databases, Collections, Documents

## Outline

[Basics](#)

[Architecture](#)

[Server](#)

[Databases](#)

[Collections](#)

[Documents](#)

[Formatting](#)

[Simple DDL and DML](#)

# Server

Starting MongoDB server with options `--dbpath`, `--port`, and `--bind_ip`

```
mongod --dbpath data --port 4000 --bind_ip 10.0.2.100
```

Starting MongoDB server

Starting MongoDB server with options `--dbpath`, `--port`, and server running on a `localhost`

```
mongod --dbpath data --port 4000 --bind_ip localhost
```

Starting MongoDB server

or simply ...

```
mongod --dbpath data --port 4000
```

Starting MongoDB server

Starting MongoDB command based shell

```
mongo --port 4000
```

Starting MongoDB command client

# Server

## Getting the first help from MongoDB shell

```
help
```

Getting MongoDB help

```
db.help()      help on db methods
show dbs       show database names
show collections show collections in current database
use db_name    set current database
... ... ...     ... ... ...
```

MongoDB help messages

# MongoDB Databases, Collections, Documents

## Outline

[Basics](#)

[Architecture](#)

[Server](#)

[Databases](#)

[Collections](#)

[Documents](#)

[Formatting](#)

[Simple DDL and DML](#)

# Databases

## Setting a default database

```
use database-name
```

Setting 'database-name' as a default database

## For example using a database `local`

```
use local
```

Setting 'local' as a default database

## Creating and switching to a new database `mydb`

```
use mydb
```

Setting 'mydb' as a default database

## Listing the databases

```
show dbs
```

Listing all databases

local **0.000GB**  
mydb **0.000GB**

Databases

# MongoDB Databases, Collections, Documents

## Outline

[Basics](#)

[Architecture](#)

[Server](#)

[Databases](#)

[Collections](#)

[Documents](#)

[Formatting](#)

[Simple DDL and DML](#)

# Collections

Creating a new collection with an empty document

```
db.mycol.insert({})
```

Creating a new collection mycol

Listing the contents of a collection

```
db.mycol.find()
```

Listing a collection mycol

```
{ "_id" : ObjectId("57e385f8ffc660a351b58010") }
```

Listing the collections

```
show collections
```

Listing the names of collections

```
mycol
```

# MongoDB Databases, Collections, Documents

## Outline

[Basics](#)

[Architecture](#)

[Server](#)

[Databases](#)

[Collections](#)

[Documents](#)

[Formatting](#)

[Simple DDL and DML](#)

# Documents

Creating a new non empty document

Inserting a document into a collection mycol

```
db.mycol.insert({ "one": "1", "many ones": [1,1,1,1] })
```

Listing the contents of a collection

MongoDB Shell

```
db.mycol.find()
```

```
{ "_id" : ObjectId("57e385f8ffc660a351b58010") }
{ "_id" : ObjectId("57e38cbeffc660a351b58012"),
  "one" : "1", "many ones" : [ 1, 1, 1, 1 ] }
```

# MongoDB Databases, Collections, Documents

## Outline

[Basics](#)

[Architecture](#)

[Server](#)

[Databases](#)

[Collections](#)

[Documents](#)

[Formatting](#)

[Simple DDL and DML](#)

# Formatting

Listing the nicely formatted contents of a collection

Listing a nicely formatted collection

```
db.mycol.find().pretty()

{ "_id" : ObjectId("57e385f8ffc660a351b58010") }
{ "_id" : ObjectId("57e38cbeffc660a351b58012"),
  "one" : "1",
  "many ones" : [ 1,
                  1,
                  1,
                  1
                ]
}
```

# MongoDB Databases, Collections, Documents

## Outline

[Basics](#)

[Architecture](#)

[Server](#)

[Databases](#)

[Collections](#)

[Documents](#)

[Formatting](#)

[Simple DDL and DML](#)

# Simple DDL and DML

Removing all documents from a collection

```
db.mycol.remove({})
```

Removing all documents from a collection

Removing a collection

```
db.mycol.drop()
```

Dropping a collection

Removing a database

```
db.dropDatabase()
```

Dropping a database

# Simple DDL and DML

Let a file `dbload.js` contains the following `insert` methods

```
Inserting documents into a collection mycol
db.mycol.insert({ "CITY": { "name": "Wollongong",
                            "population": "80K",
                            "country": "Australia",
                            "state": "New South Wales" } });
db.mycol.insert({ "EMPLOYEE": { "enum": 1234567,
                                "full-name": "Janusz R. Getta",
                                "salary": "200K",
                                "hobbies": [ "cooking",
                                             "painting",
                                             "gardening" ] } });
```

Processing a script inserts two documents into a collection `mycol`

```
load("dbload.js")
```

Listing a collection `mycol`

```
db.mycol.find().pretty()
```

Listing a nicely formatted collection mycol

# Simple DDL and DML

```
{  
    "_id" : ObjectId("57e3c817fe6a1bfd5105022a"),  
    "CITY" : {  
        "name" : "Wollongong",  
        "population" : "80K",  
        "country" : "Australia",  
        "state" : "New South Wales"  
    }  
}  
{  
    "_id" : ObjectId("57e3c817fe6a1bfd5105022b"),  
    "EMPLOYEE" : {  
        "enum" : 1234567,  
        "full-name" : "Janusz R.Getta",  
        "salary" : "200K",  
        "hobbies" : [  
            "cooking",  
            "painting",  
            "gardening"  
        ]  
    }  
}
```

# References

## MongoDB Architecture

Banker K., Bakkum P., Verch S., Garret D., Hawkins T., MongoDB in Action, 2nd ed., Manning Publishers, 2016

# CSCI235 Database Systems

## MongoDB Query Language

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# MongoDB Query language

## Outline

[MongoDB query language](#)

[A sample database](#)

[Simple queries](#)

[Queries with Boolean operations](#)

[Queries on nested documents](#)

[Queries on arrays](#)

[Projections](#)

[Queries about `NULLs` and missing keys](#)

[Iterations over a cursor](#)

# MongoDB query language

MongoDB query language is based on a concept of pattern matching

A query is expressed as a **BSON** pattern and all documents that match the pattern are included in an answer

A method **find()** can be used to match a pattern with the documents in a collection **orders**

```
db.orders.find({"_id":"ALFKI"})
```

find()

Matching of an empty pattern **{ }** with a collection **orders** returns an entire collection

```
db.orders.find({})
```

find()

Finding the first 3 documents in a collection **orders**

```
db.orders.find({}).limit(3)
```

find()

Finding all documents in a collection **orders** and listing the results in a nice format

```
db.orders.find({}).pretty()
```

find()

# MongoDB Query language

## Outline

[MongoDB query language](#)

[A sample database](#)

[Simple queries](#)

[Queries with Boolean operations](#)

[Queries on nested documents](#)

[Queries on arrays](#)

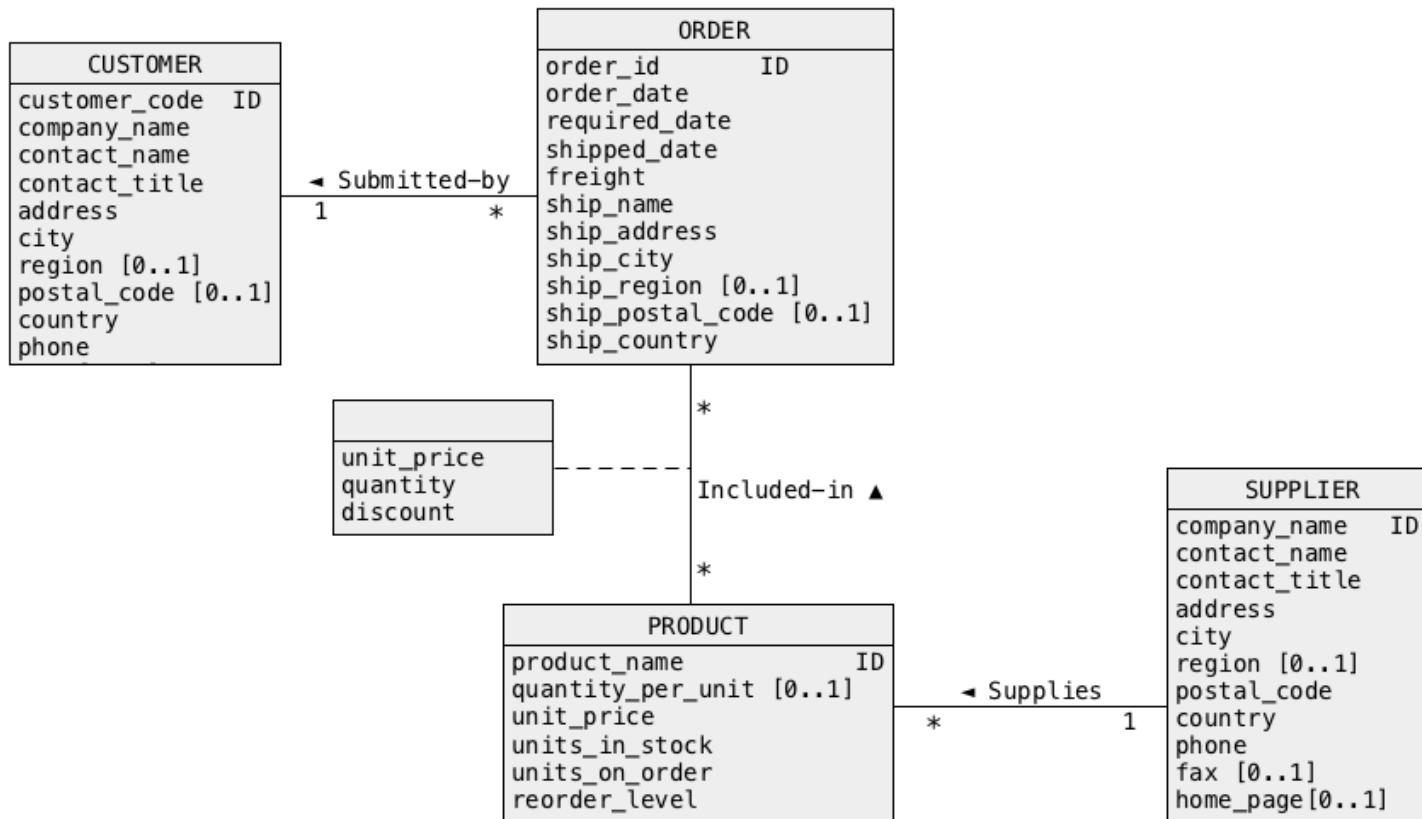
[Projections](#)

[Queries about `NULLs` and missing keys](#)

[Iterations over a cursor](#)

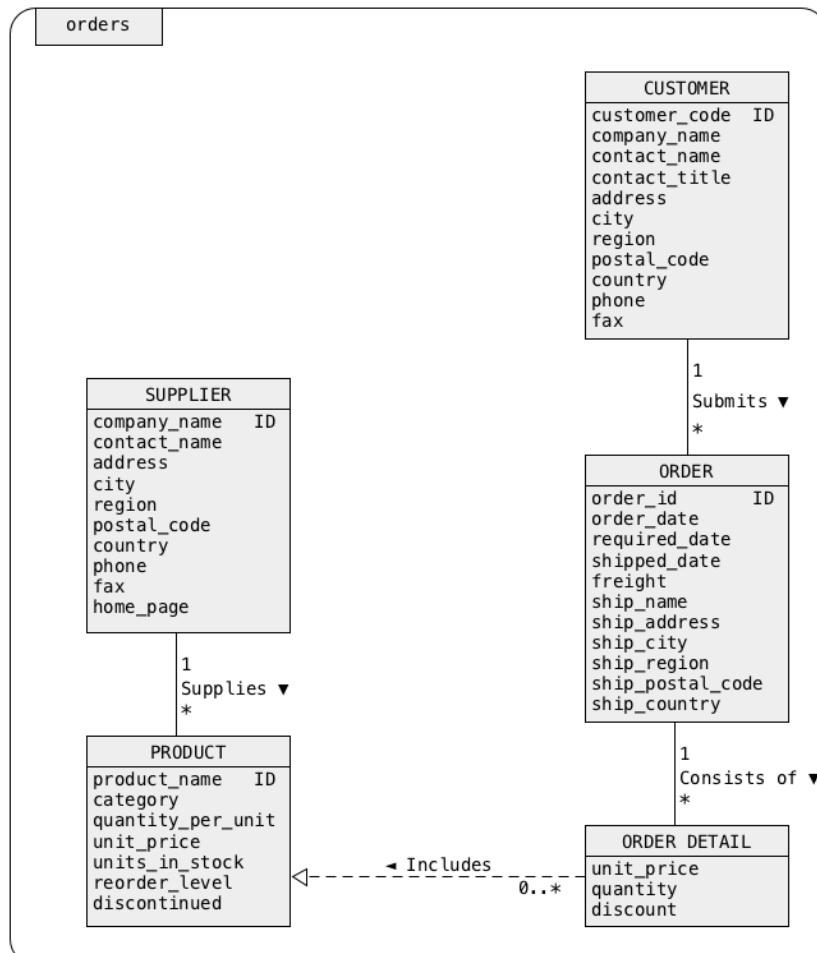
# A sample database

A conceptual schema of a database with information about **suppliers**, **products**, **customers**, **orders**, and **details of orders**



# A sample database

## A sample collection `orders`

[TOP](#)

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

6/35

# A sample database

A sample document, that belongs to a class CUSTOMER

```
{  
    "_id" : "ALFKI",  
    "CUSTOMER" : {  
        "customer code" : "ALFKI",  
        "company name" : "Alfreds Futterkiste",  
        "contact name" : "Maria Anders",  
        "contact title" : "Sales Representative",  
        "address" : "Obere Str. 57",  
        "city" : "Berlin",  
        "region" : null,  
        "postal code" : "12209",  
        "country" : "Germany",  
        "phone" : "030-0074321",  
        "fax" : "030-0076545",  
        "submits" : [ ]  
    }  
}
```

find()

# A sample database

A sample nested document, that belongs to a class **CUSTOMER**

```
{  
    "_id" : "FAMIA",  
    "CUSTOMER" : {  
        "customer code" : "FAMIA",  
        ...  
        "submits" : [  
            {  
                "ORDER" : {  
                    "order id" : 328,  
                    ...  
                    "consists of" : [  
                        {  
                            "ORDER DETAIL" : {  
                                "product name" : "Louisiana Fiery Hot Pepper Sauce",  
                                ...  
                            }  
                        },  
                        {  
                            "ORDER DETAIL" : {  
                                "product name" : "Raclette Courdavault",  
                                ...  
                            }  
                        }  
                    ]  
                }  
            }  
        ]  
    }  
}
```

find()

[TOP](#)

# A sample database

A sample nested document, that belongs to a class **SUPPLIER**

```
{  
    "_id" : "Karkki Oy",  
    "SUPPLIER" : {  
        "company name" : "Karkki Oy",  
        "contact name" : "Anne Heikonen",  
        "contact title" : "Product Manager",  
        "address" : "Valtakatu 12",  
        ...  
        "supplies" : [  
            {  
                "PRODUCT" : {  
                    "product name" : "Maxilaku",  
                    "category name" : "Confections",  
                    ...  
                }  
            },  
            {  
                "PRODUCT" : {  
                    "product name" : "Valkoinen suklaa",  
                    "category name" : "Confections",  
                    ...  
                }  
            }  
        ]  
    }  
}
```

find()

# MongoDB Query language

## Outline

[MongoDB query language](#)

[A sample database](#)

[Simple queries](#)

[Queries with Boolean operations](#)

[Queries on nested documents](#)

[Queries on arrays](#)

[Projections](#)

[Queries about `NULLs` and missing keys](#)

[Iterations over a cursor](#)

# Simple queries

Find total number of documents in a collection

```
db.orders.count()
```

count()

Find all information about all customers

```
db.orders.find({"CUSTOMER":{$exists:true}})
```

Find entire class

Find all information about all suppliers

```
db.orders.find({"SUPPLIER":{$exists:true}})
```

Find entire class

Find all information about the customers living in Germany

```
db.orders.find({"CUSTOMER.country":"Germany"})
```

Access path

Find all information about the suppliers living in a city of Oviedo

```
db.orders.find({"SUPPLIER.city":"Oviedo"})
```

Access path

Find all information about the suppliers who live in the Netherlands  
and have a contact title Accounting Manager

```
db.orders.find({"SUPPLIER.country":"Netherlands",  
    "SUPPLIER.contact title":"Accounting Manager"})
```

And

# MongoDB Query language

## Outline

[MongoDB query language](#)

[A sample database](#)

[Simple queries](#)

[Queries with Boolean operations](#)

[Queries on nested documents](#)

[Queries on arrays](#)

[Projections](#)

[Queries about `NULLs` and missing keys](#)

[Iterations over a cursor](#)

# Boolean expressions

Comparison "key" = "value"

```
{"key": "value"}
```

=

```
{"key": {$eq: "value"}}
```

=

Comparison "key" > "value"

```
{"key": {$gt: "value"}}
```

>

Disjunction ("key1" = "value1") or ("key2" = "value2")

```
{$or: [{"key1": "value1"}, {"key2": "value2"}]}
```

\$or

Conjunction ("key1" = "value1") and ("key2" = "value2")

```
{$and: [{"key1": "value1"}, {"key2": "value2"}]}
```

\$and

Boolean expression (( "key1" = "value1" ) or ( "key2" = "value2" ))  
and ( "key3" = "value3" )

```
{$and: [{$or: [{"key1": "value1"}, {"key2": "value2"}]}, {"key3": "value3"}]}
```

Boolean expression

# Boolean expressions

Negation of a comparison "key" not ="value"

```
{"key":{$not:{$eq:"value"}}}
```

not =

Negation of an expression not (( "key1"="value1" ) or  
( "key2"="value2" ) )

```
{$nor:[{"key1":"value1"}, {"key2":"value2"}]}
```

not or

Negation nor ( "key1"="value1" )

```
 {$nor:[{"key1":"value1"}]}
```

not =

# Queries with Boolean operations

Find all information about the suppliers who live in the **Netherlands** and have a contact title **Accounting Manager**

```
db.orders.find({$and: [{"SUPPLIER.country": "Netherlands"}, {"SUPPLIER.contact title": "Accounting Manager"}]})
```

\$and

Find all information about the suppliers who live in the **Netherlands** or have a contact title **Accounting Manager**

```
db.orders.find({$or: [{"SUPPLIER.country": "Netherlands"}, {"SUPPLIER.contact title": "Accounting Manager"}]})
```

\$or

Find all information about the customers who live in **France** or in **Germany**

```
db.orders.find({"CUSTOMER.country": {$in: ["France", "Germany"]}})
```

\$in

# Queries with Boolean operations

Find all information about the customers who do not live in **Germany**

```
db.orders.find({"CUSTOMER.country":{$not:{$eq:"Germany"}}})
```

\$not

Find all information about the customers who do not **both** live in **Netherlands** or have a contact title **Accounting Manager**

Find all information about the customers who do not live in **Netherlands** and do not have a contact title **Accounting Manager**

```
db.orders.find({$nor:[{"SUPPLIER.country":"Netherlands"}, {"SUPPLIER.contact title":"Accounting Manager"}]})
```

\$nor

Find all information about the customers who do not live in **Germany**

```
db.orders.find({$nor:[{"CUSTOMER.country":"Germany"}]})
```

\$nor

# MongoDB Query language

## Outline

[MongoDB query language](#)

[A sample database](#)

[Simple queries](#)

[Queries with Boolean operations](#)

[Queries on nested documents](#)

[Queries on arrays](#)

[Projections](#)

[Queries about `NULLs` and missing keys](#)

[Iterations over a cursor](#)

# MongoDB Query language

## Outline

[MongoDB query language](#)

[A sample database](#)

[Simple queries](#)

[Queries with Boolean operations](#)

[Queries on nested documents](#)

[Queries on arrays](#)

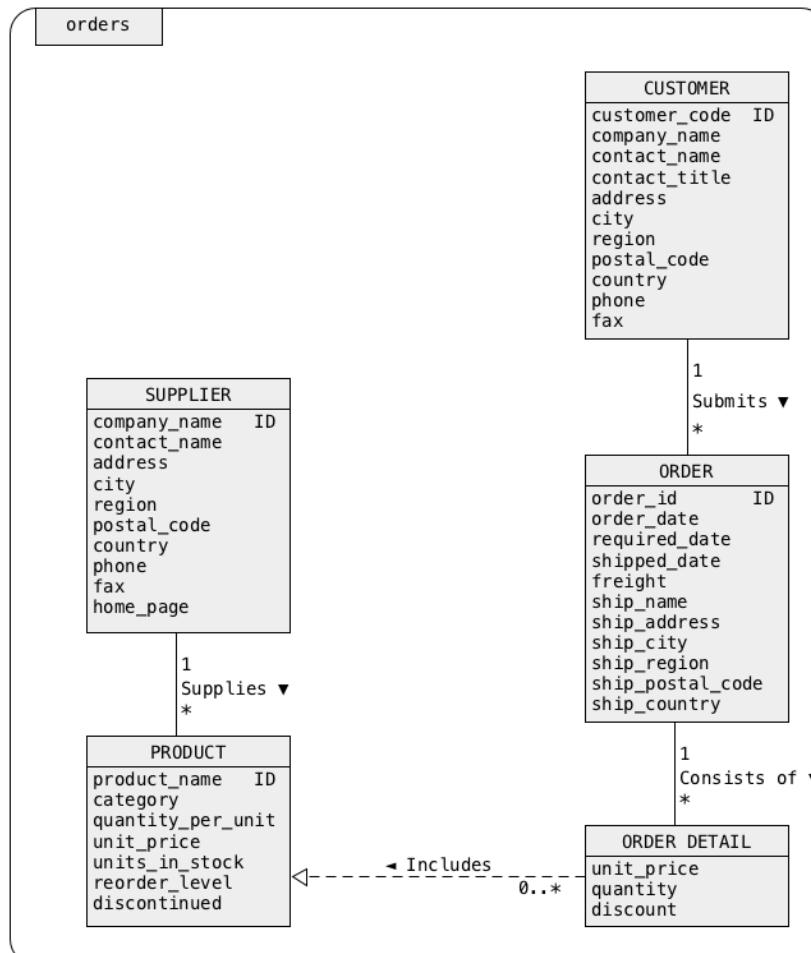
[Projections](#)

[Queries about `NULLs` and missing keys](#)

[Iterations over a cursor](#)

# Queries on nested documents

A sample collection `orders`



# Queries on nested documents

Find all information about suppliers who supply a product named **Laughing Lumberjack Lager**

```
db.orders.find({"SUPPLIER.supplies.PRODUCT.product name":"Laughing Lumberjack Lager"})
```

path

Find all information about suppliers living in **London** who supply a product named **Chai**

```
db.orders.find({$and:[{"SUPPLIER.city":"London"}, {"SUPPLIER.supplies.PRODUCT.product name":"Chai"}]})
```

path and path

Find all information about suppliers living in **London** who supply a product named **Chai** or a product named **Chang**

```
db.orders.find({$and:[{"SUPPLIER.city":"London"}, {$or:[{"SUPPLIER.supplies.PRODUCT.product name":"Chai"}, {"SUPPLIER.supplies.PRODUCT.product name":"Chang"}]}]})
```

(path or path) and path

```
db.orders.find({$and:[{"SUPPLIER.city":"London"}, {"SUPPLIER.supplies.PRODUCT.product name":{$in:["Chai", "Chang"]}}]})
```

path and path

# Queries on nested documents

Find all information about suppliers living in **London** who supply a product named **Chai** and a product named **Chang**

```
db.orders.find({$and: [{"SUPPLIER.city": "London"},  
                      {"SUPPLIER.supplies.PRODUCT.product name": "Chai"},  
                      {"SUPPLIER.supplies.PRODUCT.product name": "Chang"}]})
```

path and path and path

Find all information about suppliers who supply at least one product

```
db.orders.find({$and: [{"SUPPLIER.supplies": {$exists: true}},  
                      {"SUPPLIER.supplies": {$ne: []}}]}).count()
```

path and path

Find all information about suppliers who do not supply any products

```
db.orders.find({$and: [{"SUPPLIER.supplies": {$exists: true}},  
                      {"SUPPLIER.supplies": {$eq: []}}]}).count()
```

path and path

Find all information about customers who submitted an order for at least one product **Flotemysost**

```
db.orders.find({"CUSTOMER.submits.ORDER.consists of.ORDER DETAIL.product name": "Flotemysost"})
```

long path

# MongoDB Query language

## Outline

[MongoDB query language](#)

[A sample database](#)

[Simple queries](#)

[Queries with Boolean operations](#)

[Queries on nested documents](#)

[Queries on arrays](#)

[Projections](#)

[Queries about `NULLs` and missing keys](#)

[Iterations over a cursor](#)

# Queries on arrays

Array equal to [1,2,3,4,5]

```
{"array":{$all:[1,2,3,4,5]}}
```

find()

Array includes an element that satisfies a condition

```
{"array":{$elemMatch:{$eq:2}}}
```

find()

```
{"array":{$elemMatch:{$gt:2,$lt:4}}}
```

find()

Array includes a document satisfies a condition

```
{"array":{$elemMatch:{"key":{$eq:2}}}}
```

find()

```
{"array":{$elemMatch:{"key":{$gt:2,$lt:4}}}}
```

find()

Size of an array

```
{"array":{$size:5}}
```

find()

# Queries on arrays

Find all information about customers who submitted an order that contains only a product **Boston Crab Meat** at unit price **14.7** in quantity **20** with discount equal to **0**

```
db.orders.find({"CUSTOMER.submits.ORDER.consists of":  
    {$eq: [{"ORDER DETAIL": {"product name": "Boston Crab Meat",  
        "unit price": 14.7,  
        "quantity": 20,  
        "discount": 0}}]}})
```

Find all information about suppliers such that the second supplied product is named **Chang**

```
db.orders.find({"SUPPLIER.supplies.1.PRODUCT.product name": "Chang"})
```

Find all information about suppliers such that the first supplied product is named **Chai** and the second supplied product is named **Chang**

```
db.orders.find({$and: [{"SUPPLIER.supplies.0.PRODUCT.product name": "Chai"},  
    {"SUPPLIER.supplies.1.PRODUCT.product name": "Chang"}]})
```

# Queries on arrays

Find all information about customers who purchased at least one product with discount greater than **0.2**

```
db.orders.find({"CUSTOMER.submits.ORDER.consists of.ORDER DETAIL.discount":{$gt:0.2}})
```

path > v

```
db.orders.find({"CUSTOMER.submits.ORDER.consists of":{$elemMatch:{"ORDER DETAIL.discount":{$gt:0.2}}}})
```

path &elemMatch v

Find all information about customers who purchased at least one product with discount equal to **0.25** and quantity equal to **16**

```
db.orders.find({"CUSTOMER.submits.ORDER.consists of":{$elemMatch:{"ORDER DETAIL.discount":{$eq:0.25},  
"ORDER DETAIL.quantity":{$eq:16}}}})
```

path &elemMatch v1, ...,vn

Find all information about customers who purchased 4 products in one order

```
db.orders.find({"CUSTOMER.submits.ORDER.consists of":{$size:4}})
```

\$size

# Queries on arrays

Find all information about customers who purchased more than 4 products in one order

```
db.orders.find({"CUSTOMER.submits.ORDER.consists of.4":{$exists:true}})
```

path.n \$exists

# MongoDB Query language

## Outline

[MongoDB query language](#)

[A sample database](#)

[Simple queries](#)

[Queries with Boolean operations](#)

[Queries on nested documents](#)

[Queries on arrays](#)

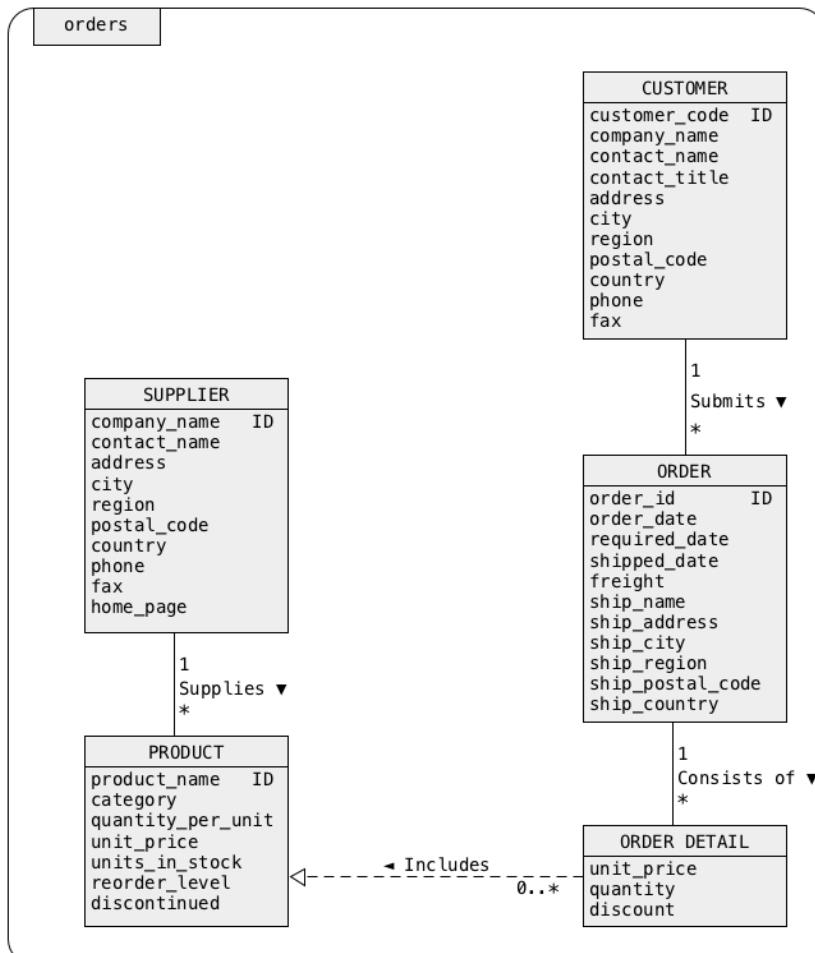
[Projections](#)

[Queries about `NULLs` and missing keys](#)

[Iterations over a cursor](#)

# Projections

A sample collection **orders**



# Projections

Find only a company name and contact name for all suppliers

Projection

```
db.orders.find({"SUPPLIER":{$exists:true}}, {"_id":0, "SUPPLIER.company name":1, "SUPPLIER.contact name":1})
```

Find all information about suppliers except a company name, contact name and \_id

Projection

```
db.orders.find({"SUPPLIER":{$exists:true}}, {"_id":0, "SUPPLIER.company name":0, "SUPPLIER.contact name":0})
```

Find all information about suppliers except products supplied by suppliers and \_id

Projection

```
db.orders.find({"SUPPLIER":{$exists:true}}, {"_id":0, "SUPPLIER.supplies":0}))
```

Find only information about products supplied by suppliers

Projection

```
db.orders.find({"SUPPLIER":{$exists:true}}, {"_id":0, "SUPPLIER.supplies":1}))
```

Projection

```
db.orders.find({"SUPPLIER":{$exists:true}}, {"_id":0, "SUPPLIER.supplies.PRODUCT":1}))
```

# Projections

Find only the names of products supplied by suppliers

Projection

```
db.orders.find({"SUPPLIER":{$exists:true}}, {"_id":0, "SUPPLIER.supplies.PRODUCT.product name":1})
```

Find only the names of products and categories of products supplied by suppliers

Projection

```
db.orders.find({"SUPPLIER":{$exists:true}}, {"_id":0, "SUPPLIER.supplies.PRODUCT.product name":1,
    "SUPPLIER.supplies.PRODUCT.category name":1})
```

Find only company names of suppliers and the names of products supplied by suppliers

Projection

```
db.orders.find({"SUPPLIER":{$exists:true}}, {"_id":0, "SUPPLIER.company name":1,
    "SUPPLIER.supplies.PRODUCT.product name":1})
```

# MongoDB Query language

## Outline

[MongoDB query language](#)

[A sample database](#)

[Simple queries](#)

[Queries with Boolean operations](#)

[Queries on nested documents](#)

[Queries on arrays](#)

[Projections](#)

[Queries about `NULLs` and missing keys](#)

[Iterations over a cursor](#)

# Queries about nulls and missing keys

Find all information about the customers who have no **region** information

```
db.orders.find({"CUSTOMER.region":null})
```

null

Find all information about the customers who have **region** information

```
db.orders.find({"CUSTOMER.region":{$not:{$eq:null}}})
```

\$not null

Find all information about the customers who have **PO Box** in their description

```
db.orders.find({"CUSTOMER.PO BOX":{$exists:true}}).count()
```

\$exists

Find all information about the customers who do not have **PO Box** in their description

```
db.orders.find({"CUSTOMER.PO BOX":{$exists:false}})
```

\$exists

```
db.orders.find({"CUSTOMER.PO BOX":{$not:{$exists:true}}})
```

\$ not \$exists

# MongoDB Query language

## Outline

[MongoDB query language](#)

[A sample database](#)

[Simple queries](#)

[Queries with Boolean operations](#)

[Queries on nested documents](#)

[Queries on arrays](#)

[Projections](#)

[Queries about `NULLs` and missing keys](#)

[Iterations over a cursor](#)

# Iterations over a cursor

Create a cursor and display all information about suppliers

```
var cursor = db.orders.find({"SUPPLIER":{$exists:true}})  
while(cursor.hasNext())  
{ print(tojson(cursor.next())); }
```

cursor

```
var cursor = db.orders.find({"SUPPLIER":{$exists:true}})  
cursor.forEach(printjson)
```

cursor

# References

## [MongoDB Reference, Operators, Query and Projection Operators](#)

Banker K., Bakkum P., Verch S., Garret D., Hawkins T., MongoDB in Action, 2nd ed., Manning Publishers, 2016

# CSCI235 Database Systems

## JSON Schema

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# JSON Schema

## Outline

### JSON Schema ? What is it ?

Basics

Types

String

Numeric types

Boolean

Null

Object

Array

Combined schemas

# JSON Schema ? What is it ?

JSON Schema is a **vocabulary** for validation of **JSON objects**

JSON Schema describes the structures and types of existing **JSON objects**

JSON Schema allows for automated validation of **JSON objects**

JSON Schema itself is a **JSON object**

The latest version of **JSON Schema** is a version **2020-12**

The following **JSON object**

```
{"city": "Sydney", "street": "Victoria", "building": 25}
```

JSON object

validates well against the following **JSON Schema**

```
{ "type": "object",
  "properties": { "city": { "type": "string" },
                  "street": { "type": "string" },
                  "building": { "type": "number",
                                "minimum": 1}
                }
}
```

JSON schema

# JSON Schema

## Outline

### JSON Schema ? What is it ?

Basics

Types

String

Numeric types

Boolean

Null

Object

Array

Combined schemas

# Basics

The simplest possible **JSON schema** is the following (an empty object)

```
{ }
```

The simplest JSON schema

The schema given above validates every **JSON object**

Extensions of the schema given above impose the restrictions on the validated objects

For example, the following **JSON schema**

```
{ "type":"object",
  "properties": { "Hello world message": { "type":"string" } }
}
```

A Hello world JSON schema

validates well the objects

```
{ "Hello world message":"Hello world !" }
```

A Hello world object

```
{ "Hello world message":"Hello world !", "Greeting":"How are you ?" }
```

A How are you object

# Basics

The following extension of the previous schema

```
{ "type": "object",
  "properties": { "Hello world message": { "type": "string",
                                             "minLength": 1,
                                             "maxLength": 13 } }
}
```

An extended Hello world JSON schema

fails validation of an object

```
{ "Hello world message": "Hello world !!!" }
```

A Hello world object

because a string **Hello world !!!** is too long (15 characters)

It validates well the objects

```
{ "Hello world message": "Hello world !" }
```

A Hello world object

```
{ "Hello world message": "Hello world !", "Greeting": "How are you ?" }
```

A How are you object

# JSON Schema

## Outline

### JSON Schema ? What is it ?

Basics

Types

String

Numeric types

Boolean

Null

Object

Array

Combined schemas

# Types

In **JSON schema** a **type** keyword is associated with a data type

**JSON schema** defines the following **basic types**: **string**, **Numeric types**, **object**, **array**, **boolean**, and **null**

A **basic type** associated with a keyword **type** can be either one of the types listed above or it can be an array of the types listed above

If **type** keyword is associated with a name of a **basic type** then a respective value must be of the same **basic type**, for example

```
{ "type": "string"}
```

Type string

It determines a type of a respective value as **string**

If **type** keyword is an **array** of **basic types** then each element of the array must be unique, for example

```
{ "type": [ "string", "number" ] }
```

Type string or number

It determines a type of a respective value as either **string** or **number**

# JSON Schema

## Outline

### JSON Schema ? What is it ?

Basics

Types

String

Numeric types

Boolean

Null

Object

Array

Combined schemas

# String

A type **string** determines a type of a respective value as a string of characters

For example

```
{ "type": "string" }
```

String type

validates well any string of characters including an empty string

**string** type has the following restrictions: **minLength**, **maxLength**, **pattern**, and **format**

For example

```
{"type": "string",  
 "minLength": 10,  
 "maxLength": 20 }
```

String type with length restrictions

validates well any string longer than 9 characters and shorter than 21 characters

# String

A **pattern** restriction is used to match a string with a given regular expression

For example

```
{"type": "string",  
 "pattern": "[1-9] [0-9]" }
```

String type with regular expression restriction

validates well any string of characters representing a number in a range from "10" to "99"

A **format** restriction is used to match a string with predefined format

For example

```
{"type": "string",  
 "format": "date" }
```

String type with format restriction

validates well any string of characters representing a date in a format "YYYY-MM-DD" for example "2020-07-21"

# JSON Schema

## Outline

### JSON Schema ? What is it ?

Basics

Types

String

Numeric types

Boolean

Null

Object

Array

Combined schemas

# Numeric types

JSON Schema has two numeric types `integer` and `number`

A type `integer` is used for validation of integer numbers

For example

```
{ "type": "integer" }
```

Integer type

validates well any integer number like `-5, 0, 7`, etc

A type `number` is used for validation of any numeric value, either integer or floating point numbers

For example

```
{ "type": "number" }
```

Number type

validates well any integer or floating point number like like `-35.7, -7, 0.0, 23, 23.4`, etc

# Numeric types

Numeric types `integer` and `number` have the following restrictions:  
`multipleOf`, `minimum`, `exclusive Minimum`, `maximum`,  
`exclusiveMaximum`

For example

```
{ "type": "integer",  
  "multipleOf": 5 }
```

Integer type with `multipleOf` restriction

validates well any integer like `0, 5, 10, 15`, etc

For example

```
{ "type": "number",  
  "multipleOf": 0.2 }
```

Number type with `multipleOf` restriction

validates well any number like `0.0, 0.2, 0.4, 0.6`, etc

# Numeric types

The restrictions `minimum`, `maximum`, `exclusiveMinimum`, and `exclusiveMaximum` can be used for expressing ranges

For example

```
{ "type":"integer",  
  "minimum": 0,  
  "exclusiveMaximum": 5}
```

Integer type with a range restriction

validates well integer numbers `0`, `1`, `2`, `3`, and `4`

# JSON Schema

## Outline

### JSON Schema ? What is it ?

Basics

Types

String

Numeric types

Boolean

Null

Object

Array

Combined schemas

# Boolean

A type `boolean` determines a type of a respective value either as `true` or `false`

For example

```
{ "type":"boolean" }
```

Boolean type

validates well the values `true` and `false` and no other values

# JSON Schema

## Outline

### JSON Schema ? What is it ?

Basics

Types

String

Numeric types

Boolean

Null

Object

Array

Combined schemas

# Null

A type `null` determines a type of a respective value as `null`

For example

```
{ "type":"null" }
```

Null type

validates well a lack of value `null` and no other values

# JSON Schema

## Outline

### JSON Schema ? What is it ?

Basics

Types

String

Numeric types

Boolean

Null

Object

Array

Combined schemas

# Object

A type **object** determines a type of a respective value as **JSON object**

For example

```
{ "type":"object" }
```

Object type

validates well any object, like for example

```
{"city":"Sydney", "street":"Victoria", "building":25}
```

Flat JSON object

```
{ "name":"School of Astronomy",
  "courses": [ {"code":"SOA101",
    "title":"Astronomy for Kids",
    "credits":3},
    {"code":"SOA201",
      "title":"Black Holes",
      "credits":6}
  ]
}
```

Nested JSON object

```
{ }
```

Empty JSON object

# Object

A key **properties** define the **key:value** pairs an object consists of

A key **properties** is associated with an object that consists of **key:value** pairs where each **key** is a name of a property and each **value** is a **JSON schema** used to validate a property

For example **JSON schema**

```
{ "type":"object",
  "properties": { "city": { "type":"string" },
                  "street": { "type":"string" },
                  "building": { "type":"number",
                                "minimum":1 }
                }
}
```

JSON schema

validates well the objects

```
{ "city":"Dapto","street":"Station","building":7}
```

JSON object

```
{ "city":"Dapto","street":"Station"}
```

JSON object

```
{ "city":"Dapto","street":"Station","building":7,"type":"skyscraper"}
```

JSON object

```
{ }
```

Empty JSON object

# Object

A key **additionalProperties** determines whether additional properties not listed in **JSON schema** are allowed

For example **JSON schema**

```
{ "type":"object",
  "properties": { "city": { "type":"string" },
                  "street": { "type":"string" },
                  "building": { "type":"number",
                                "minimum":1 }
                },
  "additionalProperties":false
}
```

JSON schema

validates well the objects

```
{ "city":"Dapto","street":"Station","building":7}
```

JSON object

```
{ "city":"Dapto","street":"Station"}
```

JSON object

```
{ }
```

Empty JSON object

# Object

A key **requiredProperties** determines the compulsory properties of an object

For example **JSON schema**

```
{ "type": "object",
  "properties": { "city": { "type": "string" },
                  "street": { "type": "string" },
                  "building": { "type": "number",
                                "minimum": 1 }
                },
  "requiredProperties": ["city", "street"]
}
```

JSON schema

validates well the objects

```
{ "city": "Dapto", "street": "Station", "building": 7 }
```

JSON object

```
{ "city": "Dapto", "street": "Station" }
```

JSON object

# Object

The keys `minProperties` and `maxProperties` determine the minimum and maximum number of properties of an object

For example **JSON schema**

```
{ "type": "object",
  "properties": { "city": { "type": "string" },
                  "street": { "type": "string" },
                  "building": { "type": "number",
                                "minimum": 1 }
                },
  "minProperties": 2,
  "maxProperties": 3
}
```

JSON schema

validates well the objects

```
{ "city": "Dapto", "street": "Station", "building": 7}
```

JSON object

```
{ "city": "Dapto", "street": "Station"}
```

JSON object

```
{ "street": "Station", "building": 7}
```

JSON object

# Object

A key **dependencies** determine the existence dependencies of one property on another

A value associated with a key **dependencies** is an object

Each entry in the object maps from a name of a property into an array of properties that are required whenever the property is present

For example **JSON schema**

```
{ "type": "object",
  "properties": { "city": { "type": "string" },
                  "street": { "type": "string" },
                  "building": { "type": "number",
                                "minimum": 1 }
                },
  "dependencies": { "street": [ "building" ] } }
```

JSON schema

validates well the objects

```
{ "city": "Dapto", "street": "Station", "building": 7 }
```

JSON object

```
{ "street": "Station", "building": 7 }
```

JSON object

```
{ "city": "Dapto", "building": 7 }
```

JSON object

# Object

A key **patternProperties** determines a syntax of key names in an object

A value associated with a key **patternProperties** is an object

For example **JSON schema**

```
{ "type":"object",
  "patternProperties": { "^"(city|CITY)": { "type":"string" },
                        "^"(street|STREET)": { "type":"string" },
                        "^"(building|BUILDING)": { "type":"number",
                                                    "minimum":1 }
                      },
  "additionalProperties":false
}
```

JSON schema

validates well the objects

```
{ "city":"Dapto","STREET":"Station","building":7}
{ "street":"Station","BUILDING":7}
{ "CITY":"Dapto","BUILDING":7}
{ }
```

JSON object

JSON object

JSON object

JSON object

# JSON Schema

## Outline

### JSON Schema ? What is it ?

Basics

Types

String

Numeric types

Null

Object

Array

Combined schemas

# Array

A type **array** determines a type of a respective value as **array**

For example

```
{ "type":"array" }
```

Array type

validates well any array, like for example

```
[1, 2, 3, 4, 5, 6, 7]
```

Array of numbers

```
["ab", "cde", "efgh", "ijklm"]
```

Array of strings

```
[ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
```

Array of arrays

```
[]
```

Empty array

# Array

A key `items` determines a type of values in an array

For example

```
{ "type":"array",
  "items": { "type":"string" }
}
```

Array type

validates well any array of strings like

```
["ab", "cde", "ab", "ijklm"]
```

Array of strings

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Array of strings

```
[]
```

Empty array

# Array

**Tuple validation** determines a type of values when an array is a collection of items, each item has a different schema, and position of each item is important

For example

```
{ "type":"array",
  "items": [ { "type":"string" },
             { "type":"integer",
               "minimum":1} ],
  "additionalItems":false
}
```

Array type with tuple validation

validates well any array of values like

```
["Station St.", 25]
```

A tuple

```
["Victoria St."]
```

Incomplete tuple

```
[]
```

Empty tuple

# Array

A key `uniqueItems` requires each item in an array to be unique

For example

```
{ "type":"array",
  "items": { "type":"string" },
  "uniqueItems":true
}
```

Array type

validates well any array of strings like

```
["ab", "cde", "efgh", "ijklm"]
```

Array of strings

```
[ "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" ]
```

Array of strings

```
[]
```

Empty array

# Array

The keys `minItems` and `maxItems` determine a size of an array

For example

```
{ "type":"array",
  "items": { "type":"string" },
  "uniqueItems":true,
  "minItems":3,
  "maxItems":5
}
```

Array type

validates well any array of strings like

```
["ab", "cde", "efgh", "ijklm"]
```

Array of strings

```
[ "0", "1", "2", "3", "4" ]
```

Array of strings

# JSON Schema

## Outline

### JSON Schema ? What is it ?

Basics

Types

String

Numeric types

Boolean

Null

Object

Array

Combined schemas

# Combined schemas

The keys `anyOf`, `allOf` and `oneOf` can be used to combine **JSON Schemas** together

For example `anyOf` keyword can be used to validate JSON object against one or more of the given schemas

```
{ "anyOf": [ {"type":"string",
              "maxLength":5},
              {"type":"boolean"} ]
}
```

Validation with `anyOf`

validates well the objects like strings up to 5 characters long or Boolean values `true` and `false`

# Combined schemas

For example **anyOf** keyword can be used to validate JSON object against one or more of the given schemas

```
{ "anyOf": [ {"type":"string",  
             "maxLength":5},  
            {"type":"string",  
             "minLength":3} ]  
}
```

Validation with anyOf

validates well any string

# Combined schemas

The keys `anyOf`, `allOf` and `oneOf` can be used to combine **JSON Schemas** together

For example `allOf` keyword can be used to validate JSON object against all of the given schemas

```
{ "allOf": [ { "type": "string",  
               "maxLength": 5},  
             { "type": "string",  
               "minLength": 3} ]  
 }
```

Validation with `allOf`

validates well any string no longer than 5 characters and no shorter than 3 characters

# Combined schemas

The keys `anyOf`, `allOf` and `oneOf` can be used to combine **JSON Schemas** together

For example `oneOf` keyword can be used to validate JSON object against exactly one of the given schemas

```
{ "oneOf": [ {"type":"string",
               "maxLength":5,
               "minLength":5},
              {"type":"integer",
               "maximum":9,
               "minimum":1} ] }
```

Validation with `oneOf`

validates well any string 5 characters long or any integer number in a range from 1 to 9 inclusive

# References

[JSON Schema](#)

[Understanding JSON Schema](#)

[MongoDB - JSON Schema validation](#)

[MongoDB - \\$jsonSchema operator](#)

# CSCI235 Database Systems

## Validation with JSON Schema

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Validations

## Outline

### Overview

JSON Schema validations

\$jsonSchema operator

Validation of any BSON documents

Validation of flat BSON documents

Validation of nested BSON documents

# Overview

Semistructured and schemaless properties of JSON/BSON data model allow for very flexible manipulation of database structures

In the same moment very flexible and uncontrolled manipulations of database structures open the possibilities for pretty easy corruption of database structures and database contents, for example, due to the random mistakes

Practice shows that certain level of verification of database consistency constraints is always needed

MongoDB provides the capability to validate documents during updates and insertions (and not deletions)

Validation rules are specified within `createCollection()` method using the `validator` option

It is possible use `collMod` command with the `validator` option to turn on/off validation rules

# Validations

## Outline

[Overview](#)

[JSON Schema validations](#)

[\\$jsonSchema operator](#)

[Validation of any BSON documents](#)

[Validation of flat BSON documents](#)

[Validation of nested BSON documents](#)

# JSON Schema validation

JSON Schema is a [BSON document](#) that defines the structure of JSON data for validation, documentation, and interaction control

JSON Schema validation of a [JSON document](#) is performed through verification of consistency of the structures and contents of the document with [JSON Schema](#)

JSON Schema is based on the concepts "borrowed" from [XML Schema](#)

A method `createCollection()` creates an empty collection and associates `validator` with a collection

If validation with [JSON Schema](#) is required then `validator` is associated with a [JSON schema](#) that determines the structures and the contents of the documents in the collection

It is possible use `collMod` method with the `validator` option to turn on/off [validation rules](#)

# Validations

## Outline

[Overview](#)

[JSON Schema validations](#)

[\\$jsonSchema operator](#)

[Validation of any BSON documents](#)

[Validation of flat BSON documents](#)

[Validation of nested BSON documents](#)

# \$jsonSchema operator

In MongoDB an operator `$jsonSchema` is associated with a **JSON Schema** to be used for validation of **BSON documents**

`$jsonSchema` operator can also be used to query with `find` command or with `$match` aggregation stage

The following application of `$jsonSchema` operator validates any **BSON document** in a collection `department`

```
db.createCollection("department",
  {"validator":{$jsonSchema:{ "bsonType": "object" } }
} );
```

\$jsonSchema operator

For example, the following **BSON document** validates well

A sample BSON document that validates with \$jsonSchema validator above

```
{"name":"School of Astronomy",
 "address":{"street":"Franz Josef Str",
            "bldg":4},
 "courses":[{"code":"SOA101",
             "title":"Astronomy for Kids",
             "credits":3} ] }
```

# Validations

## Outline

[Overview](#)

[JSON Schema validations](#)

[\\$jsonSchema operator](#)

[Validation of any BSON documents](#)

[Validation of flat BSON documents](#)

[Validation of nested BSON documents](#)

# Validation of any BSON documents

The following **JSON Schema** validates well any document in a collection **anyDocument**

```
db.createCollection("anyDocument",
  {"validator":{$jsonSchema:{"bsonType":"object"} }
});
```

JSON Schema validator that validates any document

Note the application of **bsonType** in MongoDB validation instead of **type** in standard **JSON schema**

# Validations

## Outline

[Overview](#)

[JSON Schema validations](#)

[\\$jsonSchema operator](#)

[Validation of any BSON documents](#)

[Validation of flat BSON documents](#)

[Validation of nested BSON documents](#)

# Validation of flat BSON documents

The following BSON document in a collection `tiny`

```
db.tiny.insert({ "name": "Harry Potter" });
```

A tiny BSON document

fails validation against the following JSON schema

```
db.createCollection( "tiny",
                      { "validator": { $jsonSchema:
                        { "bsonType": "object",
                          "properties": { "name": { "bsonType": "string" } },
                          "required": [ "name" ],
                          "additionalProperties": false
                        } } } );
```

\$jsonSchema validator

A validation fails because a key `additionalProperties` is associated with `false` and a key `"_id"` included in every BSON document is not on `required` list

# Validation of flat BSON documents

The following BSON document in a collection `tiny`

```
db.tiny.insert({"_id":"HP666","name":"Harry Potter"});
```

A tiny BSON document

passes validation against the following JSON schema

```
db.createCollection( "tiny",
                      {"validator":{$jsonSchema:
{"bsonType":"object",
 "properties":{ "_id": {"bsonType":"string"},
 "name": {"bsonType":"string"} },
 "required": [ "_id", "name" ],
 "additionalProperties":false
} } } );
```

\$jsonSchema validator

A validation passes because a key `additionalProperties` is associated with `false` and a key `"_id"` included in every BSON document is on `required` list

# Validation of flat BSON documents

The following BSON documents in a collection `empty`

```
db.empty.insert({"_id":"HP666"});
db.empty.insert({"_id":"HP667","name":"Harry Potter"});
db.empty.insert({"_id":"HP668","name":"Harry Potter","occupation":"wizard"});
```

BSON documents in a collection `empty`

pass validation against the following JSON schema

```
db.createCollection( "empty",
                      {"validator":{$jsonSchema:
{"bsonType":"object",
 "properties":{"_id":{"bsonType":"string"} },
 "required":["_id"],
 "additionalProperties":true
} } );
```

`$jsonSchema validator`

A validation passes because a key `additionalProperties` is associated with `true` and a key `"_id"` included in every BSON document is on `required` list

# Validation of flat BSON documents

The following BSON document in a collection `flat`

```
db.flat.insert({"_id":"HP666","name":"Harry Potter","age":NumberInt("100")});
```

A tiny BSON document

passes validation against the following JSON schema

```
db.createCollection( "flat",
                      {"validator":{$jsonSchema:
{"bsonType":"object",
 "properties":{"name":{"bsonType":"string",
                     "maxLength":100},
              "age":{"bsonType":"int",
                     "maximum":200,
                     "exclusiveMaximum":true} },
 "required":["name","age","_id"],
 "additionalProperties":false
} } );
```

\$jsonSchema validator

A default type of a value associated with a key `"_id"` is `string`

# Validation of nested BSON documents

The following BSON documents in a collection `department`

```
db.department.insert({"_id":"Finance","budget":123});  
db.department.insert({"_id":"Sales","budget":123,"fee":456});
```

BSON document

pass validation against the following JSON schema

```
db.createCollection("department",  
    { "validator":{$jsonSchema:  
        {"bsonType":"object",  
         "properties":{ "_id": {"bsonType": "string"},  
                       "budget": {"bsonType": "double",  
                                 "description": "Budget of department"},  
                       "fee": {"bsonType": "double",  
                               "description": "Fee paid"}  
                     },  
         "required": ["_id", "budget"],  
         "additionalProperties": false  
       } } } );
```

jsonSchema validator

# Validations

## Outline

[Overview](#)

[JSON Schema validations](#)

[\\$jsonSchema operator](#)

[Validation of any BSON documents](#)

[Validation of flat BSON documents](#)

[Validation of nested BSON documents](#)

# Validation of nested BSON documents

The following BSON document in a collection `nested`

```
db.nested.insert({"_id":"HP667",
                  "name":"Harry Potter",
                  "address":{"city":"Dapto",
                             "code":NumberInt("2530") }
                });
```

A nested BSON document

passes validation against the following JSON schema

```
db.createCollection( "nested",
                     {"validator":{$jsonSchema:
{"bsonType":"object",
 "properties":{ "_id":{"bsonType":"string",
                        "name":{"bsonType":"string",
                                "maxLength":100},
                        "address":{"bsonType":"object",
                                   "properties":{"city":{"bsonType":"string",
                                         "minLength":5,"maxLength":30},
                                   "code":{"bsonType":"int",
                                         "maximum":9999,
                                         "exclusiveMaximum":false} },
                        "required":["city","code"],
                        "additionalProperties":false } },
                    "name":true,
                    "address":true,
                    "_id":true,
                    "additionalProperties":false } } );
```

\$jsonSchema validator

# Validation of nested BSON documents

The following BSON document in a collection `nested`

```
db.nested.insert({"_id":"HP667",
                  "name":"Harry Potter",
                  "cars":["Ferrari","Mercedes","Hyundai"]
});
```

A nested BSON document

passes validation against the following JSON schema

```
db.createCollection( "nested",
                     {"validator":{$jsonSchema:
{"bsonType":"object",
 "properties":{ "_id": {"bsonType":"string"},
                "name": {"bsonType":"string",
                          "maxLength":100},
                "cars": {"bsonType":"array",
                          "items": {"bsonType":"string"},
                          "uniqueItems":true }
               },
 "required": ["name", "cars", "_id"],
 "additionalProperties":false
} } } );
```

\$jsonSchema validator

# Validation of nested BSON documents

The following BSON document in a collection `month`

```
db.month.insert( {"_id":"Winter",
                  "months":["June","July","August"]} );
```

BSON document

passes validation against the following JSON schema

```
db.createCollection( "month",
                      {"validator":{$jsonSchema:
{"bsonType":"object",
 "properties":{"_id":{"bsonType":"string"},
 "months":{"bsonType":"array",
           "description":"Winter",
           "items":{"bsonType":"string"} } },
 "required":["_id","months"],
 "additionalProperties":false
} } );
```

\$jsonSchema validator

# Validation of nested BSON documents

The following BSON document in a collection **addressBook**

```
db.month.insert( {"_id":"addressBook",
                  "address":[{"city":"Sydney",
                               "street":"25 Victoria St."},
                             {"city":"Wollongong",
                               "street":"125 Northfields Ave."}] } );
```

BSON document

passes validation against the following JSON schema

```
db.createCollection("department",
                    {"validator":{$jsonSchema:
                      {"bsonType":"object",
                       "properties":{ "_id":{"bsonType":"string"},
                                     "address":{"bsonType":"array",
                                                "description":"Cities and streets",
                                                "items":{"bsonType":"object",
                                                         "properties":{ "city":{"bsonType":"string"},
                                                                       "street":{"bsonType":"string"} },
                                                         "required":["city","street"],
                                                         "additionalProperties":false
                                                       }
                                                   }
                                                 },
                       "required":[ "_id", "address"],
                       "additionalProperties":false
                     } } } );
```

\$jsonSchema validator

# Validation of nested BSON documents

The following BSON documents in a collection `school`

```
db.school.insert(  
  { "name": "School of Astronomy",  
    "code": "SOA",  
    "total_staff_number": 25,  
    "budget": 10000,  
    "address": { "street": "Franz Josef Str",  
                "bldg": 4,  
                "city": "Vienna",  
                "country": "Austria"},  
    "courses": [ { "code": "SOA101",  
                  "title": "Astronomy for Kids",  
                  "credits": 3},  
                { "code": "SOA201",  
                  "title": "Black Holes",  
                  "credits": 6},  
                { "code": "SOA301",  
                  "title": "Dark Matter",  
                  "credits": 12} ]  
  } );
```

BSON document

validates well against the following JSON schema ...

# Validation of nested BSON documents

```
db.createCollection("school", { "validator": { "$jsonSchema": {  
    "bsonType": "object",  
    "required": ["name", "code", "total staff number", "budget", "address", "courses"],  
    "properties": { "name": { "bsonType": "string",  
        "description": "Name of a department" },  
        "code": { "bsonType": "string",  
        "description": "Code of a department" },  
        "total staff number": { "bsonType": "double",  
            "description": "Total staff number" },  
        "budget": { "bsonType": "double",  
            "description": "Budget of a department" },  
        "address": { "bsonType": "object",  
            "required": ["street", "bldg", "city", "country"],  
            "properties": { "street": { "bsonType": "string",  
                "description": "Street name" },  
                "bldg": { "bsonType": "double",  
                "description": "Building number" },  
                "city": { "bsonType": "string",  
                "description": "City name" },  
                "country": { "bsonType": "string",  
                "description": "Country name" } } },  
        "courses": { "bsonType": "array",  
            "items": { "bsonType": "object",  
                "required": ["code", "title", "credits"],  
                "properties": { "code": { "bsonType": "string",  
                    "description": "Subject code" },  
                    "title": { "bsonType": "string",  
                    "description": "Subject title" },  
                    "credits": { "bsonType": "double",  
                    "description": "Total credit points" } } } } } } );
```

JSON schema

# References

[JSON Schema](#)

[Understanding JSON Schema](#)

[MongoDB - JSON Schema validation](#)

[MongoDB - \\$jsonSchema operator](#)

# CSCI235 Database Systems

## MongoDB Aggregation Framework

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# MongoDB Aggregation Framework

## Outline

Aggregation framework ? What is it ?

Sample database

Aggregation operators

\$project

\$match

\$limit, \$skip and \$sample

\$sort and \$count

\$unwind

\$group

\$out

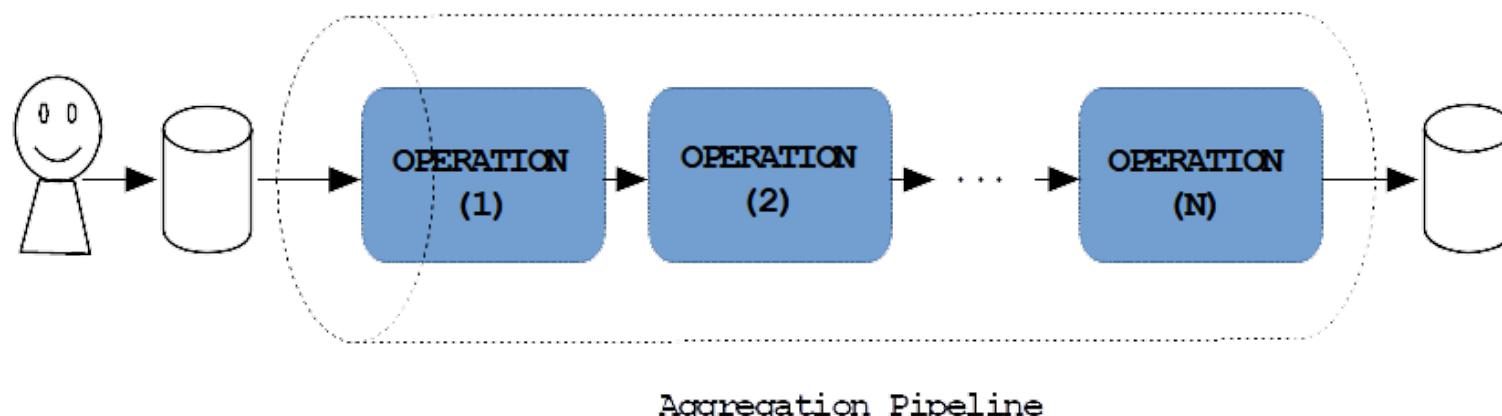
\$lookup

# Aggregation framework? What is it?

Aggregation framework is a query language that can be used to transform and to combine data from multiple documents in order to generate new information not available in any single document

Aggregation framework makes a task of database search much easier and more efficient through specification of a series of operations in an array and processing it in a single call

Aggregation framework defines an aggregation pipeline where the output from each step in the pipeline provides input to the next step



# Aggregation framework? What is it?

Every step in a **pipeline** executes a single operation on the **input documents** to transform the **input** and to generate **output document**

A **pipeline** processes a **stream of documents** through several operations like **filtering**, **projecting**, **grouping**, **sorting**, **limiting**, **skipping**, and the others

The same operations can be repeated many times in a **pipeline** in any order

**Aggregation framework** in MongoDB is similar to **SQL WITH** clause of **SELECT** statement

Some of the **aggregation operators** that can be used in an **aggregation pipeline** in MongoDB are similar to **SQL SELECT, WHERE, GROUP BY, HAVING, ORDER BY**, and **JOIN** clauses

**Pipelined data processing** is one of two basic ways how data processing can be parallelised

The other way is **partitioned data processing**

# Aggregation framework

Some of the operators that can be used in an **aggregation pipeline**:

- **\$project**: Extracts the components of a documents to be placed in an output document (similar to **SELECT** clause)
- **\$match**: Filters the documents to be processed, similar to **find()** (and similar to **WHERE** clause)
- **\$limit** and **\$skip**: Limits and skips the documents to be passed to the next operation
- **\$unwind**: Expands (unnest) an array, generating one output document for each array entry
- **\$group**: Groups documents by a specified key
- **\$sort** and **\$count**: Sorts and counts the documents
- **\$out**: Saves the results from a **pipeline** to a collection
- **\$lookup**: Joins two collections of documents
- **\$merge**: Merges two collections of documents

# MongoDB Aggregation Framework

## Outline

Aggregation framework ? What is it ?

Sample database

Aggregation operators

\$project

\$match

\$limit, \$skip and \$sample

\$sort and \$count

\$unwind

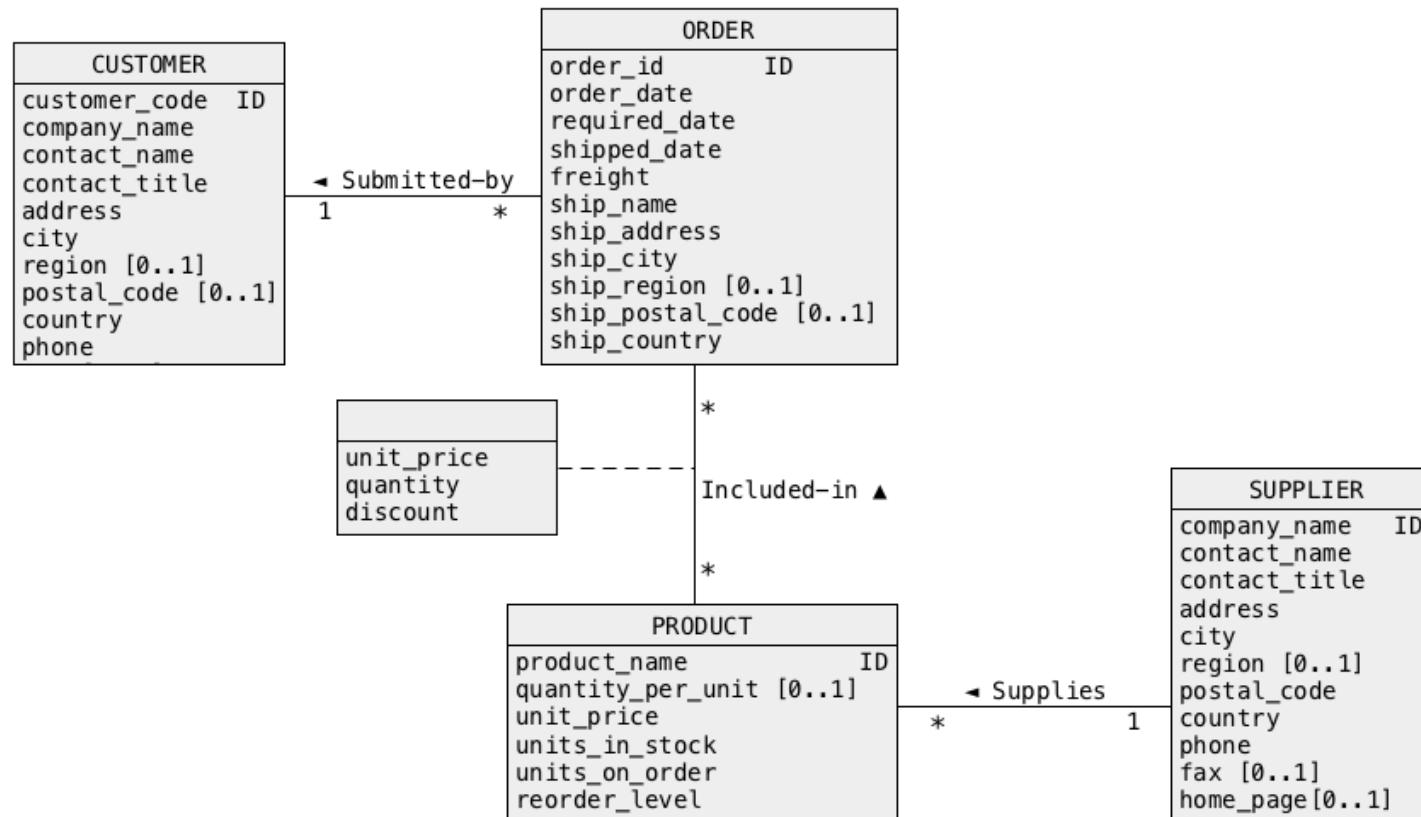
\$group

\$out

\$lookup

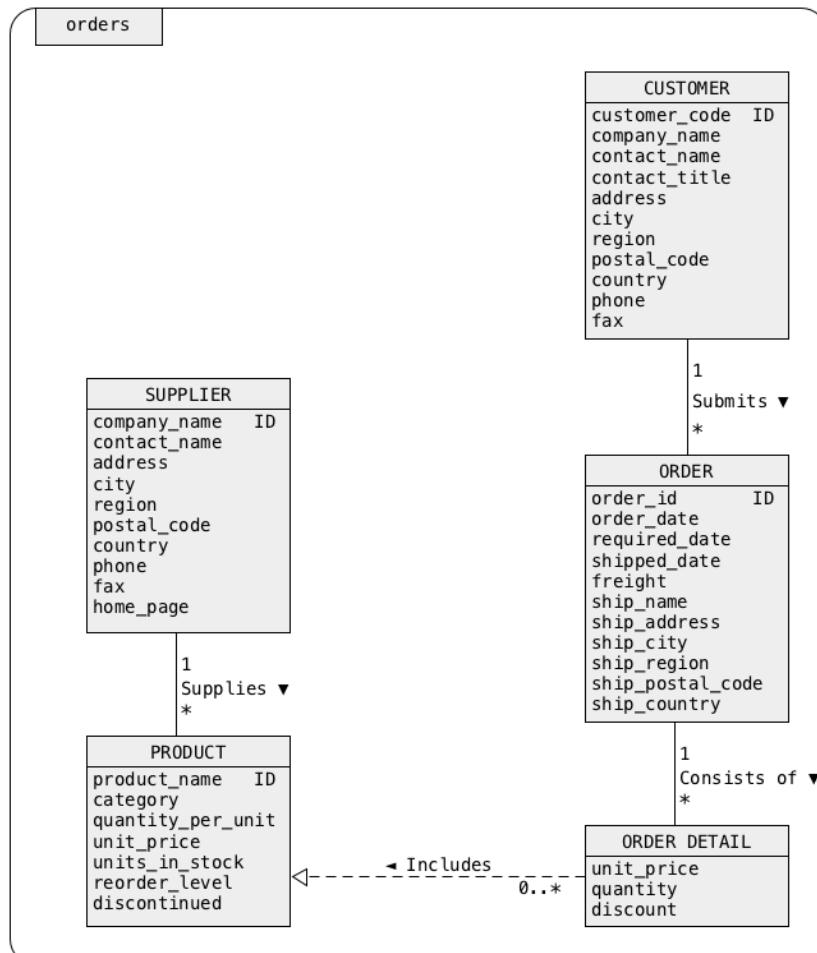
# A sample database

A conceptual schema of a database with information about **suppliers**, **products**, **customers**, **orders**, and **details of orders**



# A sample database

## A sample collection `orders`

[TOP](#)

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

8/46

# A sample database

A sample document, that belongs to a class CUSTOMER

```
{  
    "_id": "ALFKI",  
    "CUSTOMER": {  
        "customer code": "ALFKI",  
        "company name": "Alfreds Futterkiste",  
        "contact name": "Maria Anders",  
        "contact title": "Sales Representative",  
        "address": "Obere Str. 57",  
        "city": "Berlin",  
        "region": null,  
        "postal code": "12209",  
        "country": "Germany",  
        "phone": "030-0074321",  
        "fax": "030-0076545",  
        "submits": [ ]  
    }  
}
```

CUSTOMER

# A sample database

A sample nested document, that belongs to a class **CUSTOMER**

```
{  
    "_id": "FAMIA",  
    "CUSTOMER": {  
        "customer code": "FAMIA",  
        ...  
        "submits": [  
            {  
                "ORDER": {  
                    "order id": 328,  
                    ...  
                    "consists of": [  
                        {  
                            "ORDER DETAIL": {  
                                "product name": "Louisiana Fiery Hot Pepper Sauce",  
                                ...  
                            }  
                        },  
                        {  
                            "ORDER DETAIL": {  
                                "product name": "Raclette Courdavault",  
                                ...  
                            }  
                        }  
                    ]  
                }  
            }  
        ]  
    }  
}
```

CUSTOMER

[TOP](#)

# A sample database

A sample nested document, that belongs to a class **SUPPLIER**

```
{  
    "_id": "Karkki Oy",  
    "SUPPLIER": {  
        "company name": "Karkki Oy",  
        "contact name": "Anne Heikonen",  
        "contact title": "Product Manager",  
        "address": "Valtakatu 12",  
        ...  
        "supplies": [  
            {  
                "PRODUCT": {  
                    "product name": "Maxilaku",  
                    "category name": "Confections",  
                    ...  
                }  
            },  
            {  
                "PRODUCT": {  
                    "product name": "Valkoinen suklaa",  
                    "category name": "Confections",  
                    ...  
                }  
            }  
        ]  
    }  
}
```

SUPPLIER

# MongoDB Aggregation Framework

## Outline

Aggregation framework ? What is it ?

Sample database

Aggregation operators

\$project

\$match

\$limit, \$skip and \$sample

\$sort and \$count

\$unwind

\$group

\$out

\$lookup

# \$project

Operator `$project` extracts components of subdocuments, rename components, and performs operations on components

Select company name of each customer and skip document identifier

```
aggregate($project)
db.orders.aggregate([ {$project:{"CUSTOMER.company name":1,"_id":0}} ])
```

The results

```
{ "CUSTOMER" : { "company name" : "Alfreds Futterkiste" } }
{ "CUSTOMER" : { "company name" : "Ana Trujillo Emparedados y helados" } }
{ "CUSTOMER" : { "company name" : "Antonio Moreno Taquería" } }
...
...
...
```

Select company name of each customer and skip document identifier and **remove nestings**

```
aggregate($project)
db.orders.aggregate([ {$project:{"Company":"$CUSTOMER.company name","_id":0}} ])
```

The results

```
{ "Company" : "Alfreds Futterkiste" }
{ "Company" : "Ana Trujillo Emparedados y helados" }
{ "Company" : "Antonio Moreno Taquería" }
...
...
...
```

# \$project

"\$keyname" syntax is used to refer to a value associated with a key "keyname" in the aggregation framework

Select customer addresses and remove nestings

```
aggregate($project)
db.orders.aggregate([ {$project:{"Customer address": "$CUSTOMER.address", "_id": 0}} ])
```

```
{ "Customer address" : "Obere Str. 57" }
{ "Customer address" : "Avda. de la Constitución 2222" }
{ "Customer address" : "Mataderos 2312" }
...
...
...
```

The results

Select a name of each customer and concatenate it with its code

```
aggregate($project)
db.orders.aggregate([ {$project:{"Customer address":{$concat:[ "$CUSTOMER.address",
                    "-",
                    "$CUSTOMER.customer code"]}} }])
```

```
{ "_id" : "ALFKI", "Customer address" : "Johan Strauss Str. 23-ALFKI" }
{ "_id" : "ANATR", "Customer address" : "Avda. de la Constitución 2222-ANATR" }
{ "_id" : "ANTON", "Customer address" : "Mataderos 2312-ANTON" }
...
...
...
```

The results

[TOP](#)

# MongoDB Aggregation Framework

## Outline

Aggregation framework ? What is it ?

Sample database

Aggregation operators

\$project

\$match

\$limit, \$skip and \$sample

\$sort and \$count

\$unwind

\$group

\$out

\$lookup

# \$match

Operator **\$match** selects the documents that satisfy a given condition

Find suppliers located in a city **Sandvika**

```
db.orders.aggregate([{$match:{"SUPPLIER.city":"Sandvika"} }])
```

aggregate(\$match)

Find suppliers located in a city **Sandvika** and display company name, city, and the names of product supplied

```
db.orders.aggregate([{$match:{ "SUPPLIER.city": "Sandvika" }},  
{$project:{ "SUPPLIER.company_name": 1, "SUPPLIER.city": 1,  
"SUPPLIER.supplies.PRODUCT.product_name": 1, "_id": 0 }}])
```

aggregate(\$match,\$project)

Find suppliers located in **Germany** supplying a product **Rossle Sauerkraut** and display company name, city, and the names of product supplied

```
db.orders.aggregate([{$match:{ "SUPPLIER.country": "Germany" }},  
{$match:{ "SUPPLIER.supplies.PRODUCT.product_name": "Rossle Sauerkraut" }},  
{$project:{ "SUPPLIER.company_name": 1, "SUPPLIER.city": 1,  
"SUPPLIER.supplies.PRODUCT.product_name": 1, "_id": 0 }}])
```

aggregate(\$match,\$match,\$project)

# \$match

Find suppliers located in a city **Sandvika** and display company name, city, and the names of product supplied

Incorrect implementation of projection

```
aggregate($match,$project)
db.orders.aggregate([{$match:{"SUPPLIER.city":"Sandvika"}},
                     {$project:{"SUPPLIER.company_name":1}},
                     {$project:{"SUPPLIER.city":1}},
                     {$project:{"SUPPLIER.supplies.PRODUCT.product_name":1,"_id":0}}])
```

The results

```
{ "SUPPLIER" : { } }
```

# MongoDB Aggregation Framework

## Outline

Aggregation framework ? What is it ?

Sample database

Aggregation operators

\$project

\$match

\$limit, \$skip and \$sample

\$sort and \$count

\$unwind

\$group

\$out

\$lookup

# \$limit, \$skip and \$sample

Operation **\$limit** passes a given number of documents through a pipeline

Operation **\$skip** eliminates a given number of documents from a pipeline

Operation **\$sample** randomly picks a given number of documents from a pipeline

Find the first two suppliers

```
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},  
{$limit:2}])
```

aggregate(\$match,\$limit)

Find all suppliers except the first two

```
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},  
{$skip:2}])
```

aggregate(\$match,\$skip)

# \$limit, \$skip and \$sample

Find the third and the fourth supplier

```
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},  
                     {$limit:4},  
                     {$skip:2}])
```

aggregate(\$match,\$limit,\$skip)

```
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},  
                     {$skip:2},  
                     {$limit:2}])
```

aggregate(\$match,\$skip,\$limit)

Find all customers located in France and list the sample 2 documents

```
db.orders.aggregate([{$match:{"CUSTOMER.country":"France"}},  
                     {$sample:{size:2}}])
```

aggregate(\$match,\$sample)

# MongoDB Aggregation Framework

## Outline

Aggregation framework ? What is it ?

Sample database

Aggregation operators

\$project

\$match

\$limit, \$skip and \$sample

\$sort and \$count

\$unwind

\$group

\$out

\$lookup

# \$count

Operation **\$count** counts the total number of documents in a pipeline

List the total number of documents in a collection **orders**

```
aggregate($count)
db.orders.aggregate([{$count:"Total documents"}])
{ "Total documents" : 117 }
```

A result of counting

Find the total number of suppliers located in Canada

```
aggregate($match,$count)
db.orders.aggregate([{$match:{"SUPPLIER.country":"Canada"}},
{$count:"Total suppliers in Canada"}])
{ "Total suppliers in Canada" : 2 }
```

A result of counting

# \$sort

Operation **\$sort** sorts the documents

Display the names of companies of all suppliers located in **Canada** and sort the names in ascending order

```
aggregate($match,$project,$sort)
db.orders.aggregate([{$match:{"SUPPLIER.country":"Canada"}},
                     {$project:{"SUPPLIER.company name":1,"_id":0}},
                     {$sort:{"SUPPLIER.company name":1}}])
```

Why the following solution is worse than the one above ?

```
aggregate($sort,$match,$project)
db.orders.aggregate([{$sort:{"SUPPLIER.company name":1}},
                     {$match:{"SUPPLIER.country":"Canada"}},
                     {$project:{"SUPPLIER.company name":1,"_id":0}} ])
```

Find a company name and postal code of supplier that has the largest postal code

```
aggregate($match,$project,$sort)
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$project:{"SUPPLIER.company name":1,"SUPPLIER.postal code":1,"_id":0}},
                     {$sort:{"SUPPLIER.postal code":-1}},
                     {$limit:1}]))
```

# MongoDB Aggregation Framework

## Outline

Aggregation framework ? What is it ?

Sample database

Aggregation operators

\$project

\$match

\$limit, \$skip and \$sample

\$sort and \$count

\$unwind

\$group

\$out

\$lookup

# \$unwind

Operation **\$unwind** creates a separate document for each element of a given array

A document is replicated for each element of an array, i.e. an array is **unnested**

List the names of products supplied by the first 2 suppliers

```
aggregate($match,$limit,$project)
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$limit:2},
                     {$project:{"SUPPLIER.supplies.PRODUCT.product name":1,"_id":0}}])
```

The results)

```
{ "SUPPLIER" : { "supplies" : [ { "PRODUCT" : { "product name" : "Côte de Blaye" } } ] } }
{ "SUPPLIER" : { "supplies" : [ { "PRODUCT" : { "product name" : "Sasquatch Ale" } },
                               { "PRODUCT" : { "product name" : "Steeleye Stout" } },
                               { "PRODUCT" : { "product name" : "Laughing Lumberjack Lager" } } ] } }
```

# \$unwind

List the names of products supplied by the first 2 suppliers

```
aggregate($match,$limit,$project)
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$limit:2},
                     {$project:{"product":"$SUPPLIER.supplies.PRODUCT.product name","_id":0}}])
```

The results

```
{ "product" : [ "Côte de Blaye" ] }
{ "product" : [ "Sasquatch Ale", "Steeleye Stout", "Laughing Lumberjack Lager" ] }
```

```
aggregate($match,$limit,$unwind,$project)
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$limit:2},
                     {$unwind:"$SUPPLIER.supplies"},
                     {$project:{"product":"$SUPPLIER.supplies.PRODUCT.product name","_id":0}}])
```

The results

```
{ "product" : "Côte de Blaye" }
{ "product" : "Sasquatch Ale" }
{ "product" : "Steeleye Stout" }
{ "product" : "Laughing Lumberjack Lager" }
```

# \$unwind

List the names of suppliers (companies) and the names of product supplied by the first 2 suppliers

```
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
    {$limit:2},
    {$project:{"supplier":"$SUPPLIER.company name",
        "product":"$SUPPLIER.supplies.PRODUCT.product name",
        "_id":0}}])
```

aggregate(\$match,\$limit,\$unwind,\$project)

```
{ "supplier" : "Aux joyeux ecclesiastiques", "product" : [ "Côte de Blaye" ] }
{ "supplier" : "Bigfoot Breweries", "product" : [ "Sasquatch Ale", "Steeleye Stout", "Laughing Lumberjack Lager" ] }
```

The results

```
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
    {$limit:2},
    {$unwind:"$SUPPLIER.supplies"},
    {$project:{"supplier":"$SUPPLIER.company name",
        "product":"$SUPPLIER.supplies.PRODUCT.product name",
        "_id":0}}])
```

aggregate(\$match,\$limit,\$unwind,\$project)

```
{ "supplier" : "Aux joyeux ecclesiastiques", "product" : "Côte de Blaye" }
{ "supplier" : "Bigfoot Breweries", "product" : "Sasquatch Ale" }
{ "supplier" : "Bigfoot Breweries", "product" : "Steeleye Stout" }
{ "supplier" : "Bigfoot Breweries", "product" : "Laughing Lumberjack Lager" }
```

The results

# \$unwind

List the total number of products supplied by the first 2 suppliers

```
aggregate($match,$limit,$unwind,$project,$count)
db.orders.aggregate([{$match:{'$SUPPLIER':{$exists:true}}},
                     {$limit:2},
                     {$unwind:"$SUPPLIER.supplies"},
                     {$project:{'product':'$SUPPLIER.supplies.PRODUCT.product name','_id':0}},
                     {$count:"Total number of products supplied by the first 2 suppliers"}])
{ "Total number of products supplied by the first 2 suppliers" : 4 }
```

The results

# MongoDB Aggregation Framework

## Outline

Aggregation framework ? What is it ?

Sample database

Aggregation operators

\$project

\$match

\$limit, \$skip and \$sample

\$sort and \$count

\$unwind

\$group

\$out

\$lookup

# \$group

Operation **\$group** groups the documents and applies the aggregation functions to each group

Find the total number of suppliers

```
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$count:"total"}])
```

aggregate(\$match,\$count)

```
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$group:{"_id": null,"total":{$sum:1}}}] )
```

aggregate(\$match,\$group)

Find the names of countries, the suppliers come from

```
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$project:{"country":"$SUPPLIER.country","_id":0}}])
```

aggregate(\$match,\$project)

Find the **distinct** names of countries, the suppliers come from

```
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$project:{"country":"$SUPPLIER.country","_id":0}},
                     {$group:{"_id":"$country"} }])
```

aggregate(\$match,\$project\$, \$group)

# \$group

Find distinct country names together with the total number of supplier per country

```
aggregate($match,$group)
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$group:{"_id":"$SUPPLIER.country","total":{$sum:1}}}] )
```

The results

```
{ "_id" : "Italy", "total" : 2 }
{ "_id" : "Sweden", "total" : 1 }
{ "_id" : "Germany", "total" : 3 }
...
...
```

Find distinct country names together with the total number of supplier per country

```
aggregate($match,$project,$group)
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$project:{"country":"$SUPPLIER.country","_id":0}},
                     {$group:{"_id":"$country","total":{$sum:1}}}] )
```

The results

```
{ "_id" : "Italy", "total" : 2 }
{ "_id" : "Sweden", "total" : 1 }
{ "_id" : "Germany", "total" : 3 }
...
...
```

# \$group

Find the total number of products supplied by each supplier

```
aggregate($match,$unwind,$project)
```

```
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},  
                    {$unwind:"$SUPPLIER.supplies"},  
                    {$project:{"supplier":"$SUPPLIER.company name",  
                               "product":"$SUPPLIER.supplies.PRODUCT.product name",  
                               "_id":0}} ])
```

```
{ "supplier" : "Aux joyeux ecclesiastiques", "product" : "Côte de Blaye" }  
{ "supplier" : "Bigfoot Breweries", "product" : "Sasquatch Ale" }  
{ "supplier" : "Bigfoot Breweries", "product" : "Steeleye Stout" }  
{ "supplier" : "Bigfoot Breweries", "product" : "Laughing Lumberjack Lager" }  
...  
...
```

The results

```
aggregate($match,$unwind,$project,$group)
```

```
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},  
                    {$unwind:"$SUPPLIER.supplies"},  
                    {$project:{"supplier":"$SUPPLIER.company name",  
                               "product":"$SUPPLIER.supplies.PRODUCT.product name",  
                               "_id":0}},  
                    {$group:{"_id":"$supplier","total":{$sum:1}}}] )
```

```
{ "_id" : "Aux joyeux ecclesiastiques", "total" : 1 }  
{ "_id" : "Bigfoot Breweries", "total" : 3 }  
...  
...
```

The results

# \$group

Find the total prices of products per supplier

```
aggregate($match,$unwind,$project)

db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$unwind:"$SUPPLIER.supplies"},
                     {$project:{"supplier":"$SUPPLIER.company name",
                               "price":"$SUPPLIER.supplies.PRODUCT.unit price",
                               "_id":0}} ])
```

```
{ "supplier" : "Aux joyeux ecclesiastiques", "price" : 263.5 }
{ "supplier" : "Bigfoot Breweries", "price" : 14 }
{ "supplier" : "Bigfoot Breweries", "price" : 18 }
{ "supplier" : "Bigfoot Breweries", "price" : 14 }

...
```

The results

```
aggregate($match,$unwind,$project,$group)

db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$unwind:"$SUPPLIER.supplies"},
                     {$project:{"supplier":"$SUPPLIER.company name",
                               "product":"$SUPPLIER.supplies.PRODUCT.product name",
                               "price":"$SUPPLIER.supplies.PRODUCT.unit price",
                               "_id":0}},
                     {"$group:{"_id":"$supplier","total":{$sum:"$price"}}}])
```

```
{ "_id" : "Aux joyeux ecclesiastiques", "total" : 263.5 }
{ "_id" : "Bigfoot Breweries", "total" : 46 }

...
```

The results

[TOP](#)

# \$group

Find the averages of all prices of products per supplier

```
aggregate($match,$unwind,$project,$group)
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$unwind:"$SUPPLIER.supplies"},
                     {$project:{"supplier":"$SUPPLIER.company name",
                               "price":"$SUPPLIER.supplies.PRODUCT.unit price",
                               "_id":0}},
                     {$group:{"_id":"$supplier","average":{$avg:"$price}}}]})
```

The results

```
{ "_id" : "Aux joyeux ecclesiastiques", "average" : 263.5 }
{ "_id" : "Bigfoot Breweries", "average" : 15.33333333333334}
...
...
...
```

# \$group

Find the minimum and maximum prices of products per supplier

```
aggregate($match,$unwind,$project,$group)
db.orders.aggregate([{$match:{"SUPPLIER":{$exists:true}}},
                     {$unwind:"$SUPPLIER.supplies"},
                     {$project:{"supplier":"$SUPPLIER.company name",
                               "price":"$SUPPLIER.supplies.PRODUCT.unit price",
                               "_id":0}},
                     {$group:{"_id":"$supplier","min":{$min:"$price"}, "max":{$max:"$price"}}}])
```

The results

```
{ "_id" : "Aux joyeux ecclesiastiques", "min" : 263.5, "max" : 263.5 }
{ "_id" : "Bigfoot Breweries", "min" : 14, "max" : 18 }
...
...
...
```

# MongoDB Aggregation Framework

## Outline

Aggregation framework ? What is it ?

Sample database

Aggregation operators

\$project

\$match

\$limit, \$skip and \$sample

\$sort and \$count

\$unwind

\$group

\$out

\$lookup

# \$out

Operation **\$out** saves the results of processing in a collection

Find the customer code and the names of products purchased by a customer with a customer code **AROUT** and save the result in a collection **"purchases"**

```
aggregate($match,$project)
db.orders.aggregate([{$match:{"CUSTOMER.customer code":"AROUT"}},
                     {$project:{"Code":"$CUSTOMER.customer code",
                               "Purchased":"$CUSTOMER.submits.Orden.consists of.Orden Detail.product name"} } ])
{ "_id" : "AROUT", "Code" : "AROUT", "Purchased" : [
    [
        "Gorgonzola Telino",
        "Grandma's Boysenberry Spread",
        "Konbu",
        "Mozzarella di Giovanni",
        "Tofu"
    ]
]}
```

The results

# \$out

Find the customer code and the names of products purchased by a customer with a customer code **AROUT** and save the result in a collection **"purchases"**

```
aggregate($match,$project,$unwind)
db.orders.aggregate([{$match:{"CUSTOMER.customer code":"AROUT"}},
                     {$project:{"Code":"$CUSTOMER.customer code",
                               "Purchased":"$CUSTOMER.submits.ORDER.consists of. ORDER DETAIL.product name"},},
                     {$unwind:"$Purchased"}])

{ "_id" : "AROUT", "Code" : "AROUT", "Purchased" : [
    "Gorgonzola Telino",
    "Grandma's Boysenberry Spread",
    "Konbu",
    "Mozzarella di Giovanni",
    "Tofu"
]}
The results
```

# \$out

Find the customer code and the names of products purchased by a customer with a customer code **AROUT** and save the result in a collection **"purchases"**

```
aggregate($match,$project,$unwind,$unwind)
db.orders.aggregate([{$match:{"CUSTOMER.customer code":"AROUT"}},
                     {$project:{"Code":"$CUSTOMER.customer code",
                               "Purchased":"$CUSTOMER.submits.ORDER.consists of.ORDER DETAIL.product name"}},
                     {$unwind:"$Purchased"},
                     {$unwind:"$Purchased"}])
```

The results

```
{ "_id" : "AROUT", "Code" : "AROUT", "Purchased" : "Gorgonzola Telino" }
{ "_id" : "AROUT", "Code" : "AROUT", "Purchased" : "Grandma's Boysenberry Spread" }
{ "_id" : "AROUT", "Code" : "AROUT", "Purchased" : "Konbu" }
{ "_id" : "AROUT", "Code" : "AROUT", "Purchased" : "Mozzarella di Giovanni" }
{ "_id" : "AROUT", "Code" : "AROUT", "Purchased" : "Tofu" }
```

\$match,\$project,\$unwind,\$unwind,\$out

```
db.orders.aggregate([{$match:{"CUSTOMER.customer code":"AROUT"}},
                     {$project:{"Code":"$CUSTOMER.customer code",
                               "Purchased":"$CUSTOMER.submits.ORDER.consists of.ORDER DETAIL.product name"}},
                     {$unwind:"$Purchased"},
                     {$unwind:"$Purchased"},
                     {$project:{"_id":0}},
                     {"$out:"purchases"}])
```

find()

```
db.purchases.find().pretty()
```

&lt;

## \$out

Find the names, categories, and unit prices of all products supplied and save the result in a collection "**products**"

```
aggregate($project)
db.orders.aggregate([{$project:{"product":"$SUPPLIER.supplies.PRODUCT",
                                "_id":0 }} ])
```

The results

```
{ "product" : [ { "product name" : "Chef Anton's Cajun Seasoning",
                  "category name" : "Condiments",
                  "quantity per unit" : "48 - 6 oz jars",
                  "unit price" : 25,
                  "units in stock" : 53,
                  "units on order" : 0,
                  "reorder level" : 0,
                  "discontinued" : "N"
                },
                ...
                ...
              ]}
```

aggregate(\$project,\$unwind)

```
db.orders.aggregate([{$project:{"product":"$SUPPLIER.supplies.PRODUCT",
                                "_id":0 }},
                      {$unwind:"$product"} ])
```

[TOP](#)

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

40/46



# \$out

Find the names, categories, and unit prices of all products supplied and save the result in a collection "**products**"

```
aggregate($project,$unwind,$project)
db.orders.aggregate([{$project:{"product":"$SUPPLIER.supplies.PRODUCT",
    "_id":0 }},
    {$unwind:"$product"},
    {$project:{"name":"$product.product name",
        "category":"$product.category name",
        "price":"$product.unit price"} } ])
```

The results

```
{ "name" : "Côte de Blaye", "category" : "Beverages", "price" : 263.5 }
{ "name" : "Sasquatch Ale", "category" : "Beverages", "price" : 14 }
{ "name" : "Steeleye Stout", "category" : "Beverages", "price" : 18 }
...
...
```

```
aggregate($project,$unwind,$project,$out)
```

```
db.orders.aggregate([{$project:{"product":"$SUPPLIER.supplies.PRODUCT",
    "_id":0 }},
    {$unwind:"$product"},
    {$project:{"name":"$product.product name",
        "category":"$product.category name",
        "price":"$product.unit price"} },
    {$out:"products"} ])
```

TOP

```
db.products.find().pretty()
```

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

find() 41/46

# MongoDB Aggregation Framework

## Outline

Aggregation framework ? What is it ?

Sample database

Aggregation operators

\$project

\$match

\$limit, \$skip and \$sample

\$sort and \$count

\$unwind

\$group

\$out

\$lookup

# \$lookup

Operation `$lookup` performs a left outer join to an unsharded collection in the same database to filter in documents from the "joined" collection for processing

A collection `purchases` created earlier

```
{ "_id" : ObjectId("61342804f26c5a766e0e1c6f"),
  "Code" : "AROUT", "Purchased" : "Gorgonzola Telino" }
{ "_id" : ObjectId("61342804f26c5a766e0e1c70"),
  "Code" : "AROUT", "Purchased" : "Grandma's Boysenberry Spread" }
...
...
...
```

A collection `purchases`

A collection `products` created earlier

```
{ "_id" : ObjectId("61342773f26c5a766e0e1c1a"),
  "name" : "Gorgonzola Telino", "category" : "Dairy Products", "price" : 12.5 }
{ "_id" : ObjectId("61342773f26c5a766e0e1c22"),
  "name" : "Grandma's Boysenberry Spread", "category" : "Condiments", "price" : 25 }
...
...
...
```

A collection `products`

# \$lookup

A left outer join of a collection `purchases` with a collection `products` over a condition `purchases.Purchased = products.name`

```
aggregate($lookup)  
db.purchases.aggregate([{$lookup: {from:"products",  
    localField:"Purchased",  
    foreignField:"name",  
    as:"result"}}, ])  
  
purchases left outer join products  
  
{ "_id" : ObjectId("61342804f26c5a766e0e1c6f"),  
  "Code" : "AROUT", "Purchased" : "Gorgonzola Telino",  
  "result" : [ {_id" : ObjectId("61342773f26c5a766e0e1c1a"),  
    "name" : "Gorgonzola Telino",  
    "category" : "Dairy Products",  
    "price" : 12.5}  
  ]}  
}  
{ "_id" : ObjectId("61342804f26c5a766e0e1c70"),  
  "Code" : "AROUT", "Purchased" : "Grandma's Boysenberry Spread",  
  "result" : [ {_id" : ObjectId("61342773f26c5a766e0e1c22"),  
    "name" : "Grandma's Boysenberry Spread",  
    "category" : "Condiments",  
    "price" : 25}  
  ]}  
}  
...  ...  ...
```

# \$lookup

A left outer join of a collection `products` with a collection `purchases` over a condition `products.name = purchases.Purchased`

```
db.products.aggregate([{$lookup: {from:"purchases",
    localField:"name",
    foreignField:"Purchased",
    as:"result"} }])
```

aggregate(\$lookup)

products	left outer join purchases
<pre>{"_id" : ObjectId("61342773f26c5a766e0e1c22"), "name" : "Grandma's Boysenberry Spread", "category" : "Condiments", "price" : 25,</pre>	
<pre>"result" : [{"_id" : ObjectId("61342804f26c5a766e0e1c70"), "Code" : "AROUT", "Purchased" : "Grandma's Boysenberry Spread"}]</pre>	
}	
<pre>{"_id" : ObjectId("61342773f26c5a766e0e1c1a"), "name" : "Gorgonzola Telino", "category" : "Dairy Products", "price" : 12.5,</pre>	
<pre>"result" : [{"_id" : ObjectId("61342804f26c5a766e0e1c6f"), "Code" : "AROUT", "Purchased" : "Gorgonzola Telino"}]</pre>	
}	
...	...
<pre> {"_id" : ObjectId("61342773f26c5a766e0e1c0e"), "name" : "Côte de Blaye", "category" : "Beverages", "price" : 263.5,</pre>	
<pre> "result" : [ ]</pre>	
}	
<pre> {"_id" : ObjectId("61342773f26c5a766e0e1c0f"), "name" : "Sasquatch Ale", "category" : "Beverages", "price" : 14,</pre>	
<pre> "result" : [ ]</pre>	
...	...

# References

## [MongoDB Manual, Aggregation](#)

Banker K., Bakkum P., Verch S., Garret D., Hawkins T., MongoDB in Action, 2nd ed., Manning Publishers, 2016

Chodorow K. MongoDB The Definitive Guide, O'Reilly, 2013

# CSCI235 Database Systems

## MongoDB Indexing

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Indexing

## Outline

[Overview of indexing](#)

[Single key index](#)

[Compound key index](#)

[Sparse index](#)

[Multikey index](#)

[Hashed index](#)

[Geospatial index](#)

[Index administration](#)

# Overview of indexing

Indexes significantly reduce amount of time needed to access the documents

Without indexes all the documents in a collection must be accessed

Single-key index is the most appropriate for `{"key": "value"}` query conditions

For query conditions over multiple keys, for example

`{$and: [ {"key1": "value1"}, {"key2": "value2"} ]}`

compound index is the best option

If we have a compound index on `(key1, key2)` then the second index on `key1` is not really needed, however it may still speed up an access a bit

If we have a compound index on `(key1, key2)` then the second index on `key2` speeds up access a lot

An order of keys in a compound index is very important

# Indexing

## Outline

[Overview of indexing](#)

[Single key index](#)

[Compound key index](#)

[Sparse index](#)

[Multikey index](#)

[Hashed index](#)

[Geospatial index](#)

[Index administration](#)

# Single key indexes

An index on "`_id`" is an automatically created single-key index

Equality search over the values of "`_id`" is the fastest possible search

The following command creates a **single key unique index** on a key `code`

```
db.department.createIndex( {"code": 1}, {"unique":true} )
```

createIndex()

The index is **unique** because it enforces uniqueness of the values associated with key `code`, i.e. each document in a collection has a different value associated with a key `code`

An attempt to insert two documents with the same value of key `code` fails enforcing a key constraint

Unique index should be created before inserting any data

A unique index cannot be created on a collection where duplicate keys exist

# Single key indexes

The following command creates a **single key nonunique index** on a key **budget**

```
db.department.createIndex( {"budget": 1}, {"unique": false} )
```

createIndex()

# Indexing

## Outline

[Overview of indexing](#)

[Single key index](#)

[Compound key index](#)

[Sparse index](#)

[Multikey index](#)

[Hashed index](#)

[Geospatial index](#)

[Index administration](#)

# Compound key index

The following commands create the **single key nonunique indexes** on the keys **budget** and **total\_staff\_number**

```
db.department.createIndex( {"budget":1}, {"unique":false} )
```

createIndex()

```
db.department.createIndex( {"total_staff_number":1}, {"unique":false} )
```

createIndex()

A query like

```
db.department.find({ "budget":2000, :"total_staff_number":5})
```

createIndex()

uses only one of the indexes created above

A query optimizer picks the more efficient index (with higher **selectivity**)

To use both indexes we can traverse each index separately and calculate intersection of disk locations found

# Compound key index

The following commands create a **compound key nonunique index** on the keys **budget** and **total\_staff\_number**

```
db.department.createIndex( {"budget":1, "total_staff_number":1}, {"unique":false} )
```

createIndex()

A compound index is a single index where each entry is composed of more than one key

A compound index is used by a query

```
db.department.find({"budget":2000, "total_staff_number":5})
```

find()

# Indexing

## Outline

[Overview of indexing](#)

[Single key index](#)

[Compound key index](#)

[Sparse index](#)

[Multikey index](#)

[Hashed index](#)

[Geospatial index](#)

[Index administration](#)

# Sparse index

MongoDB indexes are **dense** by default

In a **dense index** for every document there is an index key even the document lacks a key

Then there exists a null entry in an index and it is possible to use an index for a query like

```
db.department.find( {"budget":null} )
```

find()

**Dense index** is inconvenient when:

- unique index on a field that doesn't appear in every document in a collection is needed
- a large number of documents in a collection have no indexed key

In a sparse index, only documents that have a value for the indexed key are indexed

```
db.department.createIndex( {"total_staff_number":1},  
                           {"unique":false, "sparse":true} )
```

createIndex()

TOP

# Indexing

## Outline

[Overview of indexing](#)

[Single key index](#)

[Compound key index](#)

[Sparse index](#)

[Multikey index](#)

[Hashed index](#)

[Geospatial index](#)

[Index administration](#)

# Multikey index

In **multikey index** multiple entries in the index reference the same document

**Multikey index** is useful for indexing fields whose values are arrays

```
db.department.createIndex( {"course.code":1} )
```

createIndex()

Each value in this **courses.code** array will appear in the index

A query on any array values can use the index to locate the document

# Indexing

## Outline

[Overview of indexing](#)

[Single key index](#)

[Compound key index](#)

[Sparse index](#)

[Multikey index](#)

[Hashed index](#)

[Geospatial index](#)

[Index administration](#)

# Hashed index

In **hashed index** the keys become the arguments of a **hash function** and a results of of **hash function** determine location of a document in a hash bucket

The hashed values will determine the ordering of the documents

Indexing

```
db.department.createIndex( {"name": "hashed"} )
```

Hashed indexes have the following restrictions:

- equality queries can be processed with an index
- range queries cannot use hashed index
- multikey hashed indexes are not allowed
- floating-point values are cast to an integer before being hashed; **1.4** and **1.5** will have the same value in hashed index

**Hashed indexes** are used for **sharding**

# Indexing

## Outline

[Overview of indexing](#)

[Single key index](#)

[Compound key index](#)

[Sparse index](#)

[Multikey index](#)

[Hashed index](#)

[Geospatial index](#)

[Index administration](#)

# Geospatial index

**Geospatial index** allows to find the documents that are close to a given location, based on latitude and longitude values stored in each document

**Geospatial index** can be used to efficiently calculate geographic distances, including the curvature of the earth

MongoDB supports different kinds of indexes, however, only the first two types of indexes listed below can be combined to a **compound index**

- 1: **Ascending B\*-tree index**
- -1: **Descending B\*-tree index**
- "hashed": **Hashtable index**; very fast for lookup by exact value, especially in very large collections. But it is not usable for inexact queries ("`$gt`", "`$regex`" or similar)
- "text": **Text index** designed for searching for words in strings with natural language
- "2d": **Geospatial index** on a flat plane
- "2dsphere": **Geospatial index** on a sphere

# Indexing

## Outline

[Overview of indexing](#)

[Single key index](#)

[Compound key index](#)

[Sparse index](#)

[Multikey index](#)

[Hashed index](#)

[Geospatial index](#)

[Index administration](#)

# Index administration

## Listing the indexes

```
db.department.getIndexes()
```

[  
  {  
    "**v**" : 2,  
    "**key**" : {  
      "**\_id**" : 1  
    },  
    "**name**" : "**\_id\_**",  
    "**ns**" : "test.department"  
  }  
]

find()

Results

# Index administration

## Creating an index

```
createIndex()
```

```
db.department.createIndex( {"name":"hashed"} )
```

```
{
```

```
    "createdCollectionAutomatically" : false,
```

```
    "numIndexesBefore" : 1,
```

```
    "numIndexesAfter" : 2,
```

```
    "ok" : 1
```

```
}
```

```
Results
```

# Index administration

## Listing the indexes

```
getIndexes()  
db.department.getIndexes()  
  
Results  
[  
  {  
    "v" : 2,  
    "key" : {  
      "_id" : 1  
    },  
    "name" : "_id_",  
    "ns" : "test.department"  
  },  
  {  
    "v" : 2,  
    "key" : {  
      "name" : "hashed"  
    },  
    "name" : "name_hashed",  
    "ns" : "test.department"  
  }  
]
```

# Index administration

Delete an index `name_hashed`

The screenshot shows a MongoDB shell interface with the following code and results:

```
db.department.dropIndex("name_hashed")
```

dropIndex()

```
db.department.getIndexes()
```

getIndexes()

```
[{"v": 2, "key": {"_id": 1}, "name": "_id_", "ns": "test.department"}]
```

Results

The code `db.department.getIndexes()` is highlighted in blue, indicating it is the command being run.

# Index administration

Creation of indexes before loading data allows indexes to be built incrementally as the data is inserted

Creation of an index on an already loaded collection may take a long time

It is possible to create an index in the background without closing a database system

```
createIndex()  
db.department.createIndex( {"total_staff_number":1}, {"background":true} )
```

It is possible to create an index in offline by taking a replica node offline, building an index, and taking node online allowing the node to catch up with master replica node

When ready we can promote a node to primary and take another secondary node offline, etc.

# Index administration

It is possible to re-build indexes in order to defragment them after a lot of updates

```
reIndex()  
db.department.reIndex()  
  
Results  
{  
    "nIndexesWas" : 1,  
    "nIndexes" : 1,  
    "indexes" : [  
        {  
            "v" : 2,  
            "key" : {  
                "_id" : 1  
            },  
            "name" : "_id_",  
            "ns" : "test.department"  
        }  
    ],  
    "ok" : 1  
}
```

# References

Banker K., Bakkum P., Verch S., Garret D., Hawkins T., MongoDB in Action, 2nd ed., Manning Publishers, 2016

MongoDB Manual, Indexes <https://docs.mongodb.com/manual/indexes/>

Chodorow K. MongoDB The Definitive Guide, O'Reilly, 2013

# CSCI235 Database Systems

## MongoDB Replication

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Replication

## Outline

### Basics

Why replication ?

Why no-replication ?

### Experiment

How does replication work ?

Commit and rollback

# Basics

**Replication** means automatic maintenance of data distributed over a number of MongoDB servers

**Replication** is implemented as a mechanism called **replica sets**

A **replica sets** is a group of nodes are configured such that they can automatically synchronize their data and failover when a node disappears

Older versions of MongoDB support a method of replication called **master-slave** (now considered as deprecated) that still can be used in MongoDB v3.0

In both approaches, a single **primary node** receives all writes, and then all **secondary nodes** read and apply those writes to themselves asynchronously

**Replica sets** use the same replication mechanism as **master-slave** with additionally ensuring automated failover

# Basics

In **replica-sets** approach if the **primary node** goes offline then whenever it is possible one of the **secondary nodes** is automatically promoted to be **primary node**

Additionally, **replica sets** provide the improvements to the previous replication method like easier recovery and more sophisticated deployment topologies

In a **replica set** approach data is not considered as committed until it is written to a majority of member nodes, i.e. more than 50% of the servers

It means that if a **replica set** has only two servers then no server can be down

If the **primary node** in a **replica set** fails before it replicates its data, other members will continue accepting writes, and any not replicated data must be rolled back, meaning it can no longer be read

# Replication

## Outline

Basics

Why replication ?

Why no-replication ?

Experiment

How does replication work ?

Commit and rollback

# Why replication ?

**Replication** provides safety mechanisms protecting database system from environment failures like:

- a network connection between the application and the database is lost,
- there is a loss of power,
- persistent storage device (HDD, SSD) fails

In addition to protecting against external failures, replication is important for **system durability**

When running without backup/journaling the original values of corrupted data cannot be easily restored

**Replication** can always guarantee a clean copy of the data files if a single node shuts down due to a hardware fault

**Replication** also facilitates redundancy, failover, maintenance, and load balancing

**Replication** is designed primarily for redundancy

# Why replication ?

It ensures that the contents of replicated nodes are synchronised with the primary node

The **replicas** can be located in the same place as a **primary node** or at different location connected with a **primary node** over wide area network

**Replication** is asynchronous and because of that any sort of network latency or partition between nodes have no impact on performance of the **primary node**

The modifications of the contents of replicated nodes can be delayed by a constant number of seconds, minutes, or even hours behind the primary node

It gives a chance to "move back in time" if the contents of the **primary node** is corrupted

For example, if someone accidentally drops a wrong collection of objects and a replica is "delayed in time" then it is possible to restore from a **replica node**

# Why replication ?

A delayed **replica** gives an administrator time to react and restore data

Another application of replication is **failover**, i.e. a situation when the **primary node** fails and one of the redundant nodes takes a role of the **primary node**

In MongoDB such "switch" is performed automatically.

Replication simplifies maintenance by allowing the high workload operations to be done on a node other than the primary at production time

For example, it is possible to run backups on a **secondary node** to avoid unnecessary additional load on the primary node and to avoid downtime

Replication allows for building large indexes on a **secondary node** simultaneously with the operations on the **primary node**

# Why replication ?

Then it is possible to swap the **secondary node** with the existing **primary** and then build the same index again on the new **secondary**

**Replication** allows to balance read operations across many replicas

Data can be simultaneously read from many separate replicas; it is the easiest (and a bit simplistic) way to scale up the system

# Replication

## Outline

Basics

Why replication ?

Why no-replication ?

Experiment

How does replication work ?

Commit and rollback

# Why no-replication ?

Replication does not improve performance a lot when:

- hardware can't process the given workload; if performance becomes constrained by the number of I/O operations per second (IOPS) persistent storage can handle then reading from a replica increases total IOPS; if writes are occurring at the same time then it consumes all IOPS ( sharding is a better option)
- the ratio of writes to reads exceeds 50%; then every write to the primary must eventually be written to all the secondary nodes as well and directing the additional reads to secondary nodes slows down replication
- an application requires consistent reads; then secondary nodes replicate asynchronously and therefore they are not guaranteed to reflect the latest writes to the primary node

# Replication

## Outline

Basics

Why replication ?

Why no-replication ?

Experiment

How does replication work ?

Commit and rollback

# Experiment

We use replica sets to create two replica nodes simulated by two Mongo processes at the ports **4000** and **4001**

Start **Terminal** and process the following shell commands

```
cd ~  
mkdir node0  
mkdir node1
```

Command shell

In **Terminal** window start **mongod** server with data located in **node0**, listening to a port **4000** and attached to a replica set "**rs0**"

```
mongod --dbpath node0 --port 4000 --replSet "rs0"
```

Command shell

Minimize **Terminal** window (do not close it !)

Open another **Terminal** window and start mongod server with data located in **node1** listening to a port **4001** and attached to the same replica set "**rs0**"

```
mongod --dbpath node1 --port 4001 --replSet "rs0"
```

Command shell

# Experiment

Minimize **Terminal** window (do not close it !)

Start **Terminal** and then start **mongo** client and connect to a server listening at a port **4000**

```
mongo --port 4000
```

Command shell

At **mongo** client prompt > process the following commands

```
rs.initiate()
```

Initiate()

Reply is such that the current node is **SECONDARY**

```
rs.conf()
```

Initiate()

The current node became **PRIMARY**

Add **mongod** server at **4001** as a secondary node

```
rs.add("localhost:4001")
```

add()

# Experiment

Check status

```
rs.status()
```

status()

Open yet another Terminal window and connect to mongod server at 4001

```
mongo --port 4001
```

Command shell

Make the node slave

```
rs.slaveOk()
```

slaveOk()

# Experiment

Now, to test replication set we shall create a new collection and we shall insert a new document at mongo server at a port 4000; replication mechanism supposed to copy insertion to mongod server at a port 4001

Return to a window where mongo client is connected to mongod server at a port 4000 and process the following commands

```
use mydb
db.names.insert({"full-name":"James Bond"})
db.names.find()
```

port 4000

Move to a window where mongo client is connected to mongod server at a port 4001 and execute the commands

```
use mydb
db.names.find()
```

port 4001

# Experiment

An attempt to insert a new document by a client connected to `mongod` server at a port `4001` fails due to slave mode of the server

To shutdown both replication servers execute in both windows the following statements

```
use admin  
db.shutdownServer()
```

shutdown

After replication set has been created and after shutdown it is possible to restart it in the following way

In `Terminal` window start `mongod` server with data located in `node0`, listening to a port `4000` and attached to a replica set "`rs0`"

```
mongod --dbpath node0 --port 4000 --replSet "rs0"
```

Command shell

# Experiment

Minimize **Terminal** window (do not close it !)

Open another **Terminal** window and start mongod server with data located in **node1** listening to a port **4001** and attached to the same replica set

```
mongod --dbpath node1 --port 4001 --replSet "rs0"
```

Command shell

Minimize **Terminal** window (do not close it !)

Open yet another **Terminal** window and connect to **mongod** server at **4001**

```
mongo --port 4001
```

Command shell

Make the node slave

```
rs.slaveOk()
```

slaveOk()

# Experiment

Start **Terminal** and then start **mongo** client and connect to a server listening at a port **4000**

```
mongo --port 4000
```

Command shell

Return to a window where **mongo** client is connected to **mongod** server at a port **4000** and process the following commands

```
use mydb
db.names.insert({"full-name":"Harry Potter"})
db.names.find()
```

port 4000

Move to a window where **mongo** client is connected to **mongod** server at a port **4001** and execute the commands

```
use mydb
db.names.find()
```

port 4001

# Replication

## Outline

Basics

Why replication ?

Why no-replication ?

Experiment

How does replication work ?

Commit and rollback

# How does replication work ?

Replica sets rely on two basic mechanisms: an **oplog** and a **heartbeat**

**Oplog (operation log)** is a space restricted collection that lives in a database called **local** on every replica node and records all changes to the data

Every time a client writes to the primary node, an entry with enough information to reproduce the write is automatically added to the primary node's **oplog**.

Once the write is replicated to a given secondary node then its **oplog** also stores a record of the write

When a given secondary node is ready to update itself, it performs the following actions:

- first, it looks at the timestamp of the latest entry in its own **oplog**
- next, it queries the primary node's **oplog** for all entries greater than that timestamp
- finally, it writes the data and adds each of those entries to its own **oplog**

# How does replication work ?

Then in a case of failover, any secondary promoted to primary will have an **oplog** that the other secondaries can replicate from; it enables replica set recovery

Secondary nodes use **long polling** to immediately apply new entries from the primary's oplog

**Long polling** means the secondary makes a long-lived request to the primary

When the primary receives a modification, it responds to the waiting request immediately

The secondary nodes will usually be almost completely up to date

If the secondary nodes fall behind because of network partitions or maintenance on secondaries, the latest timestamp in each secondary's oplog can be used to monitor any replication lag

The replica set **heartbeat** facilitates election and failover

# How does replication work ?

By default, each replica set member pings all the other members every two seconds

As long as every node remains healthy and responsive, the replica set continue its work

Every replica set wants to ensure that exactly one primary node exists at all times

With no majority, the primary demotes itself to a secondary

If the heartbeats fail due to some kind of network partition, the other nodes will still be online

If the arbiter and secondary are still up and able to see each other, then according to the rule of the majority, the remaining secondary will become a primary

# Replication

## Outline

Basics

Why replication ?

Why no-replication ?

Experiment

How does replication work ?

Commit and rollback

# Commit and rollback

Writes are not considered as committed until they are replicated to a majority of nodes

Operations on a single document are always atomic and operations that involve multiple documents are not atomic

Consider the following scenario:

- a series of writes to the primary node did not get replicated to the secondary node,
- primary node goes offline and the secondary is promoted to primary and new writes go to primary
- old primary comes back online as secondary and tries to replicate from the new primary
- old primary has a series of writes that don't exist in the new primary node **oplog**
- It triggers a **rollback**

**Rollback** reverses all writes that were never replicated to a majority of nodes

# Commit and rollback

The writes are removed from both the secondary's oplog and the collection where they reside

If a secondary node has registered a deleted document, the node will look for the deleted document in another replica and restore it to itself

The same is true for dropped collections and updated documents

The reverted writes are stored in the rollback subdirectory of the path in the relevant node

For each collection with rolled-back writes, a separate BSON file will be created the filename of which includes the time of the rollback

In the event that you need to restore the reverted documents, you can examine these BSON files using the `bsondump` utility and manually restore them, possibly using `mongorestore`

# References

Banker K., Bakkum P., Verch S., Garret D., Hawkins T., MongoDB in Action, 2nd ed., Manning Publishers, 2016

[MongoDB Manual, Replication](#)

# CSCI235 Database Systems

## MongoDB Sharding

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# Sharding

## Outline

### Basics

Why sharding ?

Architecture

Distribution

Experiment

Querying and indexing

Production

# Basics

**Sharding** is the process of partitioning a large dataset into smaller and more manageable pieces

A **shared nothing** architecture is a distributed computing architecture in which each computing node does not share data with other the nodes

In database systems it is called as **sharding (shared nothing)**

What to do when:

- large amount of data and greater read/write throughput demands make commodity database servers not sufficient ?
- the database servers are not be able to address enough RAM, or they might not have enough CPU cores, to process the workload efficiently ?
- due to large amount of data it is not practical to store and to manage backups on one disk or **RAID** storage (**Redundant Array of Inexpensive Disks**) ?

# Basics

A solution to these problems is to distribute a database and database processing across more than one server

The method for doing this in Mongo DB is called **sharding**.

**Sharding** makes a database system complex due to administrative and performance overhead

# Sharding

## Outline

[Basics](#)

[Why sharding ?](#)

[Architecture](#)

[Distribution](#)

[Experiment](#)

[Querying and indexing](#)

[Production](#)

# Why sharding ?

There are two main reasons to **shard**:

- storage distribution
- load distribution

If monitoring of storage capacity shows that at certain moment the database applications require more storage than it is available and adding more storage is impossible then **sharding** is the best option

**Mongodb monitoring** means running `db.stats()` and `db.collection.stats()` in the `mongo` shell to get the statistics about the storage usage of the current database and `collection` within it

**Load** means **CPU and RAM utilization, I/O bandwidth, network transmission** used by the requests from the clients => **response time**

If at certain moment **response time** does not match the client's expectations then it triggers a decision to **shard**

A decision to **shard** depends on network usage, disk usage, CPU usage, and RAM usage

# Sharding

## Outline

[Basics](#)

[Why sharding ?](#)

[Architecture](#)

[Distribution](#)

[Experiment](#)

[Querying and indexing](#)

[Production](#)

# Architecture

In MongoDB a sharded cluster consists of **shards**, **mongos routers**, and **config servers**

**Shards** store the application data

**mongos routers** or system administrators are connected directly to the shards

A **shard** is either a single **mongod server** or a **replica set** that stores a partition of the application data

Like an **unsharded deployment**, each **shard** can be a single node for **development and testing stage**

Each shard should be a **replica set** in **production** stage because it is able to provide automatic **replication** and failover mechanisms

**Shards** are the only places where the application data gets saved in a **sharded cluster**

# Architecture

It is possible to connect to an individual **shard** as to a single **computing node** or a **replica set**, then it is possible to see only a portion of the total data stored in entire **shard**

Because each **shard** contains only part of entire data it is necessary to route the operations to the appropriate **shards**

**mongos routers** cache the cluster **metadata** and use it to route operations to the correct shard or shards

**mongos routers** provide clients with a single point of contact with the cluster

**mongos** provide a view of a **sharded cluster** the same as a view an **unsharded one**

Because **mongos** processes are lightweight and nonpersistent they can be deployed on the same machines as the **application servers**

Then only one network step is required for requests to any given **shard**

# Architecture

As **mongos** processes are non-persistent then an additional process is needed to durably store the **metadata** needed to properly manage the cluster

**Config servers** persistently store **metadata** about the cluster, including which **shard** has what subset of the data

**Metadata** includes the global cluster configuration, the locations of each database, collection, and the particular ranges of data in a collection and a change log preserving a history of the migrations of data across **shards**

For example, every time a **mongos** process is started, it fetches a copy of the **metadata** from **config servers** to get a coherent view of the **shard** cluster

**Shard cluster** requires several **config servers** not deployed as a replica set

Write operations on a **config server** use a two-phase commit protocol to ensure the data consistency across the **config servers**

# Architecture

In production environment more than one **config server** must be used and all **config servers** must reside on the separate machines to provide redundancy

# Sharding

## Outline

[Basics](#)

[Why sharding ?](#)

[Architecture](#)

[Distribution](#)

[Experiment](#)

[Querying and indexing](#)

[Production](#)

# Distribution

There are four levels of data granularity in MongoDB:

- **Document**: the smallest unit of data in MongoDB; a **document** represents a single object in the system (like a row in a relational database)
- **Chunk**: a group of documents clustered by the values on a field; a **chunk** is a concept that exists only in sharded setups; a **chunk** is created by the logical grouping of documents based on their values for a key or set of keys, known as a **shard key**
- **Collection**: a named grouping of documents within a database; a **collection** allows users to separate a database into logical groupings that make sense for the application
- **Database**: a set of collections of documents; a combination of a database name and a collection name is unique throughout the system, and it is commonly referred to as the **namespace**

# Distribution

Data can be distributed in a **sharded cluster** in the following ways:

- at a level of an **entire database** where each database along with all its collections is put on its own shard
- at a level of **partitions** or **chunks** of a collection, where the documents within a collection itself are spread out over multiple shards based on values of a key or set of keys (**shard key**) in the documents

# Database Distribution

Each database in a sharded cluster is assigned to a different **shard**

A database itself is not sharded

It is some sort of **manual sharding (partitioning)**

MongoDB has nothing to do with how well data is **partitioned** and it is completely up to a user to decide which database is located into which **shard**

One example of a real application for database distribution is MongoDB as a service

In such implementation of **sharding** customers can pay for access to a single MongoDB database

# Collection Distribution

The second method is **sharding** an individual collection

It is an **automatic sharding** in which MongoDB itself makes all the partitioning decisions, without any direct intervention from the applications

For example, consider the following document:

```
{  
  "_id": ObjectId("4d6e9b89b600c2c196442c21"),  
  "filename": "spreadsheet-1",  
  "updated_at": ISODate("2011-03-02T19:22:54.845Z"),  
  "username": "banks",  
  "data": "raw document data"  
}
```

BSON

If all the documents in a collection have this format, and if we choose **"\_id"** key and the **"username"** key as a **shard key** then a pair of values associated with **"\_id"** and **"name"** in each document is used to determine what chunk the document belongs to

# Collection Distribution

Sharding in MongoDB is **range-based**

It means that each **chunk** represents a range of **shard keys**

To determine what **chunk** a document belongs to, MongoDB extracts the values for a **shard key** and then finds a **chunk** whose shard **key range** contains the given **shard key** values

# Sharding

## Outline

[Basics](#)

[Why sharding ?](#)

[Architecture](#)

[Distribution](#)

[Experiment](#)

[Querying and indexing](#)

[Production](#)

# Experiment

A process of setting up a sharded cluster consists of three steps:

- starting the `mongod` and `mongos` servers through spawning all the individual `mongod` and `mongos` processes that make up the cluster
- configuring the cluster through updating the configuration such that the replica sets are initialized and the shards are added to the cluster and the nodes are able to communicate with each other
- sharding collections such that it can be spread across multiple shards

Start [Terminal](#) and process the following shell commands to create **config server** (just one)

```
mkdir conf1  
mkdir conf2
```

Command shell

Process the following command in a new [Terminal](#) window

```
mongod --configsvr --repSet conf --dbpath conf1 --port 4001
```

Command shell

# Experiment

Process the following command in a new [Terminal](#) window

```
mongod --configsvr --repSet conf --dbpath conf2 --port 4002
```

Command shell

Process the following command in a new [Terminal](#) window

```
mongo --port 4001
rs.initiate()
rs.conf()
rs.add("localhost:4002")
rs.status()
exit
```

port 4001

```
mongo --port 4002
rs.slaveOk()
exit
```

port 4002

# Experiment

Process the following command in a new [Terminal](#) window to create data shards (two replication sets each one consisting of two servers)

```
mkdir data1-1  
mkdir data1-2
```

Command shell

Process the following command in a new [Terminal](#) window

```
mongod --shardsvr --repSet data1 --dbpath data1-1 --port 4003
```

Command shell

Process the following command in a new [Terminal](#) window

```
mongod --shardsvr --repSet data1 --dbpath data1-2 --port 4004
```

Command shell

# Experiment

Process the following command in a new [Terminal](#) window

```
mongo --port 4003
rs.initiate()
rs.conf()
rs.add("localhost:4004")
rs.status()
exit
```

port 4003

```
mongo --port 4004
rs.slaveOk()
exit
```

port 4003

Process the following command in a new [Terminal](#) window

```
mkdir data2-1
mkdir data2-2
```

Command shell

Process the following command in a new [Terminal](#) window

```
mongod --shardsvr --repSet data2 --dbpath data2-1 --port 4005
```

Command shell

# Experiment

Process the following command in a new [Terminal](#) window

```
mongod --shardsvr --repSet data2 --dbpath data2-2 --port 4006
```

Command shell

Process the following command in a new [Terminal](#) window

```
mongo --port 4005
rs.initiate()
rs.conf()
rs.add("localhost:4006")
rs.status()
exit
```

port 4005

```
mongo --port 4006
rs.slaveOk()
exit
```

port 4005

# Experiment

Process the following command in a new **Terminal** window to start **mongos**

```
mongos --configdb conf/localhost:4001,localhost:4002 --port 4000
```

Command shell

Process the following command in a new **Terminal** window to create shards on replica sets on ports 4003 and 4005

```
mongo --port 4000  
sh.addShard("data1/localhost:4003")  
sh.addShard("data2/localhost:4005")
```

Command shell

List the data shards

```
db.getSiblingDB("config").shards.find()
```

getSiblingDB()

# Experiment

## Enable sharding of a database

```
sh.enableSharding("test")
db.getSiblingDB("config").databases.find()
```

enableSharding()

## Insert a collection into shard

```
use test
db.testcol.insert({"fname":"James", "lname":"Bond", "shard-key":"000"})
db.testcol.insert({"fname":"James", "lname":"Bond", "shard-key":"001"})
db.testcol.insert({"fname":"James", "lname":"Bond", "shard-key":"002"})
db.testcol.insert({"fname":"James", "lname":"Bond", "shard-key":"003"})
db.testcol.insert({"fname":"James", "lname":"Bond", "shard-key":"004"})
db.testcol.insert({"fname":"James", "lname":"Bond", "shard-key":"005"})
db.testcol.insert({"fname":"James", "lname":"Bond", "shard-key":"006"})
db.testcol.insert({"fname":"James", "lname":"Bond", "shard-key":"007"})
```

insert()

## Create index on "shard-key"

```
db.testcol.createIndex({"shard-key":1})
```

createIndex()

# Experiment

## Shard a collection

```
sh.shardCollection("test.testcol", {"shard-key": 1})
```

shardCollection()

## Get statistics

```
db.test_collection.stats()
```

stats()

```
... ... ...
"shards" : {
  "data2" : {
    "ns" : "test.testcol",
    "size" : 592,
    "count" : 8,
    "avgObjSize" : 74,
    "storageSize" : 32768,
...
}
```

Statistics

# Experiment

Create large collection "test\_collection"

```
use test
var bulk = db.test_collection.initializeUnorderedBulkOp();
people = ["Marc", "Bill", "George", "Eliot", "Matt", "Trey", "Tracy",
"Greg", "Steve", "Kristina", "Katie", "Jeff"];
for(var i=0; i<1000000; i++){
    user_id = i;
    name = people[Math.floor(Math.random()*people.length)];
    number = Math.floor(Math.random()*10001);
    bulk.insert( { "user_id":user_id, "name":name, "number":number });
}
bulk.execute();
```

test\_collection

Create index on "user\_id"

```
db.test_collection.createIndex({user_id:1})
```

createIndex()

# Experiment

Shard a collection over "user\_id"

```
sh.shardCollection("test.test_collection", {"user_id":1})
```

shardCollection()

Get statistics

```
sh.shardCollection("test.test_collection", {"user_id":1})
```

shardCollection()

... ... ... ...

Statistics

```
"shards" : {  
    "data1" : {  
        "ns" : "test.test_collection",  
        "size" : 184478484,  
        "count" : 2604399,  
        "avgObjSize" : 70,  
        "storageSize" : 80326656,  
    }  
}
```

... ... ... ...

Statistics

```
...  
    "data2" : {  
        "ns" : "test.test_collection",  
        "size" : 135758751,  
        "count" : 1916602,  
        "avgObjSize" : 70,  
        "storageSize" : 67858432,  
    }  
}
```

# Experiment

## Find sharding status

```
sh.status()                                         status()
```

```
databases:                                         Status
```

```
{ "_id" : "config", "primary" : "config", "partitioned" : true }
```

```
    config.system.sessions
```

```
        shard key: { "_id" : 1 }
```

```
        unique: false
```

```
        balancing: true
```

```
        chunks:
```

```
            data1      1
```

```
            { "_id" : { "$minKey" : 1 } } -->>
```

```
            { "_id" : { "$maxKey" : 1 } } on : data1 Timestamp(1, 0)
```

```
...     ...     ...     ...
```

# Experiment

## Sharding status

```
test.test_collection
  shard key: { "user_id" : 1 }
  unique: false
  balancing: true
  chunks:
    data1    10
    data2     9
    { "user_id" : { "$minKey" : 1 } } --> { "user_id" : 0 } on : data2 Timestamp(8, 0)
    { "user_id" : 0 } --> { "user_id" : 166667 } on : data2 Timestamp(9, 0)
    { "user_id" : 166667 } --> { "user_id" : 289501 } on : data1 Timestamp(9, 1)
    { "user_id" : 289501 } --> { "user_id" : 414501 } on : data1 Timestamp(7, 5)
    { "user_id" : 414501 } --> { "user_id" : 479349 } on : data1 Timestamp(7, 6)
    { "user_id" : 479349 } --> { "user_id" : 604349 } on : data1 Timestamp(3, 0)
    { "user_id" : 604349 } --> { "user_id" : 789699 } on : data1 Timestamp(5, 0)
    { "user_id" : 789699 } --> { "user_id" : 958699 } on : data2 Timestamp(7, 1)
    { "user_id" : 958699 } --> { "user_id" : 1167398 } on : data2 Timestamp(3, 2)
    { "user_id" : 1167398 } --> { "user_id" : 1417399 } on : data2 Timestamp(3, 3)
    { "user_id" : 1417399 } --> { "user_id" : 1667400 } on : data2 Timestamp(3, 4)
    { "user_id" : 1667400 } --> { "user_id" : 1982999 } on : data2 Timestamp(3, 5)
    { "user_id" : 1982999 } --> { "user_id" : 2232999 } on : data1 Timestamp(5, 2)
    { "user_id" : 2232999 } --> { "user_id" : 2483000 } on : data1 Timestamp(5, 3)
    { "user_id" : 2483000 } --> { "user_id" : 2733001 } on : data1 Timestamp(5, 4)
    { "user_id" : 2733001 } --> { "user_id" : 3007999 } on : data1 Timestamp(5, 5)
    { "user_id" : 3007999 } --> { "user_id" : 3257999 } on : data2 Timestamp(6, 2)
    { "user_id" : 3257999 } --> { "user_id" : 3520999 } on : data2 Timestamp(6, 3)
    { "user_id" : 3520999 } --> { "user_id" : { "$maxKey" : 1 } } on : data1 Timestamp(7, 0)
```

Status

# Experiment

## Sharding status

```
test.testcol
  shard key: { "shard-key" : 1 }
  unique: false
  balancing: true
  chunks:
    data2   1
    { "shard-key" : { "$minKey" : 1 } } -->>
    { "shard-key" : { "$maxKey" : 1 } } on : data2 Timestamp(1, 0)
```

# Sharding

## Outline

[Basics](#)

[Why sharding ?](#)

[Architecture](#)

[Distribution](#)

[Experiment](#)

[Querying and indexing](#)

[Production](#)

# Querying and indexing

From a database application perspective, there is no difference between querying a sharded cluster and querying a single unsharded database

In both cases, the query interface and the process of iterating over the result set are the same

How does it work inside the system ?

Config servers maintain a mapping of shard key ranges to shards

If a query includes a **shard key**, then mongos can quickly find which shard contains the query result set; it is called a **targeted query**

If the shard key is not part of the query then a query planner visits all shards to fulfill the query completely; it is called as a **global or scatter/gather query**

In **sharded** environment indexing is an important part of optimizing performance

# Querying and indexing

Indexing of a **sharded cluster** has the following properties:

- each shard maintains its own indexes
- when an index is created on a sharded collection, each shard builds a separate index for its portion of the collection
- it means that the sharded collections on each shard should have the same indexes, otherwise query performance is inconsistent
- sharded collections permit unique indexes on the "`_id`" field and on the shard key only
- unique indexes are prohibited elsewhere because enforcing them would require inter-shard communication

# Sharding

## Outline

[Basics](#)

[Why sharding ?](#)

[Architecture](#)

[Distribution](#)

[Experiment](#)

[Querying and indexing](#)

[Production](#)

# Production

Theoretically to launch the sample MongoDB shard cluster, you had to start a total of nine processes (three `mongods` for each replica set, plus three config servers)

In practice there is no need for so many processes

Replicated `mongods` are the most resource-intensive processes in a shard cluster and must be given their own machines

Replica set arbiters incur little overhead and they don't need their own servers.

Config servers store a relatively small amount of data

This means that config servers don't necessarily need their own machines, either

# Production

From what you already know about replica sets and shard clusters, you can construct a list of minimum deployment requirements:

- each member of a replica set, whether it's a complete replica or an arbiter, needs to live on a distinct machine
- every replicating replica set member needs its own machine.
- replica set arbiters are lightweight enough to share a machine with another process
- config servers can optionally share a machine; the only hard requirement is that all config servers in the config cluster reside on distinct machines

# References

Banker K., Bakkum P., Verch S., Garret D., Hawkins T., MongoDB in Action, 2nd ed., Manning Publishers, 2016

[MongoDB Manual, Sharding](#)

# CSCI235 Database Systems

# NoSQL, NewSQL, and Other Database Systems

Dr Janusz R. Getta

School of Computing and Information Technology -  
University of Wollongong

# NoSQL, NewSQL, and Other Database Systems

## Outline

XML Database Systems

Key-value Stores and Document Database Systems

Graph Database Systems

Column Stores

Extensible Record Stores

In-Memory Database Systems

Data Stream Processing Systems

NewSQL Database Systems

# XML Database Systems

**Native XML database systems:** XML document management systems

- [eXistDB](#): Java-based open source XML database; supports indexing, XPath, XQuery, XML:DBI (Java API)
- [BaseX](#): Java-based system that stores XML documents in a tabular representation; support indexing, conversion to JSON, XML:DB API

**XML support in relational a database systems:** columns of type [XMLType](#), e.g. ANSI SQL definition ([SQL/XML](#)) and Oracle, Postgres, DB2, MySQL, and SQL Server

# NoSQL, NewSQL, and Other Database Systems

## Outline

[XML Database Systems](#)

[Key-value Stores and Document Database Systems](#)

[Graph Database Systems](#)

[Column Stores](#)

[Extensible Record Stores](#)

[In-Memory Database Systems](#)

[Data Stream Processing Systems](#)

[NewSQL Database Systems](#)

# Key-value Stores and Document Database Systems

Key-value pair is a tuple of two strings ("key", "value")

A key-value store stores key-value pairs

A value is retrieved by specifying a key

A key-value store is a prototype of a schemaless database system

It is possible to put any key-value into a store and no restrictions are enforced on a format or structure of a value

A key-value store offers three operations: put, get, and delete

A value in key-value pair may contain other key-value pairs or sequences of keys

Dynamo DB is a key-value database implemented by Amazon

Redis is an advanced in-memory key-value store with command line interface; it supports several data types and data structures like strings, linked lists, unsorted sets, sorted sets, hash maps, bit arrays

# Key-value Stores and Document Database Systems

A **document database** stores and retrieves documents, which can be in JSON, BSON, etc format

Documents are self-describing, hierarchical data structures

The structures of documents are similar to each other but not exactly the same

Documents are stored as the value part of **key-value** store

```
{  
  "firstname": "James",  
  "likes": ["biking", "hiking", "viking"],  
  "city": "Boston",  
  "friend": null,  
  "addresses": [{"state": "NSW", "city": "Sydney"},  
                {"state": "Vic", "city": "Melbourne"}]  
}
```

JSON

**MongoDB** and **RavenDB** are the document database systems based on **BSON** storage data model

# NoSQL, NewSQL, and Other Database Systems

## Outline

[XML Database Systems](#)

[Key-value Stores and Document Database Systems](#)

[Graph Database Systems](#)

[Column Stores](#)

[Extensible Record Stores](#)

[In-Memory Database Systems](#)

[Data Stream Processing Systems](#)

[NewSQL Database Systems](#)

# Graph databases

Graph databases store entities and relationships between the entities

Nodes are instances of objects in the applications

Edges have properties and have directional significance

```
Node james = graphDb.createNode();
james.setProperty("name", "James");
Node harry = graphDb.createNode();
harry.setProperty("name", "Harry");
james.createRelationship(harry, FRIEND);
harry.createRelationship(james, FRIEND);
```

Graph data model

Graph databases ensure consistency through ACID-compliant transactions

Graph databases use domain specific languages for traversing graph structures

Properties of nodes can be indexed

A commonly used technique for horizontal scaling is sharding

Any link-rich domain is well suited for graph databases

[TOP](#)

Created by Janusz R. Getta, CSCI235 Database Systems, Spring 2021

8/25

# Graph databases

**Neo4j** is a Java-based graph database based on **Property Graph** data model

**Property Graph** data model provides a richer model for representing complex data than **RDF** (presented later) by associating both nodes and relationships with attributes

**Neo4j** supports billions of nodes, **ACID** compliant transactions, and multiversion consistency

**Neo4j** implements a declarative graph query language **Cypher** comparable to **SQL** or **SPARQL** (query language for **RDF**)

**Oracle Spatial and Graph**: it includes high performance, enterprise-scale, commercial spatial and graph database and analytics for Oracle Database 12c it supports large-scale Geographic Information Systems (**GIS**)

# NoSQL, NewSQL, and Other Database Systems

## Outline

XML Database Systems

Key-value Stores and Document Database Systems

Graph Database Systems

Column Stores

Extensible Record Stores

In-Memory Database Systems

Data Stream Processing Systems

NewSQL Database Systems

# Column stores

**Column store** (or columnar database management system) is a database management system (**DBMS**) that stores data tables by column rather than by row

Practical use of a column store versus a row store differs little in the relational DBMS world

To reduce overhead imposed DML operations columnar databases generally implement some form of **write-optimized delta store**

**Delta store** is optimized for frequent writes like tables stored in row-by-row format

**Delta store** is merged with the main columnar-oriented store

The merge will occur periodically, or whenever the amount of data in the delta store exceeds a threshold

Prior to the merge, queries access both the delta store and the column store in order to return complete and accurate results

# Column stores

KDB was the first commercially available column-oriented database developed in 1993 followed in 1995 by **Sybase IQ**

**Sybase IQ** (mid 1990) (“Intelligent Query”) data warehousing platform implemented relational tables in column-by-column mode

**C-Store** and later on **Vertica** (2005-2011) implemented a native columnar database

**MonetDB** was released under an open-source license in 2004

# NoSQL, NewSQL, and Other Database Systems

## Outline

XML Database Systems

Key-value Stores and Document Database Systems

Graph Database Systems

Column Stores

Extensible Record Stores

In-Memory Database Systems

Data Stream Processing Systems

NewSQL Database Systems

# Extensible Record Stores

Extensible Record Store organizes data into **extensible tables**

Extensible table consists of **rows**

Each **row** is uniquely identified by a **row key**

Data within a **row** is grouped by a **column family**

**Column families** have an important impact on the **physical implementation** of **extensible table**

Every **row** has the same **column families** although some of them can be **empty**

Data within a **column family** is addressed via its **column qualifier**, or simply, **column name**

Hence, a combination of **row key**, **column family**, and **column qualifier** uniquely identifies a **cell**

**Values in cells** do not have a data type and are always treated as sequences of bytes

# Extensible Record Stores

The table is lexicographically sorted on the row keys

Each cell has multiple versions, typically represented by the timestamp of when they were inserted into the table

Timestamp1      Timestamp2

Row Key	Column Family - Personal			Column Family - Office	
	Name	Residence	Phone	Phone	Address
00001	John	415-111-1234		415-212-5544	1021 Market St
00002	Paul	408-432-9922		415-212-5544	1021 Market St
00003	Ron	415-993-2124	415-212-5544		1021 Market St
00004	Rob	818-243-9988	408-998-4322		4455 Bird Ave
00005	Carly	206-221-9123	408-998-4325		4455 Bird Ave
00006	Scott	818-231-2566	650-443-2211	543 Dale Ave	

Cells

Values within a cell have multiple versions

Versions are identified by their version number, which by default is a timestamp when the cell was written

# Extensible Record Stores

Foundation of **Extensible Record Stores** is Google's system called **Big Table**

The other systems include

- **Apache Cassandra**: it stores column families in so called "keyspace"; creates table command creates a new column family in the keyspace; indexes can be created on columns
- **Apache HBase**: it stores tables in namespaces; it offers command line interface, SQL interface called **Phoenix**; and Java API

# NoSQL, NewSQL, and Other Database Systems

## Outline

[XML Database Systems](#)

[Key-value Stores and Document Database Systems](#)

[Graph Database Systems](#)

[Column Stores](#)

[Extensible Record Stores](#)

[In-Memory Database Systems](#)

[Data Stream Processing Systems](#)

[NewSQL Database Systems](#)

# In-Memory Database Systems

A more paradigm-shifting trend has been the increasing practicality of storing complete databases in **main memory** (RAM)

Architectural problems of main memory databases:

- **Cache-less architecture**: there is no point caching in memory what is already stored in memory
- **Alternative persistence model**: data in memory disappears when the power is turned off

In-memory databases either

- replicate data to other members of a cluster or
- write complete database images to disk files or
- write out transaction/operation records to an append-only disk file (called a transaction log or journal)

# In-Memory Database Systems

The most popular in-memory systems include:

- **TimesTen**: (owned by Oracle since 2005); it is entirely based on transactional relational database architecture; it can be used as a standalone in-memory database or as a caching database supplementing the traditional disk-based Oracle RDBMS; when the database is started, all data is loaded from checkpoint files into main memory; periodically or when required database data is written to checkpoint files
- **Redis**: (owned by VMware since 2013); it is an in-memory key-value store; it uses a [snapshot files to](#) store the copies of the entire system at a given point in time; [append only file](#) keeps a journal of changes that can be used to “roll forward” the database from a snapshot in the event of a failure; configuration options allow the users to configure writes to the [append files](#) after every operation, at one-second intervals, or based on operating-system-determined flush intervals
- **VoltDB**: it is a [hybrid](#) (in-memory/HDD) ACID-compliant relational database based on [shared nothing \(sharding\)](#) architecture; it supports SQL access from within pre-compiled Java stored procedures; it relies on horizontal partitioning down to the individual hardware thread to scale

# In-Memory Database Systems

The most popular in-memory systems include:

- **SAP HANA**: (owned by SAP, 2010) it is a relational database that combines in-memory technology with a columnar storage option, installed on an optimized hardware configuration; relational tables in **HANA** can be configured for row-oriented or columnar storage; on start-up, row-based tables and selected column tables are loaded from the database files and other column tables can be loaded on demand; reads and writes to row-oriented tables are applied directly; updates to column-based tables are buffered in the delta store; queries against column tables must read from both the delta store and the main column store
- **DB2 with BLU acceleration**: (owned by IBM) it is a hybrid relational database that stores frequently used data on **SSD**; it allows for better performance at a price closer to **HDD** based systems than purely **SDD** based systems, e.g. Oracle's Exadata database machine

# NoSQL, NewSQL, and Other Database Systems

## Outline

[XML Database Systems](#)

[Key-value Stores and Document Database Systems](#)

[Graph Database Systems](#)

[Column Stores](#)

[Extensible Record Stores](#)

[In-Memory Database Systems](#)

[Data Stream Processing Systems](#)

[NewSQL Database Systems](#)

# Data Stream Processing Systems

A **data stream** is an infinite sequence of transient values and when recorded persistent values

**Data streams** can be produced by sensor networks or by network traffic monitoring, stock exchange monitoring, etc

**Data Stream Management System (DSMS)** processes data streams by interminably running **continuous queries** over and over again

**DSMS** must handle the queries that might be running for days, months, or even years

**DSMS** must aggregate data from the entire stream, e.g. overall average of all values, or from a subset of data called as sliding window, e.g. an average of values in the last 30 days

Majority of **DSMS** are based on the relational data model and their continuous query languages are similar to SQL

In such a case data items from the streams are represented as a pair **(timestamp,tuple)** where tuple corresponds to a row in an infinite relational table

# NoSQL, NewSQL, and Other Database Systems

## Outline

[XML Database Systems](#)

[Key-value Stores and Document Database Systems](#)

[Graph Database Systems](#)

[Column Stores](#)

[Extensible Record Stores](#)

[In-Memory Database Systems](#)

[Data Stream Processing Systems](#)

[NewSQL Database Systems](#)

# NewSQL Database Systems

A term **NewSQL** describes adoption of the design principles underlying **NoSQL** systems to build new distributed relational database systems with SQL interface

Such re-design of conventional relational DBMSs would make them **scalable**, i.e. it would be possible to store relational data in a number of servers while efficiently processing SQL queries

Additionally, **NewSQL** systems suppose to transparently handle hardware failures in a network while still maintaining **ACID** compliant transaction protocols

The first type of **NewSQL** systems are completely new database platforms designed to operate in a distributed cluster of shared-nothing nodes, in which each node owns a subset of the data, e.g. **Google Spanner**, **CockroachDB**, **Clustrix**, **VoltDB**, **MemSQL**, **NuoDB**, and **Trafodion**

The second type are highly optimized storage engines for SQL that provide the same programming interface as SQL, but scale better than built-in engines, e.g. **MySQL Cluster**, **Infobright**, **TokuDB**, and **MyRocks**

# References

Wiese L. Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases, De Gruyter Oldenburg, 2015

Harrison G., Next Generation Databases NoSQL NewSQL Big Data, Apress, 2015