考点

Week1

- DML, DDL, DCL, TCL 分别有哪些

Week2

- ERD
    - 根据要求画图— TUT 有相关练习
    - 识别 ERD 中不符合要求的部分 — TUT 有相关练习
    - ERD 转换成 SQL — LEC13, QUIZ
        - The relationship with Key Constraint (thick arrow) does not need a table. We can use FK to represent it.

Week3

- CHAR VS VARCHAR
- 各种 sql syntax，包括如何根据要求创建 schema — TUT 有相关练习
    - ALTER Syntax
    - 包括 constraint
    - 包括识别哪些 insertion row 会违反 constraint
- 各种 key 的定义
- ERD 和 RM 的转换

Week4

- Plain Text 转换成 RA
- RA 和 sql 的转换 — TUT 有相关练习
- sql 和 RA 的转换

- 解释 sql 语句的含义 — TUT 有相关练习
- sql 语句互相替换 — TUT 有相关练习
    - 比如 IN 可以和 OR 在语义上进行替换
- 知道如何解释 RA in plain text — TUT 有相关练习
- 各种 JOIN (RA 里面的和 SQL 里面的都有)
- 熟悉 plain description 对应的 set operation
    - UNION, INTERSECT, EXCEPT
    - UNION ALL 等
    - SET OPERATION 对应的英文 — WEEK6 TUT Q2A
        - 比如 AND 一般用 intersect 实现 – quiz

Week5

- Know how to create/insert IC
    - Know the IC syntax — TUT
- know how to defer check for FK
- Assertion VS CHECK
- Know the trigger syntax — TUT
    - row VS statement
- Know the Assertion syntax
    - Exists
    - Not Exists

Week6

- 怎么用 set comparison operator 实现 SQL query — TUT

- View 的语法
- Aggregation — TUT
- SQL 语句执行顺序
- NULL and Three-valued Logic — TUT
    - COALESCE

Week8

- 需要知道如何识别出 FD
- 要知道怎么推导 super key 和 candidate key
- 要知道如何判断 super key 是不是 candidate key
- 判断是不是 loss-less decomposition + Dependency preservation
- 判断当前的 FDs 符合最高哪一层 Normal Form

Week9

- Transaction Syntax
- Identify update anomalies when ACID properties are not enforced
    - 即能够识别 3 种 Concurrent Access Issues，TUTQ2
- 知道 4 种事务隔离级
    - 分别能处理什么样的问题
    - 在不同时间段读取数据是什么样的 — TUT Q3
- 判断 conflict serializability
    - 一种 swap，
    - 一种画箭头
- 判断会不会 DEAD LOCK

Week11

- Physical storage medium
    - Permanent storage (external) — secondary storage
    - Transient storage (internal) — primary memory
    - Tertiary storage
- Disk (magnetic disk, secondary medium)
    - Sector, block, track. Cylinder
        - Blocking factor
    - Time for disk read/write
    - MTTF (Mean Time to Failure)
- Key approaches for physical disk-based transfer
    - Block transfer
    - Cylinder-based
    - Multiple disks — RAID
    - Disk scheduling — Elevator approach
    - Prefetching/double buffering
- which buffer strategy should we use?
    - LRU
    - MRU
- Calculating space and time — TUT
    - Records per page
    - Number of pages

    - Overhead rate
    - Fill factor
- Data Access Method
    - Unordered (Heap) Files — Linear Scan
    - Sorted Files — Binary Search
    - Indexes (B+ Tree) — Index Scan
- B+ Tree Index
    - 计算 level
    - 判断 composite search key 运用是否正确

Week12

- Query Processing
    - Parser
    - Optimizer
    - Plan Executor & Operator Evaluator
- Query Optimization
    - Heuristic-based Optimization — Logical
        - 使用 algebraic rules 简化 Relational Algebra Expression
    - Cost Estimate Optimization — Physical
- 比较不同的 Join implementation，看看哪个的 I/O 最小
    - Nested-loop
    - Block-nested
    - Index-nested

- External Merge Sort
- Query Execution
    - Materialization
    - Pipelining

Week1

DML: Data Management Language

- Select, insert, update, delete

DDL: Data Definition Language

- Create, Alter, Drop

DCL: Data Control Language

- Grant, Revoke

TCL：控制事务。

COMMIT、ROLLBACK、SAVEPOINT

Logical data independence:

protection of the applications from changes in the logical structure of the data

Physical data independence:

protection of the conceptual schema (and applications) from physical layout changes

Week2

- 需要能够甄别出 ERD 中的错误
- 能根据 plain text 画出对应的 ERD，重点关注 Constraint 部分
- 箭头从 Entity 指向 Relationship

Entity

Rectangle

Weak Entity (1:N relationship; Strong Entity 和 relation 之间不能有箭头，否则破坏了 1:N 关系; Weak Entity 和 relation 之间必须为粗箭头)

Double Diamond – weak relationship

Double Rectangle – Weak Entity

Dotted Underline – Discriminator (Partial Key), weak entity must use discriminator and strong entity's PK to Form a Composite PK

Attributes

Eclipse

Multi-value Attributes

Double Eclipse; 比如一个人可以有多个 Skill 或者电话号码

Composite Attributes

多画一条线连起来; 比如 address 还包含: street, city, state, zipcode

Derived Attribute

Dot Eclipse

Primary Key

Underline

Foreign Key

Does not show

Partial Key

Doted Underline

Relationship (不需要画 Key，因为用连接的 Entity PKs 作为组合主键)

Diamond

Aggregation Relationship

Dotted Box

Conveys a relationship between two relationships

Key Constraint (N-to-1)

Thin Arrow

At most one

Total/Partial Participation (N-to-M)

Total: Thick Line; at least one

Partial: Thin Line

Combine Key and Total Participation

Thick Arrow

Exactly One

Cardinality

用 label 在线旁边表示

* 表示无限大

0···1 用细线

1...3 用粗线

IsA

这里没有 thick arrow（即一个父类必须属于一个子类），但是等同于 thick line + Disjoint

没有 cardinality

Overlap Constraint

Disjoint: 只能属于最多一种子类

标注在线旁边

Overlapping (default): 可以属于多个子类

无需表示

Covering Constraint

Total: 父类必须属于子类

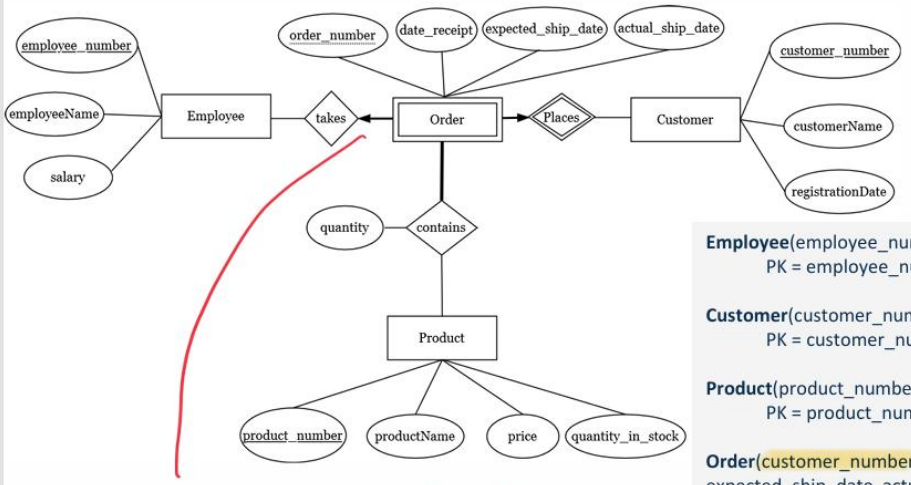
Thick Line

Partial (default): 父类可以不属于子类

Thin Line

TUT 2

Create the schema diagram equivalent to the following ER diagram



Employee(employee_number, employeeName, salary)
PK = employee_number

Customer(customer_number, customerName, registrationDate)
PK = customer_number

Product(product_number, productName, price, quantity_in_stock)
PK = product_number

Order(customer_number, order_number, date_receipt, expected_ship_date, actual_ship_date, employee_number)
PK = (customer_number, order_number)
FK = customer_number --> Customer
employee_number --> Employee

Contains(customer_number, order_number, product_number, quantity)
PK = (customer_number, order_number, product_number)
FK = (customer_number, order_number) --> Order
product_number --> Product

16

- Entity 都有 Table
- Relationship 再判断

- key constraint relationship does not need a table. we can use FK to replace it

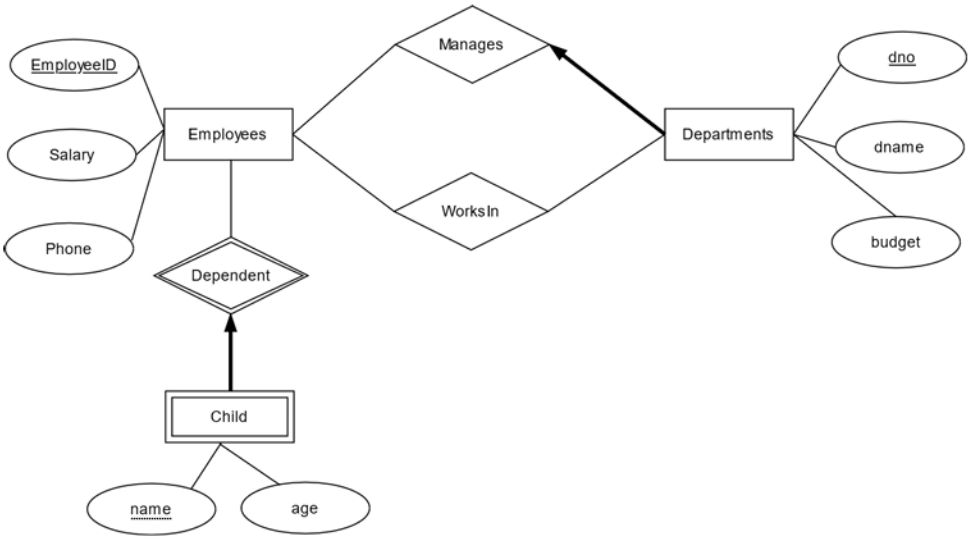
箭头指向的就是 FK 所引用的

没有箭头就表示当前 relationship 要分别引用 2 边 Entity

A company database needs to store information about employees (identified by employeeID, with salary and phone as attributes), departments (identified by dno, with dname and budget as attributes), and children of employees (with name and age as attributes).

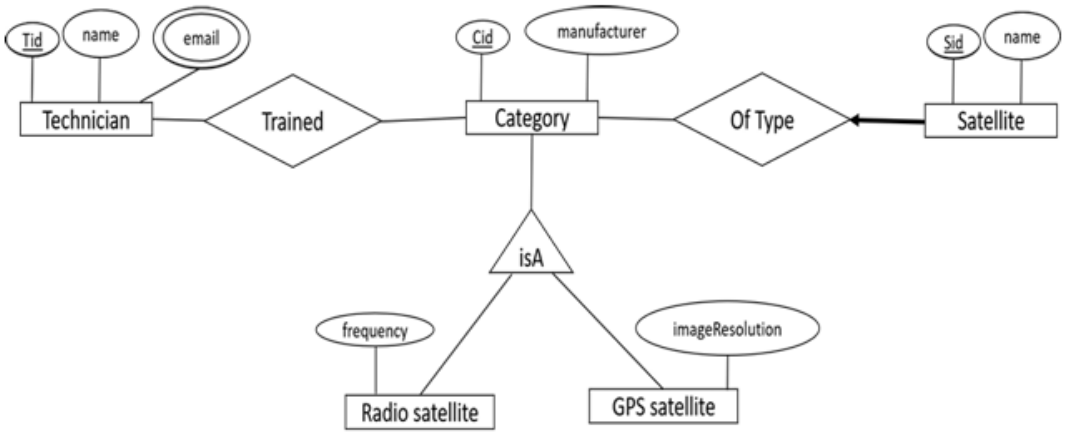
Employees work in departments; each department is managed by an employee; a child must be identified uniquely by name when the parent (who is an employee; assume that only one parent works for the company) is known. We are not interested in information about a child once the parent leaves the company.

Draw an ER diagram that captures this information.



Question 7: (4 marks)

This question is based on the following E-R diagram which describes the information kept on a satellite station.



a. Translate this E-R diagram into a relational model using the following **textual notation**.

Note: Each relation should be written in the form:

Name(attribute 1, attribute 2, …) PK=(attribute list), FK=(attribute list->parent relation),...

An 'attribute list' is one or more, comma-separated attribute names. Each relation must have a primary key (PK) defined. A relation can have zero or more FKs specified.

For example: Enrolment(studentId, courseId, mark) PK=(studentId, courseId),

FK=(studentId → Student, coursed → Course)

Do <u>not</u> write this relational model in SQL DDL syntax, use the above convention to describe your model.

b. Write SQL CREATE TABLE statements for the *Technician* and *Category* relation from your relational model, as well as any associated relationship relations between these two relations. You should show all relevant attributes, types and key constraints for these relations. You should include foreign key integrity constraints where suitable.

Solution:

a.

Technician(Tid, name)

PK=(Tid)

Email(Tid, email)

PK=(Tid, email)

FK=(Tid → Technician)

Category(cid, manufacturer)

PK=(cid)

RadioSatellite(cid, frequency)

PK=(cid)

FK=(cid → Category)

GPSSatellite(cid, imageResolution)

PK=(cid)

FK=(cid → Category)

Trained(cid, Tid)

PK=(cid, Tid)

FK=(cid → Category, Tid → Technician)

Satellite(sid, name, cid)

PK=(sid)

FK=(cid → Category)

b.

```
CREATE TABLE Technician( Tid INTEGER PRIMARY KEY,
                         name Varchar(50);

CREATE TABLE Email( Tid INTEGER,
                    email Varchar(100
                    PRIMARY KEY(Tid, email),
                    FOREIGN KEY(Tid) REFERENCES Technician);

CREATE TABLE Category( Cid INTEGER PRIMARY KEY,
                       manufacturer Varchar(50));

CREATE TABLE Trained( Tid INTEGER,
                      Cid INTEGER,
                      PRIMARY KEY(Tid, Cid),
                      FOREIGN KEY(Tid) REFERENCES Technician,
                      FOREIGN KEY(Cid) REFERENCES Category);
```

Week3

- 知道 RM 如何定义 FK constraint 才能满足 ERD，比如 FK 加上 NOT NULL 就能表示 exactly one

Real word Relation — RDBMS

- allow duplicated rows
- support ordering tuples and attributes
- allows “null”

Advantage/Disadvantage of NULL

Advantage:

- NULL can be useful because using an ordinary value with a specific meaning may not always work. 比如求 mean 的时候，如果我们用-1 代表 null 就会出错

Disadvantage:

- NULL may cause complications in the definition of many operations

CHAR VS VARCHAR

CHAR 是固定长度

VARCHAR 是动态长度

CREATE TABLE name**(…);**

DROP TABLE name **CASCADE**

ALTER TABLE

- **添加列**: ALTER TABLE Flight ADD test1 INTEGER, ADD test2 INTEGER; 不能同时添加 constraint
- **删除列**: ALTER TABLE Flight DROP test1;
- **添加 Constraint**: 只能先删除旧的外键约束，再添加一个新的外键约束
  - ALTER TABLE Orders ADD FOREIGN KEY (customer_id) REFERENCES Customers(customer_id);
  - ALTER TABLE Orders ADD UNIQUE (column_name);
- 删除 Constraint: 除了 **NOT NULL** 和 **DEFAULT** 以外，别的都需要对应的 **Constraint 名字**来删除
  - ALTER TABLE table_name DROP CONSTRAINT constraint_name;
  - ALTER TABLE table_name ALTER COLUMN column_name DROP NOT NULL;
- **重命名列**: ALTER TABLE table_name RENAME COLUMN old_column_name TO new_column_name;
- **重命名表**: ALTER TABLE old_table_name RENAME TO new_table_name;
- 修改 Default Value 或者 data type
  - ALTER TABLE table_name ALTER COLUMN column_name SET DEFAULT 7.77;
  - ALTER TABLE table_name ALTER COLUMN column_name TYPE new_data_type;

PRIMARY KEY（主键）

- **唯一性**：每个表只能有一个主键（At most one per table），并且主键字段的值必须是唯一的。
- **不允许 NULL 值**：主键字段不能包含空值（Automatically disallows NULL values）
- **Composite PK 可以自己设定**，但必须满足唯一性约束。

CANDIDATE KEY

- **唯一性**：候选键中的字段也必须是唯一的（all must be declared as UNIQUE）。
- **最小性（Minimality）**：没有多余的属性，可以删除任何一个属性而不再满足唯一性的条件。这里的最小是不是指 size，而是说不能去掉任何属性。
- **可以包含 NULL 值**

SUPER KEY

- **如果只满足唯一性，那么就叫做 Super Key**
- **所有 Candidate key 和 PK 都是 Super key**

FOREIGN KEY

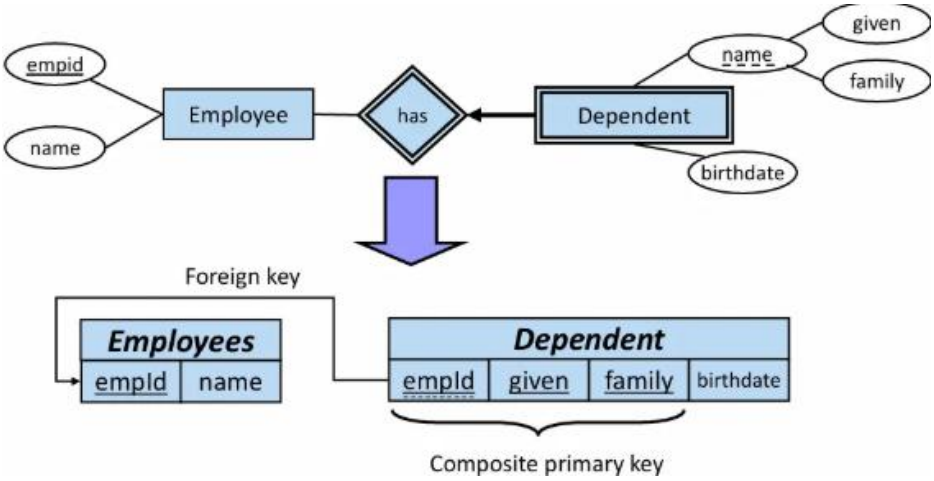
- **如默认允许 NULL 值**
- **If there must be a parent tuple, then must combine with NOT NULL constraint**

Mapping relationship types From ERD To RM

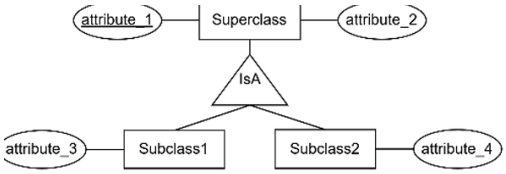
- 不包含 derived variable
- FK 的箭头应该指向被引用的主键
- 实线表示 PK，虚线表示 FK
- 可以有多条实线组成 Composite PK
- 一个 attribute 可以同时有虚线和实线，表示它同时属于 composite key 和 foreign key，通常出现在 Weak Entity 和 IsA relationship 中
- 在 IsA relationship 中，子类应该和父类共享 Primary Key

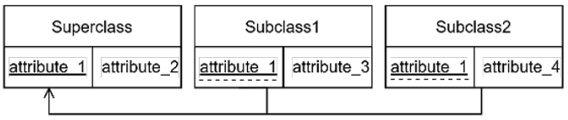
TUT 3

Weak Entity



IsA





Multivalued & Composite Attribute

IsA

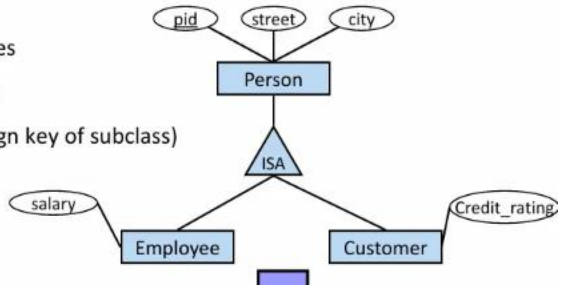
r superclass

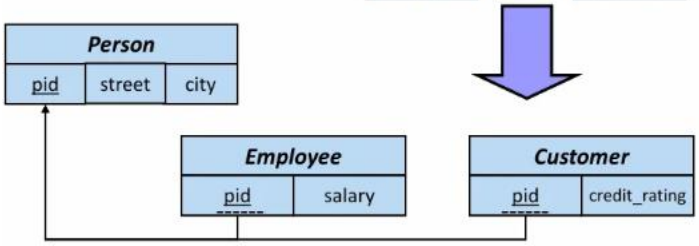
y key + common attributes

e tables for subclasses

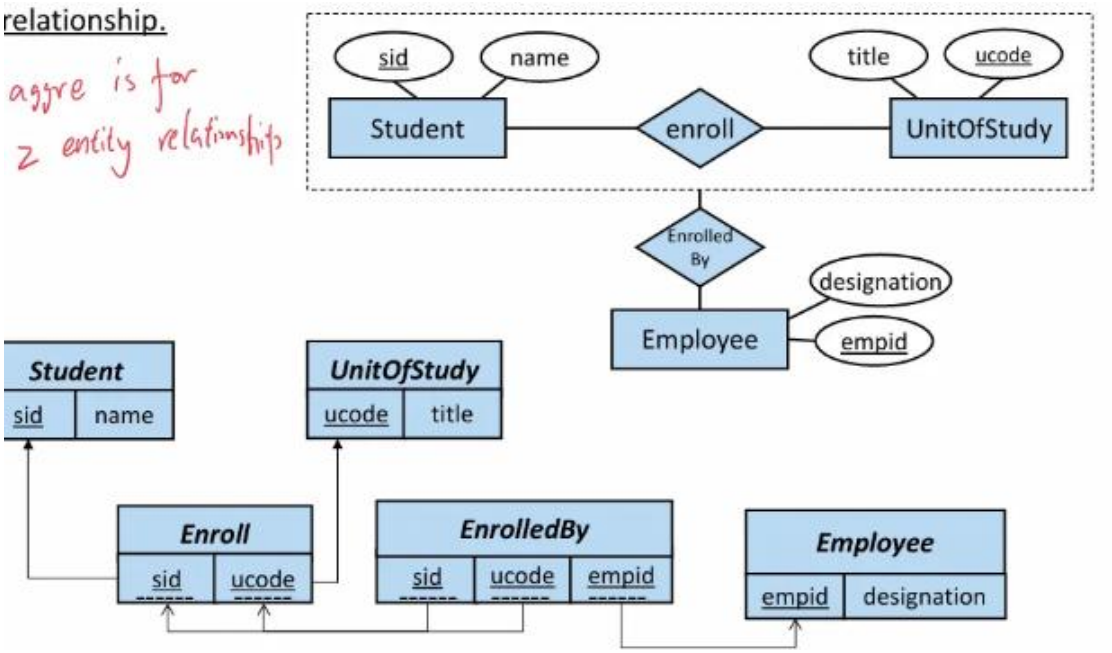
y key of superclass (foreign key of subclass)

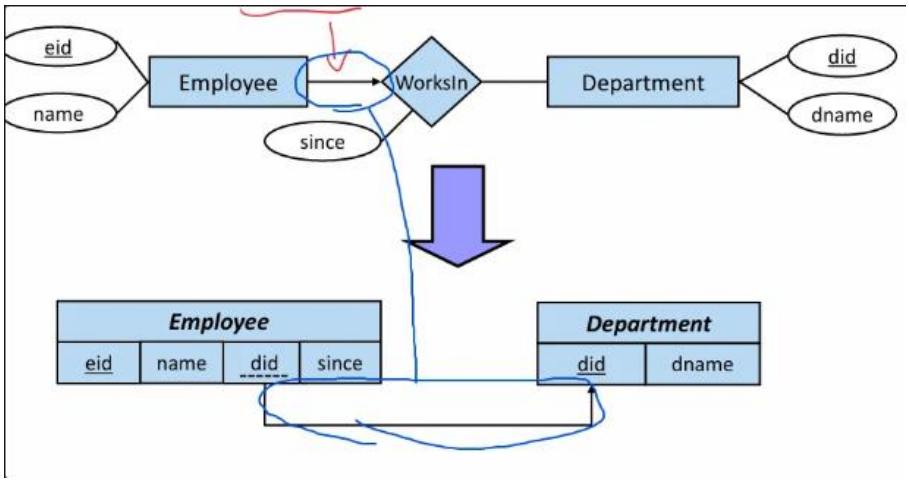
fic attributes





Aggregation



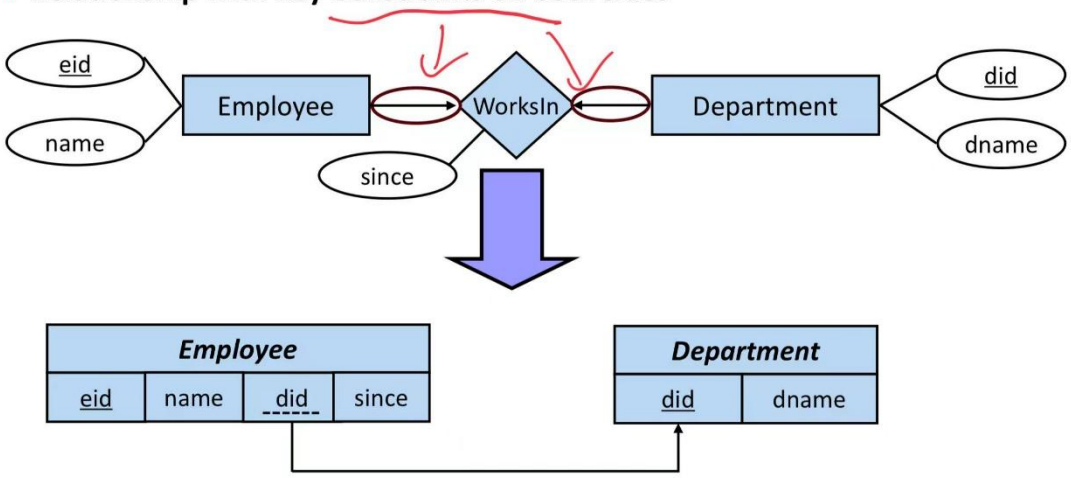


不需要设置 uniqueness for did

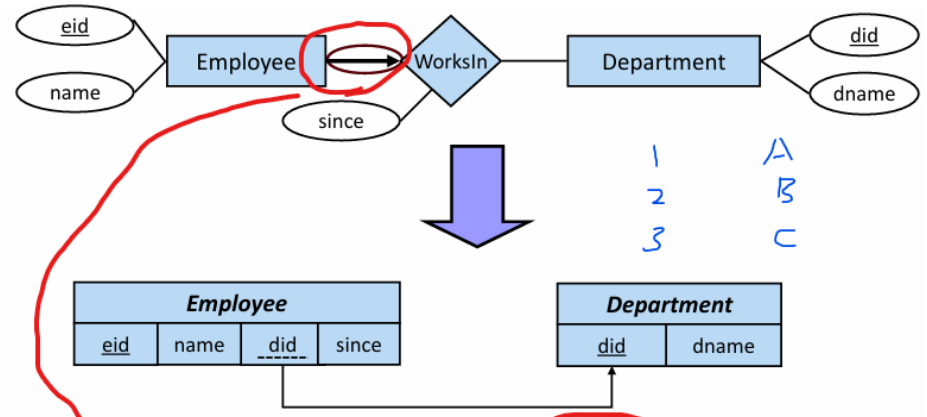
Employee works in at most 1 Department

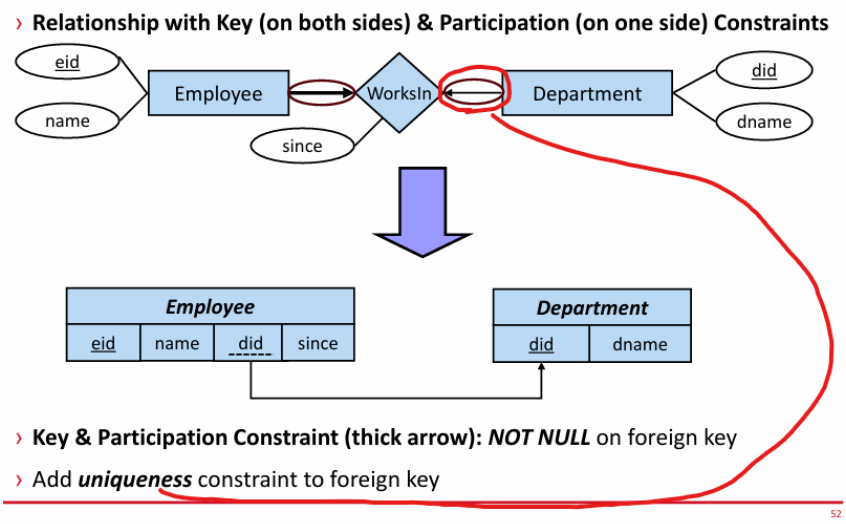
1 Department can have 0 to Many Employees

› **Relationship with Key Constraints on *both sides***



Add *uniqueness constraint* to foreign key did
Each Employee works in at most one Department
Each Department has at most one Employee





接下来看看哪些 INSERTION ROW 即使“违反”Constraint 也能插入成功

给出 Schema

```
CREATE TABLE Plane (
        plane_id VARCHAR(8) PRIMARY KEY,
        category VARCHAR(10) NOT NULL
                CHECK( category IN ('jet', 'turboprop')),
        capacity INTEGER NOT NULL
);
```

下面的需要注意

```
-- capacity should be integer (but still works):
INSERT INTO Plane VALUES ('ax-9999', 'turboprop', '50');
```

当你向一个 INTEGER 类型字段插入 '50' 这样的字符串文字时，它会自动尝试将其转换为整数

```
-- plane_id should be string (but still works):
INSERT INTO Plane VALUES (123456, 'jet', 255);
```

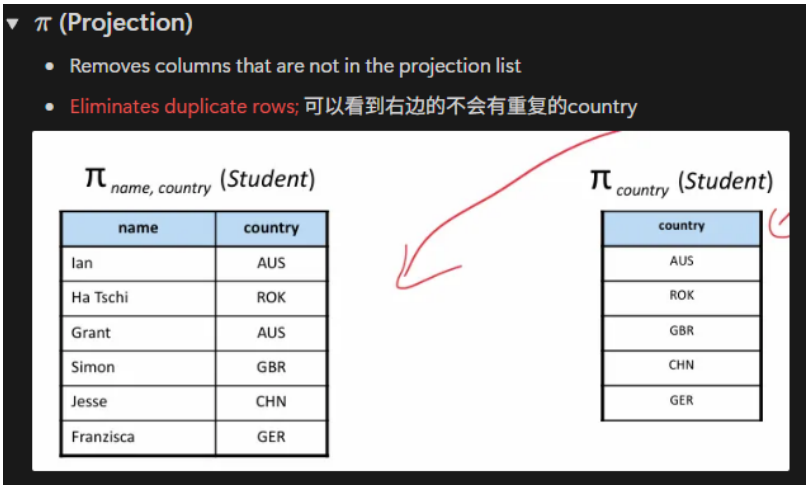
PostgreSQL 允许将 INTEGER 自动转换为 TEXT / VARCHAR

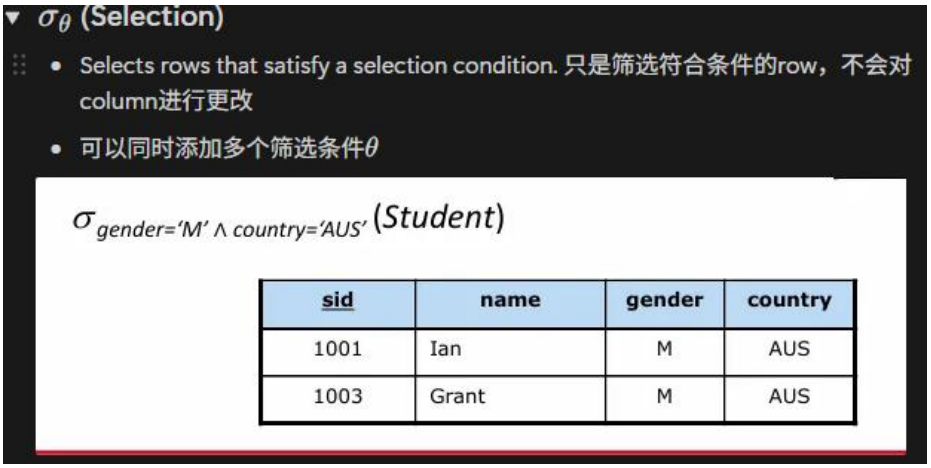
Week4

- 知道如何用 RA 表示一个 query
- 熟悉 plain description 对应的 set operation

RA (Relational Algebra)

- **6 Basic Operator: Projection (不会返回重复行), Selection, Cross, Union, Difference, Rename**





×(Cartesian/Cross product)

R	
A	B
α	1
β	2

X

S		
C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

=

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

- $R \times S = \{ts \mid t \in R \wedge s \in S\}$
- If two table have the same attribute name, then use the `rename` operation
- If R or S is empty, then $R \times S$ is also empty

∪ (Union)

Set Union $R \cup S$

- Definition: $R \cup S = \{t \mid t \in R \vee t \in S\}$

- R and S must have compatible schema, even for value domain
  - 下面只有在筛选出Student里面的sid列后才能进行difference计算，否则不满足上面的描述
  - 对于union和intersection也是同理

Student			
sid	name	gender	country
1001	Ian	M	AUS
1002	Ha Tschi	F	ROK
1003	Grant	M	AUS
1004	Simon	M	GBR
1005	Jesse	F	CHN
1006	Franzisca	F	GER

their schmon is different

Student – Postgraduate ?

Postgraduate
sid
1003
1004
1005

Now they have same attributes

π_{sid}(Student) - Postgraduate

sol difference

sid
1001
1002
1006

— (Difference)

Set Difference $R - S$

- Definition: $R - S = \{ t \mid t \in R \wedge t \notin S \}$

- R and S must have compatible schema, even for value domain
  - 下面只有在筛选出Student里面的sid列后才能进行difference计算，否则不满足上面的描述
  - 对于union和intersection也是同理

Student

sid	name	gender	country
1001	Ian	M	AUS
1002	Ha Tschi	F	ROK
1003	Grant	M	AUS
1004	Simon	M	GBR
1005	Jesse	F	CHN
1006	Franzisca	F	GER

their schmon is different

Student – Postgraduate ?

Postgraduate
sid
1003
1004
1005

Now they have same attributes

π_{sid}(Student) - Postgraduate

set difference

sid
1001
1002
1006

ρ (Rename)

只更改表名

$\rho_x(E)$: change relationship (i.e. 表名) E to x

既更改表名，又更改列名

$\rho_{x(A_1,A_2,\ldots)}(E)$: change relationship (i.e. 表名) E to x, also change the attributes name

比如下面的就是把"UnitOfStudy"换成了"UOS"，并且还重新命名了fields

$\rho_{UOS(ucode,title,credits)}(UnitOfStudy)$

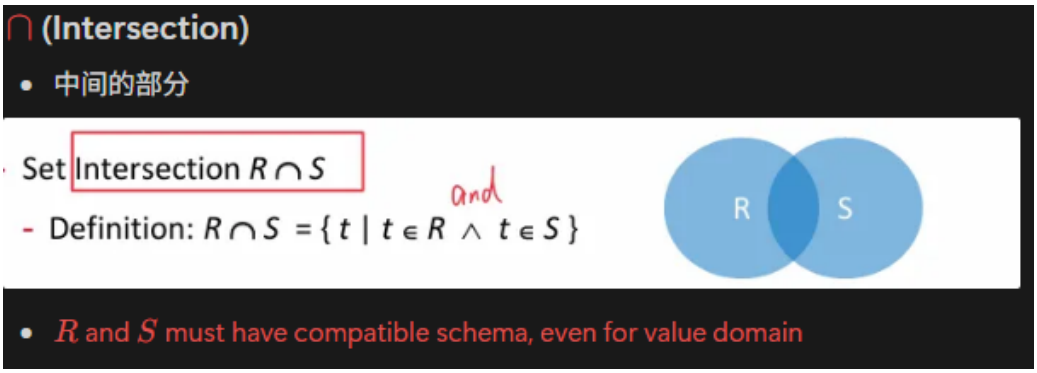
UnitOfStudy

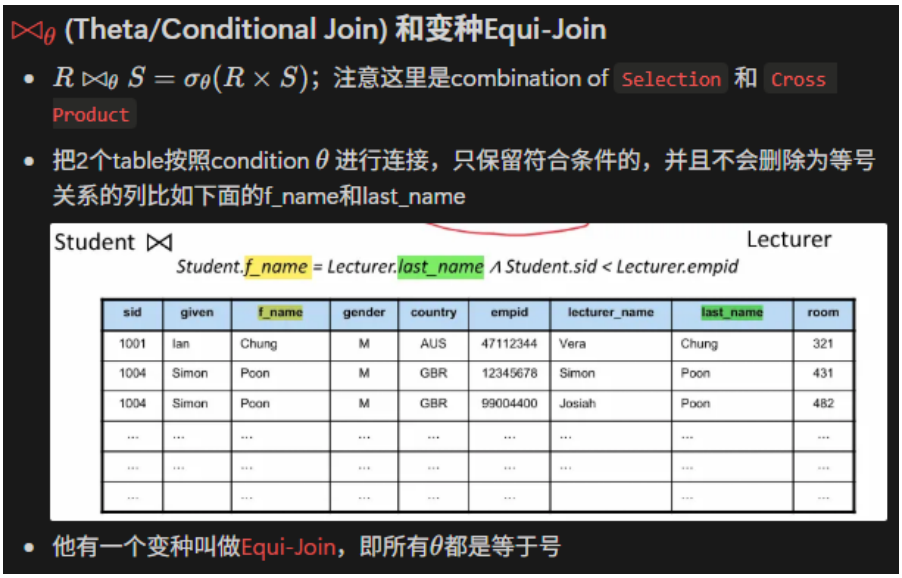
uos_code	title	points
COMPS138	Relational DBMS	6
COMPS318	Data Mining	6
INFO6007	IT Project Management	6
SOFT1002	Algorithms	12
ISYS3207	IS Project	4
COMPS702	Thesis	18

UOS

ucode	title	credits
COMPS138	Relational DBMS	6
COMPS318	Data Mining	6
INFO6007	IT Project Management	6
SOFT1002	Algorithms	12
ISYS3207	IS Project	4
COMPS702	Thesis	18

- 3 Derived: Intersection, Conditional Join, Natural Join





sid	given	f_name	gender	country	empid	lecturer_name	last_name	room
1001	Ian	Chung	M	AUS	47112344	Vera	Chung	321
1004	Simon	Poon	M	GBR	12345678	Simon	Poon	431
1004	Simon	Poon	M	GBR	99004400	Josiah	Poon	482
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
...		...	...	...	...		...	...

Enrolled

sid	uos_code
1001	COMP5138
1002	COMP5702
1003	COMP5138
1006	COMP5318
1001	INFO6007
1003	ISYS3207

UnitOfStudy

uos_code	title	points
COMP5138	Relational DBMS	6
COMP5318	Data Mining	6
INFO6007	IT Project Management	6
SOFT1002	Algorithms	12
ISYS3207	IS Project	4
COMP5702	Thesis	18

=

sid	uos_code	title	points
1001	COMP5138	Relational DBMS	6
1002	COMP5702	Thesis	18
1003	COMP5138	Relational DBMS	6
1006	COMP5318	Data Mining	6
1001	INFO6007	IT Project Management	6
1003	ISYS3207	IS Project	4

JOIN

- Implicit join is Cartesian join/CROSS join
  - 类似于 FROM A, B
  - 如果 A 或 B 为空，则 CROSS 也为空
- Inner join，特殊写法 using 替代 on；省略 Inner
  - Theta Join（θ-Join）

    - **Equi-Join 等值连接**
    - **Natural Join**
        - **Natural Join Caveat**

Natural Join Caveat：如果两个表中没有相同的列名，则 `NATURAL JOIN` 会返回笛卡尔积（不推荐）。

- $R \bowtie S = \pi_{unique\_attributes}(\sigma_{equality\_of\_common\_attributes}(R \times S))$

- **Outer join**
    - **LEFF JOIN**
        - **NATURAL LEFT OUTER JOIN:** 不需要 ON 语句，它会自动匹配两个表中相同名称的列，并以左表（Employee）为主表. 结果可能和 LEFT JOIN 不同，因为 NATURAL JOIN 可能匹配多个同名列，而 LEFT JOIN 只会匹配 ON 指定的列。
    - **RIGHT JOIN**
    - **FULL JOIN:** 不会返回重复的行。在右表没有孤儿行的情况下等同于 LEFT JOIN
- **Natural Join vs. Explicit Equi-Join:** 如果用 explicit equi-join，会导致连接字段出现 2 次,除非用 USING 替代 ON

Set operation UNION, INTERSECT, EXCEPT

- UNION 和 UNION ALL 的区别: UNION 会自动删除 duplicate，而 UNION ALL 会保留 duplicates
- 计算指定元素在两个 table 进行 XXX ALL operation 后的出现次数

Suppose a tuple occurs m times in R and n times in S, then it occurs:

- R UNION ALL S: $m + n$ times
- R INTERSECT ALL S: $min(m, n)$
- R EXCEPT ALL S: $max(0, m - n)$

- **两个 Set 需要有相同的的 Schema**
    - **Same number of columns:** R and S must have the same number of columns.
    - **Same data types:** The corresponding columns in both relations must have compatible (typically the same) data types.
- EXCEPT 表示在 A 表不在 B 表
- 在使用 set operator 后可以使用 ORDER BY

TUT 4

- Additional (derived) operations:
  - E.g.: *intersection*, *join*. [Not essential, but very useful]
  - Equivalence: Intersection: $R \cap S = R - (R-S)$

    Join: $R \bowtie_\Theta S = \sigma_\Theta (R \times S)$

Intersection of
R = {1, 2, 3}
S = {3, 4} = {3}

Difference

R – S = {1, 2 }
R – (R - S) = {3}

$(1,2,3) - (1,2) = \{3\}$

- Example: List the names of all students enrolled in 'Relational DBMS'
  - One way is
    - π_{name} ($\sigma_{title='Relational\ DBMS'}$ ((Student ⋈ Enrolled) ⋈ UnitOfStudy))
  - Another (*more efficient*) way is:
    - π_{name} (Student ⋈ (Enrolled ⋈ ($\sigma_{title='Relational\ DBMS'}$ (UnitOfStudy))))

先 filter 几条

先 filter 一条

Student

sid	name	gender	country
1001	Ian	M	AUS
1002	Ha Tschi	F	ROK
1003	Grant	M	AUS
1004	Simon	M	GBR
1005	Jesse	F	CHN
1006	Franzisca	F	GER

Enrolled

sid	uos_code	semester
1001	COMP5138	2023-S2
1002	COMP5702	2023-S2
1003	COMP5138	2023-S2
1006	COMP5318	2023-S2
1001	INFO6007	2023-S1
1003	ISYS3207	2023-S2

UnitOfStudy

uos_code	title	points
COMP5138	Relational DBMS	6
COMP5318	Data Mining	6
INFO6007	IT Project Management	6
SOFT1002	Algorithms	12
ISYS3207	IS Project	4
COMP5702	Thesis	18

```
SELECT S.Name
FROM Student S, Transcript T
WHERE S.studId = T.studId
AND T.uosCode IN ('INFO2005','INFO2120')
```

可以用 OR 在语义上替换 IN

```
Without using IN operator or a set construct:

SELECT S.name
FROM Student S, Transcript T
WHERE S.studId = T.studId
AND ( T.uosCode = 'INFO2005' OR T.uosCode = 'INFO2120' )
```

$$\pi_{Name}(\sigma_{uosCode='INFO2005' \vee uosCode='INFO2120'}(Student \bowtie Transcript))$$

```
Book (isbn, title, publisher, publicationYear)
Author (aname, birthdate)
Publisher (pname, address)
Wrote (isbn, aname) // which author wrote which book
```

c) $\pi_{aname}(\sigma_{title='A\ First\ Course\ in\ Datbase\ Systems'}(Book \bowtie Wrote))$ Natural join

List the author(s) of the book 'A First Course in Database Systems'.

d) $\pi_{address}(\sigma_{title='Databases' \vee title='Data\ Management'}(Publisher \bowtie_{pname=publisher} Book))$ Theta join

Show the address of the publisher(s) of books titled 'Databases' or 'Data Management'.

c) Find all authors (name) who published at least one book with Acme Publishers

$$\pi_{aname}(\sigma_{publisher='Acme'}(Book \bowtie Wrote))$$

d) Find all authors (name) who never published a book with Acme Publishers.

$$\pi_{aname}(Author) - \pi_{aname}(\sigma_{publisher='Acme'}(Book \bowtie Wrote))$$

Most efficient answer

不会返回重复 row

SET OPERATION

Question 4: (2 marks)

Given the following schema:

Flood(floodID, floodYear, floodDetails)

Cities(cityID, cityName, totalPopulation)

FloodAffectedCities(cityID, floodID, totalRainfall, damageDetails)

(a) Write a Relational Algebra expression (RA) that finds the id of all cities which were affected by floods both before 1995 and after 2020.

(b) Write a Relational Algebra expression (RA) that finds the name of all the cities which were never affected by floods.

Solution: 不是 or

a.

$$\pi_{cityID}(\sigma_{floodYear<\ 1995'}(Flood \bowtie FloodAffectedCities)$$

∩

$$\pi_{cityID}(\sigma_{floodYear>\ 2020'}(Flood \bowtie FloodAffectedCities)$$

b.

$$\pi_{cityName}(Cities \bowtie (\pi_{cityID}(Cities) - \pi_{cityID}(FloodAffectedCities)))$$

Week5

- CHECK 是在插入数据前进行的，不管用不用 INITIALLY IMMEDIATE 等语句；
- CHECK 是针对行的
- CHECK 里面不支持 subquery，不支持聚合函数
- 需要注意检查的顺序
- **Only Domain constraints are modifiable**

Domain Constraints（static）

- NOT NULL / NULL
- DEFAULT
- CHECK -- User defined domain
    - 确保每个属性的值符合预定义的数据类型和取值规则。比如，成绩”只能是 ’A’，’B’，’C’，’D’，’F’ 之一
    - CREATE DOMAIN，用于创造新的数据类型
        - CREATE DOMAIN GradeDomain AS CHAR(1) DEFAULT 'P' CHECK (VALUE IN ('F', 'P', 'C', 'D', 'H'));

Key Constraints & Referential Integrity（static）

- Primary key
- Unique
- FOREIGN key
    - **Referential Integrity**
        - 主要关注的是 REFERENCES，后面这些 action 是额外的，因为默认是不会有任何操作的。
        - CASCADE
        - SET NULL
        - SET DEFAULT，如果对应的 col 没有 default constraint 会报错
        - ON DELETE/UPDATE CASCADE / SET NULL / SET DEFAULT
        - 两个可以同时存在，比如 ON DELETE .... ON UPDATE ....
    - **Delay**
        - NOT DEFERRABLE (默认的)，表明检测方式不能修改
        - DEFERRABLE INITIALLY IMMEDIATE，立刻检测
            - FOREIGN KEY (lecturer_id) REFERENCES Lecturer(lecturer_id) DEFERRABLE INITIALLY IMMEDIATE
        - DEFERRABLE INITIALLY DEFERRED，等 transaction 结束再检查
        - 修改上面 2 种的语法
            - SET CONSTRAINTS name IMMEDIATE
            - SET CONSTRAINTS name DEFERRED

Semantic Integrity Constraints（static）

- 一般用 ASSERTION 来解决跨表检测，因为 CHECK 无法跨表检测
    - CHECK，ASSERTION 是在插入或更新 **之前**触发的
- ASSERTION Cannot modify data
- 2 种语法

- CREATE ASSERTION assertion-name CHECK (condition)
  - CREATE ASSERTION smallclub CHECK ((SELECT COUNT(*) FROM Sailors) + (SELECT COUNT(*) FROM Boats) < 10);
- 通过 CREATE OR REPLACE FUNCTION 的形式来达成；不会 return new

```
--For a sailing club to be categorized as small, we require that the sum of the number of boats and
--number of sailors, be less than 10 at all times.
```

```
create or replace function maxcountsb() returns boolean as $maxc$
begin
if ((SELECT COUNT(sailors.sid) FROM Sailors)
        + (SELECT COUNT(boats.bid) FROM boats) < 9)
    then return true;
    else
    return false;
    end if;
end;
$maxc$ language plpgsql;

CREATE TABLE Boats (
            bid    INTEGER,
            PRIMARY KEY (bid),
            color  CHAR(10)--,
            --CHECK (maxcountsb())
            );
```

Trigger (Dynamic)

- 3 components
  - ON event: what activates the trigger
  - IF condition: test the condition's truth to determine whether to execute an action；不是所有的 trigger 都有 condition
  - THEN action: what happens if the condition is true
- Trigger can modify data
- 总共有 2 种 granularity
  - Row-level：每有一行进行改动，则检查一次
  - Statement-level：每条语句检查一次；不是每个事务，因此事务中如果出现多个语句，则这里会触发多次

语句级触发器

```sql
CREATE FUNCTION function_name()
RETURNS trigger AS $$
BEGIN
  -- 触发时执行的操作
  RETURN NULL; -- 对于 AFTER 触发器可以返回 NULL，表示不进行操作
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_name
AFTER INSERT OR UPDATE OR DELETE ON table_name -- 不能在一个语句中同
FOR EACH STATEMENT
EXECUTE FUNCTION function_name();
```

行级触发器

```sql
CREATE FUNCTION function_name()
RETURNS trigger AS $$
BEGIN
  -- 可以访问 NEW.column_name 或 OLD.column_name
  RETURN NEW;  -- BEFORE 触发器必须返回 NEW
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_name
BEFORE INSERT OR UPDATE OR DELETE ON table_name
FOR EACH ROW
EXECUTE FUNCTION function_name();
```

只能要么是 BEFORE 要么是 AFTER

其中针对 BEFORE, AFTER, INSERT, UPDATE, DELETE 都是可选项

- Use BEFORE triggers: Usually for checking integrity constraints
- Use AFTER triggers: Usually for integrity maintenance and update propagation

```sql
-- 定义一个触发函数：把对视图的 INSERT 转换为对 employee 表的 INSERT
CREATE FUNCTION insert_employee_into_view()
RETURNS trigger AS $$
BEGIN
  -- 把插入视图的行为改为插入真实表
  INSERT INTO employee(name, dept_id)
  VALUES (NEW.name, (SELECT id FROM department WHERE dept_name = NE
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

-- 定义 INSTEAD OF 触发器
CREATE TRIGGER employee_view_insert
INSTEAD OF INSERT ON employee_view
FOR EACH ROW
EXECUTE FUNCTION insert_employee_into_view();
```

这里是针对 view 的

TUT5

```
ALTER TABLE Requires ADD CONSTRAINT Requires_uoSCode_fk_UOS FOREIGN KEY
(uoSCode) REFERENCES UnitOfStudy(uoSCode) ON DELETE CASCADE;
```

ASSERTION

1. The number of students enrolled in a unit-of-study must equal the current enrolment;

```
CREATE ASSERTION EnrollmentAssert CHECK (
    NOT EXISTS (
        SELECT 1
        FROM UoSOffering o
        WHERE enrollment != (SELECT COUNT(*)
            FROM Transcript t
            WHERE t.uosCode=o.uosCode AND
            t.semester=o.semester AND
            t.year=o.year)
    )
);
```

2. The room assigned to a unit-of-study must have at least as many seats as the maximum allowed enrolment for the unit;

```
CREATE ASSERTION RoomCapacityAssert CHECK (
    NOT EXISTS (
        SELECT 1
        FROM (UoSOffering NATURAL JOIN Lecture)
            NATURAL JOIN ClassRoom
        WHERE seats < maxEnrollment
    )
);
```

TRIGGER

什么时候用 `RETURN NEW` 或 `RETURN NULL`？

触发器类型	推荐返回值	含义/说明
`BEFORE INSERT/UPDATE`	`RETURN NEW` 或 `RETURN NULL`	可以修改或取消即将执行的插入/更新操作。`NULL` 表示取消操作。
`AFTER INSERT/UPDATE/DELETE`	`RETURN NEW`（或 `RETURN OLD`）	必须返回一个记录，但返回值不会被使用，只是语法要求。

1. We can write a trigger to update the enrolment number for a unit of study offering *when a student is added* (in Transcript), as shown below:

```
CREATE OR REPLACE FUNCTION updateEnrol() RETURNS trigger AS $$
BEGIN
    UPDATE UoSOffering U
    SET enrollment = enrollment+1
    WHERE U.uosCOde = NEW.uosCode AND semester = NEW.semester
        AND year = NEW.year;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER UpdateEnrol AFTER INSERT ON Transcript
FOR EACH ROW EXECUTE PROCEDURE updateEnrol();
```

ROW VS STATEMENT
AFTER VS BEFORE

Question 6: (2 marks)

Consider two relations Projects(ProjID, Location, Budget) and Employees(EmpID, Name, ProjID), where ProjID in *Employees* is a foreign key to *Projects*. Additionally, an employee can be assigned to *at most one* single project.

1. If we delete a project tuple in the table Projects:
   (i) What type of integrity constraint needs to be checked?
   (ii) What are the possible ways to enforce this type of constraint?

2. If we use a trigger to implement the integrity constraint ON DELETE CASCADE in the Employees table, state the type of TRIGGER clauses you will use, namely choose (i) BEFORE vs. AFTER, and (ii) ROW vs. STATEMENT. State the reasons for your choice. 0 marks for no explanation.

1. (i) Yes, there is a referential integrity constraint from Employee that needs to be checked. (ii) to enforce this integrity, we have the following choices: it can be *rejected*, the *delete is cascaded*, or a *default value is set*.
2. We will use a STATEMENT as there may be more than one tuple in Employees that have a match to the deleted row in Projects. We will use the AFTER clause because the deletion of rows in Employee happens after the matching project row is deleted: it is about integrity maintenance.

Week6

Set Comparison Operators/Nested Query

- [NOT] IN
- [NOT] EXISTS

```sql
-- 查询所有选修任何一门课程的学生
SELECT name
FROM Student s
WHERE EXISTS (
    SELECT 1
    FROM Enrolled e
    WHERE e.student_id = s.id
);
```

- ALL

```sql
-- 查询比所有 "COMP5138" 选课学生的 GPA 都高的学生
SELECT name
FROM Student
WHERE gpa > ALL (
    SELECT s2.gpa
    FROM Student s2
    JOIN Enrolled e2 ON s2.id = e2.student_id
    WHERE e2.uos_code = 'COMP5138'
);
```

和‘找出 GPA 是所有 COMP5138 的学生中最高的（可能并列）‘不是一个意思，所以不用>=

- SOME

```sql
-- 查询 GPA 高于部分（至少一个） "COMP5138" 学生的学生
SELECT name
FROM Student
WHERE gpa > SOME (
    SELECT s2.gpa
    FROM Student s2
    JOIN Enrolled e2 ON s2.id = e2.student_id
    WHERE e2.uos_code = 'COMP5138'
);
```

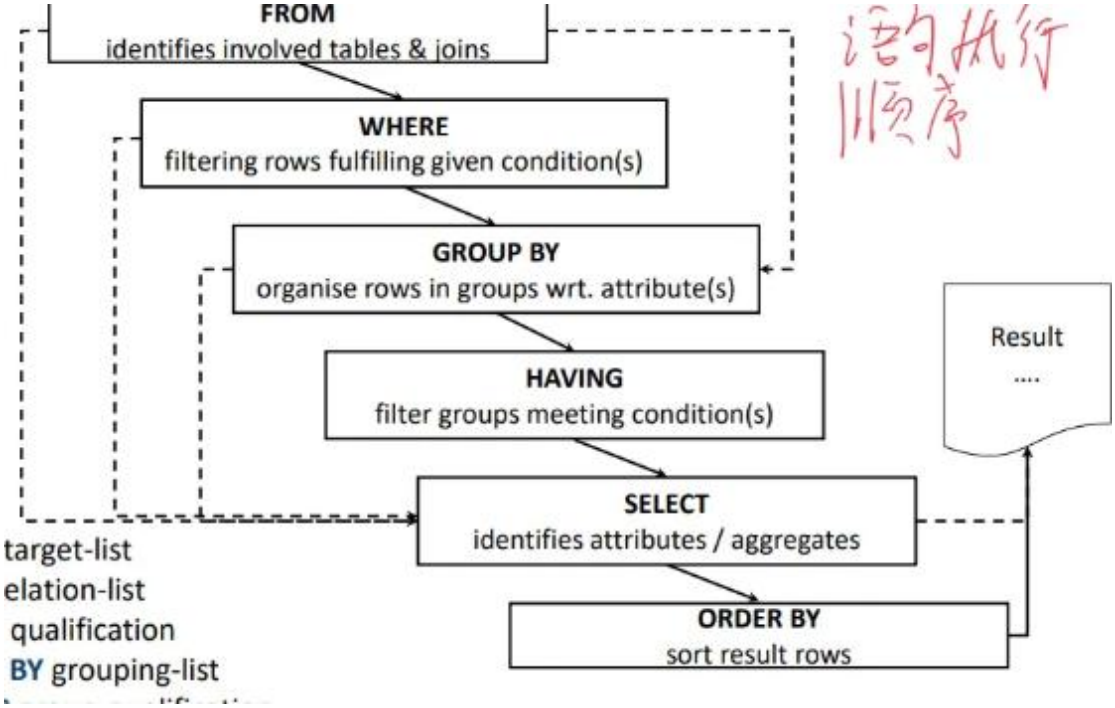
View

- 不是真实存在的
- CREATE VIEW view_name AS ⋯ ;
- CREATE VIEW student_enrollment AS SELECT sid, name, title, semester FROM student NATURAL JOIN Enrolled NATURAL JOIN unitofstudy;

Aggregate function

- COUNT(*)计算 NULL，别的都不算，包括 COUNT(x)
- 不能 MIN(AVG())
- 都对 DUPLICATE 起效，除非 COUNT(DISTINCT x)
- If you use GROUP BY function, then in SELECT or HAVING line, you can only include aggregate function or the attribute you use for GROUP BY
- Predicates in the HAVING clause are applied after the formation of groups, whereas predicates in the WHERE clause are applied before forming groups

SQL 语句执行顺序



NULL and Three-valued Logic

- 任何和 NULL 进行的运算都是 NULL，比如 5+NULL=NULL
- Any comparison with NULL returns unknown，比如 5<NULL 是 unkown
    - Result of WHERE clause predicate is treated as false if it evaluates to unknown
    - SELECT sid FROM enrolled WHERE marks < 50;
        - ignores all students without a mark, that is if a student with null mark, then we will not out put it in above query
- Three-Valued Logic
    - UNKOWN OR TRUE = True
    - UNKOWN AND FALSE = False
    - 别的都是 UNKOWN

TUT6

a) a = 10

Answer: All tuples with a being 10 combined with any value for b, including NULL. Examples: (10, 0), (10, 1), ..., (10,-1), ..., (10, NULL)

d) a < 10 AND NOT b = 20

Answer: Similar to the previous answer: All tuples where a < 10 and not NULL, and b ≠ 20 and also not NULL.

可以看到 where 会把带有 NULL 的判断归类为 False

a) Which lecturers (by id and name) have taught both 'INFO2120' and 'INFO3404'? Write a SQL query to answer this question using a SET operator.

Answer :

```
SELECT id, name
FROM AcademicStaff JOIN UoSOffering ON id=instructorId
WHERE uosCode = 'INFO2120'
INTERSECT
SELECT id, name
FROM AcademicStaff JOIN UoSOffering ON id=instructorId
WHERE uosCode = 'INFO3404';
```

b) Which lecturers (by id and name) have taught both 'INFO2120' and 'INFO3404'? Answer this using a sub-query without SET operators. Make sure your result doesn't include duplicates.

Answer :

```
SELECT DISTINCT id, name
FROM AcademicStaff JOIN UoSOffering ON id=instructorId
WHERE uosCode = 'INFO2120'
AND id IN ( SELECT instructorId
 FROM UoSOffering
 WHERE uosCode = 'INFO3404' );
```

c) Write a SQL query to give the **student IDs** of all students who have enrolled in only one lecture using GROUP BY, and order the result by student ID. A lecture is a unit_of_study in a semester of a year.

Answer:

```
SELECT studId
FROM Transcript
GROUP BY studId
HAVING count(*) = 1
ORDER BY studId;
```

需要注意的是 HAVING 是对 GROUP 进一步进行 filter

g) [Advanced, Optional] Write a SQL query to give the **student IDs** of all students who have enrolled in only one **unit_of_study**, and order the result by student ID. Note that, a student can enrol in the same unit_of_study multiple times, which is still counted as one unit_of_study.

Answer :

```
SELECT studId
FROM Transcript
GROUP BY studId
HAVING count(DISTINCT uoSCode) = 1
ORDER BY studId;
```

h) [Advanced, Optional] Write a SQL query to give the **student IDs** and **names** of all students who have enrolled in only one unit_of_study, and order the result by student ID. Note that, a student can enrol in the same unit_of_study multiple times, which is still counted as one unit_of_study.

Answer :

```
SELECT studId, name
```

Select
name
studId
From Student
where
studId in (g)


```
FROM Student NATURAL JOIN (
 SELECT DISTINCT studId, uoSCode
 FROM Transcript ) AS T
GROUP BY studId, name
HAVING count(*) = 1
ORDER BY studId;
```

Question 5: (2 marks)

Consider the database consisting of the following tables:

Employee(employee_id, employee_name, employee_city, manager_id)

Project(project_id, project_title, budget)

AssignedTo(project_id, employee_id)

Note that a manager must be an employee. Therefore, in employee table, manager_id *references* employee_id.

(a) Find the name of the employees who live in the same city as their manager.

```
a. SELECT m.employee_name
FROM employee e, employee m
WHERE e.employee_id = m.manager_id and
      e.employee_city = m.employee_city;
```

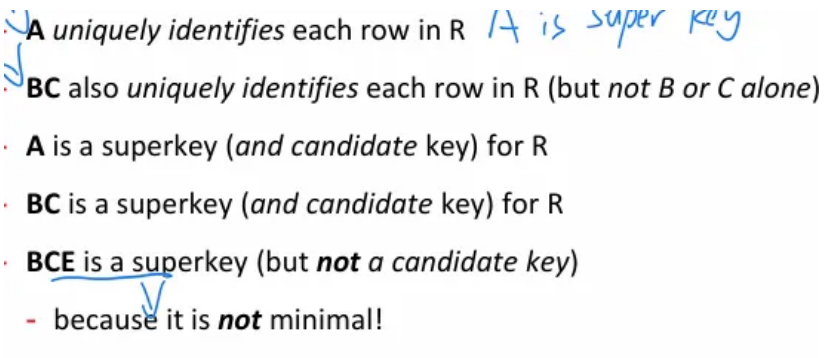
(b) Find the id of those projects that have the highest number of employees assigned to them.

```
b. SELECT project_id
      FROM AssignedTo
      GROUP BY project_id
      HAVING COUNT(employee_id) >=ALL (SELECT count(employee_id)
                                       FROM AssignedTo
                                       GROUP BY project_id);
```

Week8

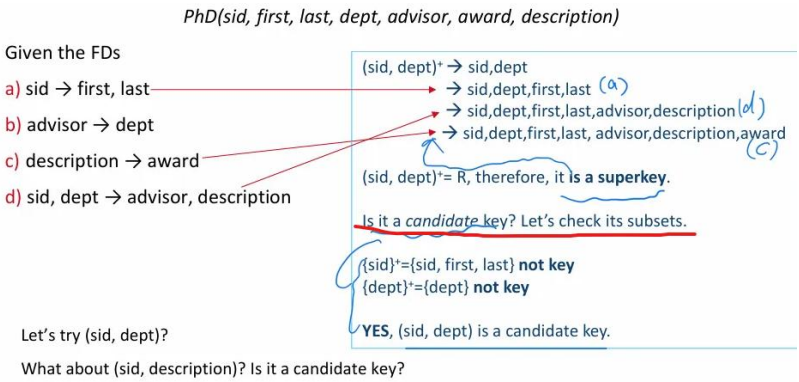
Functional Dependency X->Y

- It's an n-1 relationship, hence 1-1 relationship. Namely, **X 的值唯一确定 Y 的值**。即只要你知道 X，就能**唯一**确定 Y (同一 X 不能有不同 Y)
- 我们可以通过查看一个实例来判断某个函数依赖是否不成立，即通过一个反例可以推翻 FD
- 无法通过查看任何数量的实例来确定 FD 是否成立，因为函数依赖描述的是关系中所有可能合法实例的规则。必须查看所有实例才行
- Inference Rules
  - **Reflexivity (Trivial FDs): A 属于 B 则，A->B**
  - **Augmentation: A->B 则 AC->BC**
  - **Transitivity: A->B, B->C, A->C**
  - **Decomposition: A->BC 则 A->B and A->C**
  - **Union: A->B, A->C 则 A->BC**



Closure of Attribute

- 如果一个 set of attributes 通过已知的 FD 可以得到所有的 relation 里面的 element，它就是 Super key
- 通过甄别只在 FD 左边出现的，但是没在 FD 右边出现的 attributes 可以得到 part of all keys



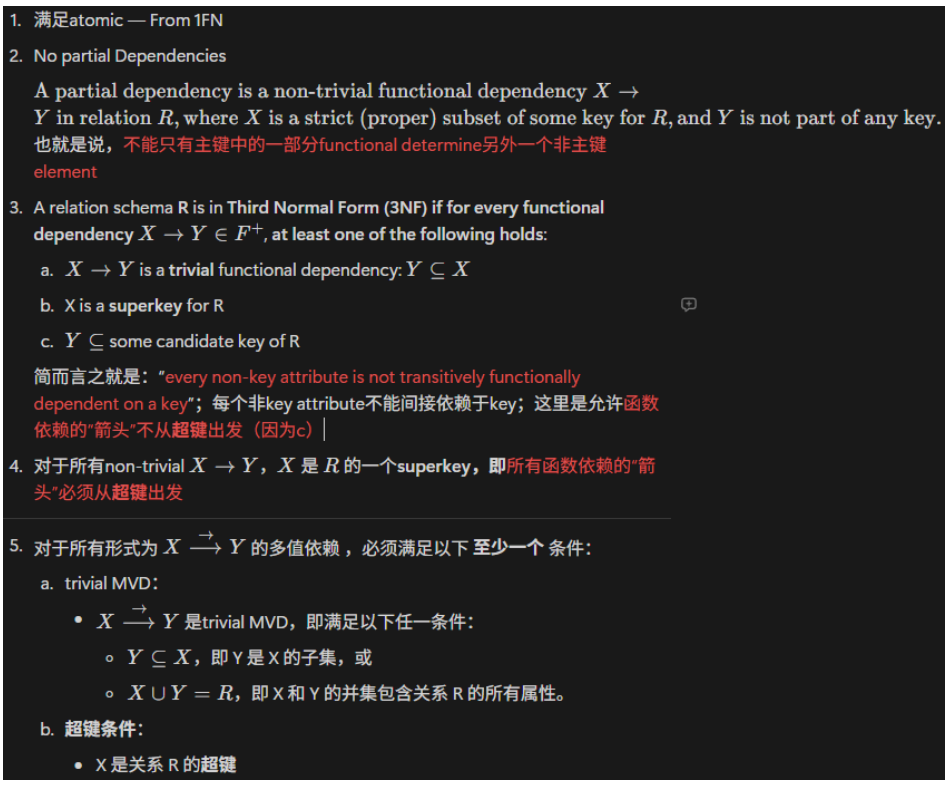
Schema Decomposition (2 methods)

- **Dependency preservation**: No FDs are lost in the decomposition

- Lossless-join (also called non-additive) decomposition: Re-joining (natural join) a decomposition of R should give us back R!

Normal Forms

- Higher restriction **Not necessarily good for mostly retrieval operations**";
- Higher NF include Lower NF
- 1NF: 确保 atomic
  - no multivalued attributes (atomic)
- 2NF: 消除 partial dependency
  - no part of key to non-key (partial dependency: LHS – key's proper subset, RHS – not part of key)
- 3NF: 消除 Transitive dependency
  - no non-key to non-key (not transitive dependency if: LHS – super key, RHS – key's proper subset)
- BCNF: 确保 X 必须是超键
  - no non-key to part of key (LHS must be super key)
- 4NF: solve MVD (Multivalued Dependency)



employee_name	project_id	personal_phone_number
Bob	P1	047012345
Bob	P3	046098765
Bob	P1	046098765
Bob	P3	047012345
Lily	P1	045067543
Fiona	P7	043085432

Is this relation in 4NF?

No: There is at least one non-trivial multivalued dependency

存在多值依赖：$employee\_name \twoheadrightarrow Project\_id$,

而Y不是X的子集，并且X UNION Y不是R

同时$employee_name$ is not a superkey.

TUT8

FD

Consider the following table called *PhD*:

SID	first	last	dept	advisor	award	description
1234	Brian	Cox	IT	Codd	Rejected	Work not deemed sufficient
3456	Albert	Einstein	IT	Boyce	Conditional	Accepted with minor corrections
3456	Albert	Einstein	Physics	Newton	Accepted	Accepted with no corrections
7546	Alan	Turing	IT	Codd	Accepted	Accepted with no corrections
4879	Brian	Cox	Physics	Newton	Conditional	Accepted with minor corrections
4879	Brian	Cox	Media	Attenborough	Accepted	Accepted with no corrections

What *functional dependencies* might exist in the **PhD relation** above?

Sid → name

Br.an Cox → sid is not

dept → advisor

award → description ✓

description → award ✓

branch_name	assests	city	loan_no	customer_name	amount
Mall St	9000000	Sydney	17	Jones	1000
Logan	5000000	Melbourne	23	Smith	2000
Queen	500000	Perth	15	Hayes	1500
Mall St	9000000	Sydney	14	Jackson	1500
King George	10000000	Brisbane	93	Carry	500
Queen	500000	Perth	25	Glenn	2500
Bondi	15000000	Adelaide	10	Brooks	2500
Logan	5000000	Melbourne	30	Johnson	750

› Do you see any other functional dependency?

- For instance, are the following FDs correct?

loan_no → customer_name, amount? **YES**

loan_no → branch_name? **YES**

city → customer_name? **NO**

city → assets? **YES**

sydney — Jackson, Jones

Value are unique

I. destination, departs, airline → gate

II. gate → airline

III. contact → name

IV. name, departs → gate, pickup

V. gate, departs → destination

b) Consider the following collection of tuples. Why is this instance not a legal state for the database?

Destination	Departs	Airline	Gate	Name	Contact	Pickup
Berlin	1/06/2012 11:25	Lufthansa	3	Justin Thyme	0416594563	1
Madrid	1/07/2012 14:30	Iberian	4	Willy Makit	0497699256	2
London	3/05/2012 6:10	British Airways	7	Hugo First	0433574387	5
Moscow	1/07/2012 17:50	Aeroflot	6	Rick OhChet	0416594563	7
Berlin	1/06/2012 11:25	Qantas	1	Dick Taite	0469254233	4
Kuala Lumpur	1/08/2012 14:30	Cathay	7	Hugo First	0433574387	2
Singapore	1/08/2012 14:30	Qantas	2	Hugo First	0433574387	2
London	1/07/2012 17:50	Lufthansa	3	Justin Thyme	0413456789	4

需要根据 FD 来判断，即一个 X 对应一个 Y，但是 Y 不需要不同

这里说的是 Contact -> Name

British Airways and Cathay can't share the same gate 7.

Rick can't use Justin's phone number (but it's fine that Justin has 2 numbers).

Hugo can't make two flights at the same time on 1/8/12. Maybe he changed flights and the old flight didn't get removed.

Attribute Closure

a) Is (contact, departs, airline) a candidate key from the above functional dependencies. Can you find an alternative?

The only attributes of the original relation that never appear on the right of an FD are *departs* and *contact*, so these must be part of all keys.

So start with (*departs*, *contact*). What is its closure (*departs*, *contact*)$^+$? The following steps grow the set of implied attributes by incorporating the FDs listed at the start.

Implied attributes	Updated closure	Comment
departs, contact	*{departs, contact}*	Trivial attributes
Name	*{departs, name, contact}*	From *contact* via FD III
gate, pickup	*{departs, gate, name, contact, pickup}*	From *name*, *departs* via FD IV
Destination	*{destination, departs, gate, name, contact, pickup}*	From *gate*, *departs* via FD V
Airline	*{destination, departs, airline, gate, name, contact, pickup}*	From *gate* via FD II

这里是先判断除所有的 key 都要包含 departs, contace。因此先求他们的 closure。再就(contact, departs, airline)最多只能是 Super key，因为不满足 minimal。

b) Is the relation in 3NF?

gate->airline does NOT meet the 3NF restriction that either the LHS is a superkey or at least for the RHS to be part of a key. (hence cannot be in 3NF). In addition，还要检查 2NF 和 1NF, contact->name 不满足 2NF(Partial dependenct)，因为 contact 是 candidate key 中的一个；下面是老师给的例子

Teacher_name	UnitOfStudy	Teacher_position
Mary	COMP9120	Lecturer
Mary	COMP5313	Lecturer

In this, we assume that Teacher_name -> Teacher_position, and {Teacher_name, UnitOfStudy} -> Teacher_position. As such, although the relation doesn't have any transitive dependencies, it is not in 3NF because it doesn't satisfy 2NF.

Lossless-Join

c) Explain whether it is a lossless-join decomposition to decompose the relation into the following:

R1(destination, departs, gate)

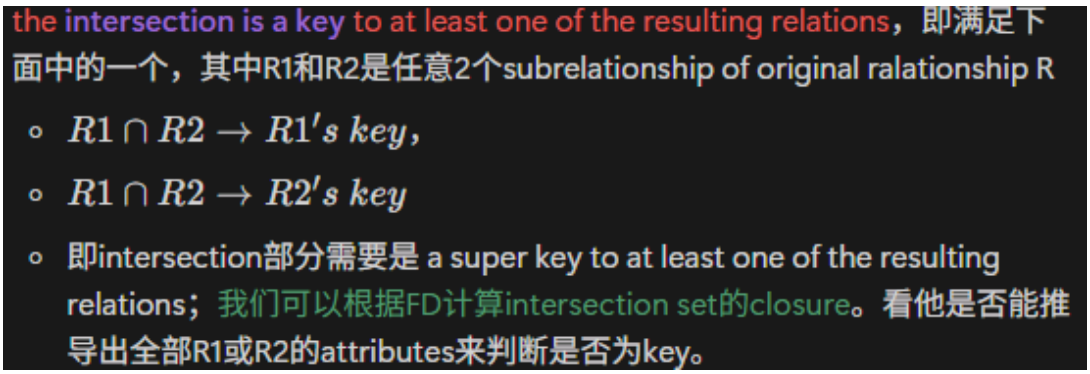
R2(contact, departs, pickup)

R3(gate, airline)

R4(contact, name)

这里因为 R3 和 R4 是直接和本来的 FD 相关，因此可以从 R 中删除 airline, name 这 2 个 attributes，变成 R7(dest, dep, gate, contact, pickup)；这里之所以不删除 gate 和 contact 是因为 R3 和 R7 的 intersection 为 gate 才能保证 gate 是 R3 的 superkey，对于 R4 同理。

现在来看 R7 (dest, dep, gate, contact, pickup)，他如果分解成 R1 和 R2，则它们的 intersection 是(departs)，根据现有的 FDs，我们足以推断出 departs 的 closure 不能包含所有在 R1 或 R2 里面的 attributes，因此不是 lossless 的。



正确的步骤是一个 Ri 一个 Ri 的检查，比如先从 R 里面分出 R3，再分出 R4，然后每一步检查分的是不是满足 lossless。

d) Give a lossless-join decomposition of the original relation into BCNF relations.

很重要!!!

首先不满足 BCNF 的有

所有 FD

那么，我们先拿最小的 FD 开刀，因此分出了（注意这里是一步步来的，这里为了方便所以一起分了）

R3(gate, airline),

R4(contact, name)

这里在 R3 和 R4 中，都满足 BCNF 因此不用再分；此外因为

gate → airline

contact → name

所以我们分别用 gate 和 contact 作为 R3 和 R4 的 key,

现在还剩下 R7(destination, departs, gate, contact, pickup)，还是不满足 BCNF，因为

contact, departs → gate, pickup；从(a)中知道(contact,depart)是 super key，所以没问题

gate, departs → destination；但是 gate 和 departs 不是 super key，所以有问题

因此需要把(gate, departs, destination)分成 R5，在这里面，我们需要把 gate 和 departs 作为 super key。

最后剩下 R6(departs, gate, contact, pickup)，而我们发现 R6 也满足 BCNF 因为

contact, departs → gate, pickup

最后得到作为 decoposed set

R3(gate, airline),

R4(contact, name)

R5(destination, departs, gate) R6(departs, gate, contact, pickup)

Loss-Less Join and dependency preserving

Consider a relation **R(EmployeeID, Name, Department, Manager, Salary, Position)** with the following functional dependencies:

EmployeeID → Name, Department, Position
Department → Manager
Position → Salary

1. Compute the closure of **{EmployeeID, Department}**, i.e., **{EmployeeID, Department}$^+$**.

We start with {EmployeeID, Department} and apply the functional dependencies to determine all attributes we can derive.

- EmployeeID → Name, Department, Position- Since we have EmployeeID, we can derive Name, Department, and Position. (We already have Department.)

- Department → Manager- Since we have Department, we can derive Manager.

- Position → Salary- Since we have Position, we can derive Salary.

$\{EmployeeID, Department\}^+ = \{EmployeeID, Name, Department, Position, Manager, Salary\}$

2. Is {EmployeeID, Department} a candidate key?

A candidate key is a minimal set of attributes that uniquely determines all attributes in the relation.

From 1. It is a **superkey** because it determines the relation R.

Now let us check whether EmployeeID or Department are keys themselves. Compute $\{EmployeeID\}^+$ and $\{Department\}^+$.

$\{EmployeeID\}^+ = \{$ EmployeeID, Name, Department, Position, Manager, Salary$\}$ which is a key and it is minimal.

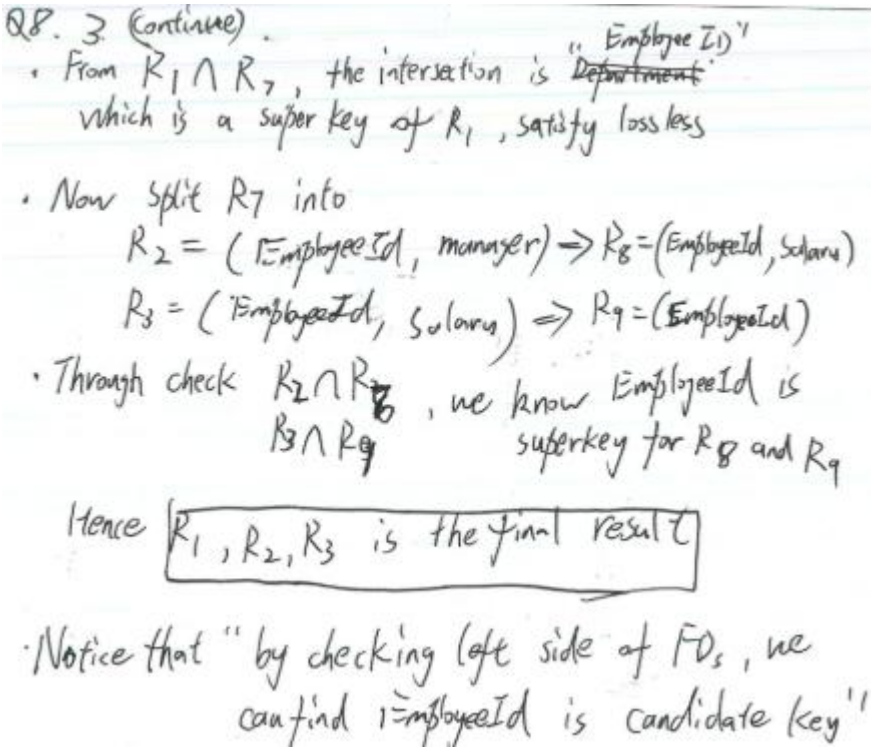
$\{Department\}^+ = \{Department, Manager\}$

Therefore, {EmployeeID, Department} is **not** a candidate key because it is *not* minimal.

3. The candidate Key is Employee Id from 2. Above. The relation R is not in BCNF because of the violation caused by the FD Department → Manager

Decompose the relation R into relations : R1 = (Department, Manager) and R2 = (EmployeeID, Name, Department, Salary, Position), R1 is BCNF compliant. However, R2 is not because of the violation of FD Position → Salary**.** Decompose R2 into relations: R3 = (Position, Salary) and R4 = (EmployeeID, Name, Department, Position). Now R3 and R4 are BCNF compliant.

3. No, since Department & Position are not Super key (very obvious)
split in belowing way:
R_1 = (EmployeeID, Name, Department, Position)
now the remaining is (EmployeeID, Manager, Salary) = R_2



Suppose you are given a relation R with four attributes ABCD. For the given sets of FDs,

A → B,

BC → D,

A → C

do the following:

(a) Identify the candidate key(s) for R.

(b) Identify the best normal form that R satisfies (1NF, 2NF, 3NF, or BCNF).

(c) If R is not in BCNF, decompose it into a set of BCNF relations that preserve the dependencies.

Solution:

a) Candidate key: A

b) R is in 2NF but not 3NF (because of the FD: BC → D).

c) BC → D violates BCNF since BC does not contain a key. So, we split up R as in: BCD, ABC.

Week9

Transaction Syntax

BEGIN;
SQL;
COMMIT;
ROLLBACK/ABORT;

这里会在 SQL 失败时回滚，如果没失败就不回滚；ABORT 和 ROLLBACK 等价
在 postgreSQL 里面，会自动确保 Atomic，因此即使没有 ROLLBACK，在 SQL 失败的时候也不会修改数据；但是仍然需要 ROLLBACK 来结束事务

- **COMMIT 成功**:
  - 事务中的所有更改都会永久保存到数据库。
- **COMMIT 失败**（可能的原因：磁盘空间不足、网络断开等）:
  - **事务仍然处于活动状态**，不会自动回滚。
  - 你需要手动执行 ROLLBACK 进行回滚，或者尝试再次 COMMIT。

ACID

Atomicity — log

- A real-world transaction is expected to happen or not happen at all
- 通过 ROLLBACK 达成

Consistency

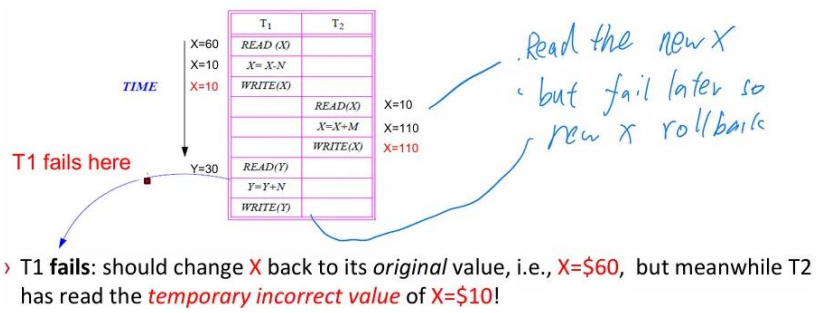
- 数据库**无法修正你写错的业务逻辑（Business logic）**。如果你的事务代码错了，数据库再强也没法保你一致。
- can only modify data in allowed ways. 即需要满足数据库中的 Constraints

Durability — log, Recover manager, exclusive lock

- Once a transaction is committed, its effects should persist in a database, and these effects should be permanent even if the system crashes.
- Recovery Manager of DBMS responsible for this part
- If a transaction aborts, depending on the recovery protocol, use the log to undo/redo the transaction.
  - Undo: 将数据项恢复到事务开始之前的值。
  - Redo: 将已经提交的事务操作再“做一遍”，确保它们的更改是永久性的。

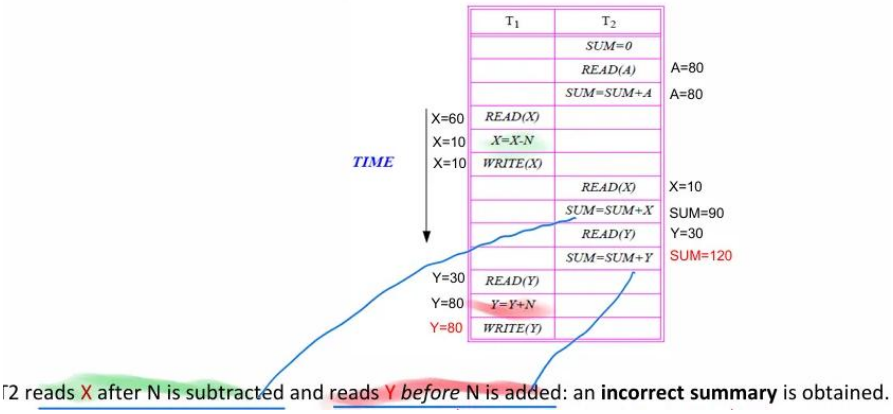
Isolation — Concurrently, interleaved

- Transactions can run concurrently; meaning their operations can be interleaved.
- **3 Concurrent Access Issues**
  - **Temporary update**: 一个事务对某项数据做了修改，**另一个事务读取了该临时值**，但前一个事务最后却失败或回滚，导致另一个事务**基于错误值作出错误决策**。



› T1 **fails**: should change X back to its *original* value, i.e., X=$60, but meanwhile T2 has read the *temporary incorrect value* of X=$10!

- **Incorrect summary**：一个事务对某些数据做聚合操作（如 SUM），但这些数据**在聚合过程中被其他事务修改了部分值**，导致聚合结果**混合了旧值和新值**



T2 reads X after N is subtracted and reads Y *before* N is added: an **incorrect summary** is obtained.

- **Lost update**：两个事务几乎同时对同一数据项进行读取并修改，但**后一个事务的写操作覆盖了前一个事务的写操作**，导致前一个事务的修改被“**丢失**”。

1. **事务 T1 读取某个数据项（如 `balance = 100`）。**
2. **事务 T2 也读取相同的数据项（`balance = 100`）。**
3. **T1 修改数据（如 `balance = balance + 50` → `150`），但尚未提交。**
4. **T2 也修改数据（如 `balance = balance - 20` → `80`），并先提交。**
5. **T1 提交后，`balance` 被覆盖为 `150`，而 T2 的修改（`-20`）丢失了。**

最终结果应该是 `130`（`100 + 50 - 20`），但由于并发问题，实际结果是 `150`（T2 的更新被 T1 覆盖）。

- 4 Isolation Level

```
SET TRANSACTION ISOLATION LEVEL
{ SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }


-- Example
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- 这里开始写查询或更新语句
SELECT * FROM my_table;

COMMIT;
```

- **Read Uncommitted** — Dirty Read
  - 允许事务读取其他尚未提交事务的数据，也叫做 Dirty Read。
  - PostgreSQL 不支持此隔离级别，直接转换为 Read Committed。
- **Read Committed** — Solve **Temporary update**
  - 每次查询只能读取已经提交的数据。no dirty reads.
  - 防止脏读，但在同一个事务中两次读取同一行数据，可能得到不同的结果。
  - **是 PostgreSQL 的默认隔离级别**。
- **Repeatable Read** — Solve **incorrect summary problems**
  - 只能“看到”在它开始之前已经提交的事务所做的更改。
  - transaction 看不到：
    - 其他事务还没有提交的数据 — 避免 Read Uncommitted
    - 以及它执行期间，其他事务新提交的更改 — 避免 Read Committed
- **Serializable** — Solve **lost update problem**
  - 整个执行结果就像是所有事务串行执行一样。

Serializability (Optimistic)

- 你可以通过分析两个事务之间的调度是否满足 **serializability**（特别是 **conflict serializability**）来判断它们是否满足最高的隔离级别 SERIALIZABLE
- **Conflict Serializability -> Serializability**
- **Serializability**
  - A schedule is serializable if and only if it is equivalent to **some** serial schedule；some 表示有一个就行
  - Serializability is expensive to check
- **Conflict Serializability**
  - 如果你能**通过交换不冲突的操作**，把这个调度转换成一个 **串行调度（所有事务一个接一个执行）**，那么它是 conflict serializable; A schedule is conflict serializable if it is conflict equivalent to a serial schedule.
  - **Conflict Pair**
    - **Not a conflict pair (A1, A2)：**
      - A1 读 A，A2 读 A（Read-Read）→ 没有冲突（读操作不会影响另一个读）
      - A1 读 A，A2 读 B（不同数据项）→ 没有冲突
      - A1 写 A，A2 读 B（不同数据项）→ 没有冲突
    - **conflict pair (A1, A2)：**
      - A1 读 A，A2 写 A（Read-Write）→ 有冲突：T1 的读取可能受 T2 的写入影响，或反过来

        - A1 写 A，A2 读 A（Write-Read）→ 有冲突：顺序不同，读到的值不同
        - A1 写 A，A2 写 A（Write-Write）→ 有冲突：最终 A 的值取决于谁最后写
    - 通过 Non-conflicting swappings 或 Precedence Graph 来 check

Lock (Optimistic)

- **粒度大（粗粒度）**：并发性差，多个事务难以同时操作不同的数据项。 (too coarse) - no effective concurrency
- **粒度小（细粒度）**：虽然并发性高，但加锁和管理的**开销大**。(too fine) - lock overhead high
- Shared Lock (S) & Exclusive Lock(X); T1 持有，T2 申请

Read locks: "**Shared**" lock (S)
Write locks: "**Exclusive**" lock (X)

T2 Requests \ Held by T1	**Shared** Read	**Exclusive** Write
Shared Read	OK	T2 wait on T1
Exclusive Write	T2 wait on T1	T2 wait on T1

- **在使用完某个数据项后立即释放锁可能会导致起不到锁的作用!**
- **2PL**
    - 只能先加锁(growing)，再释放锁(Shrinking)
    - 在**没有死锁**和**没有事务失败**的前提下，2PL 能够确保调度的执行是**等价于某种串行执行顺序**的，即满足可串行性要求
    - Basic
        - 可以在 commit 前释放锁
    - Strict
        - 只能在 commit 后释放锁
- **Dead Lock**
    - Example
        - 事务 T1 已经**持有了数据 A 的锁**，并且**正在请求 B 的锁**；
        - 同时，事务 T2 **持有了 B 的锁**，并且**正在请求 A 的锁**；
        - 两个事务相互等待，谁都无法继续执行，这就形成了**死锁**。
    - Solution
        - Deadlock Prevention — Static 2PL
            - **每个事务**在开始时**预声明其读取集（共享锁）和写入集（排他锁）**。

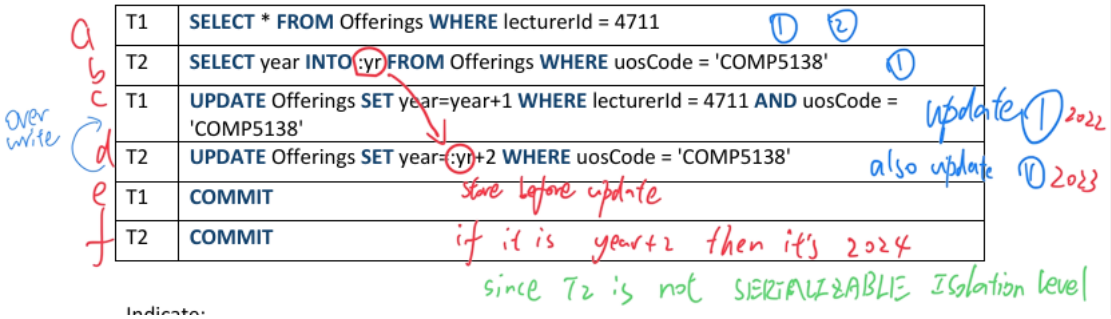
- Deadlock Detection — TIMEOUT

TUT9

uosCode	year	semester	lecturerId
COMP5138	2021	S1	4711
INFO2120	2021	S2	4711

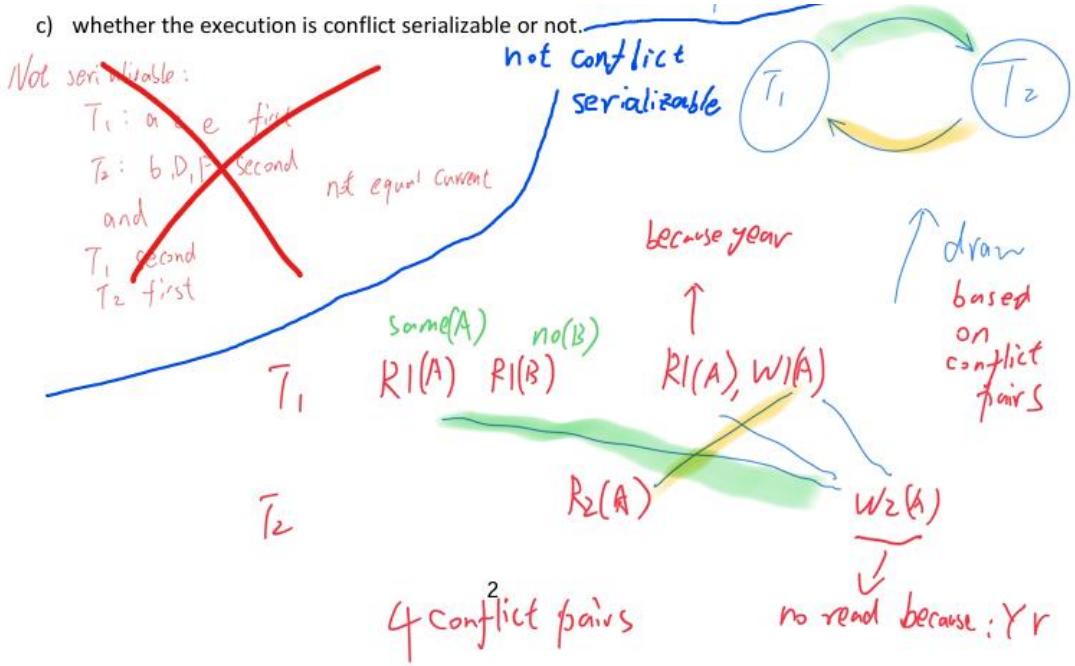
Consider the following hypothetical interleaved execution of two transactions T1 and T2 in a DBMS where concurrency control (such as locking) is not done; that is, each statement is executed as it is submitted, using the most up-to-date values of the database contents.

T1	**SELECT** * **FROM** Offerings **WHERE** lecturerId = 4711
T2	**SELECT** year **INTO** :yr **FROM** Offerings **WHERE** uosCode = 'COMP5138'
T1	**UPDATE** Offerings **SET** year=year+1 **WHERE** lecturerId = 4711 **AND** uosCode = 'COMP5138'
T2	**UPDATE** Offerings **SET** year=:yr+2 **WHERE** uosCode = 'COMP5138'
T1	**COMMIT**
T2	**COMMIT**



b) whether the execution produces any update anomalies Lost update

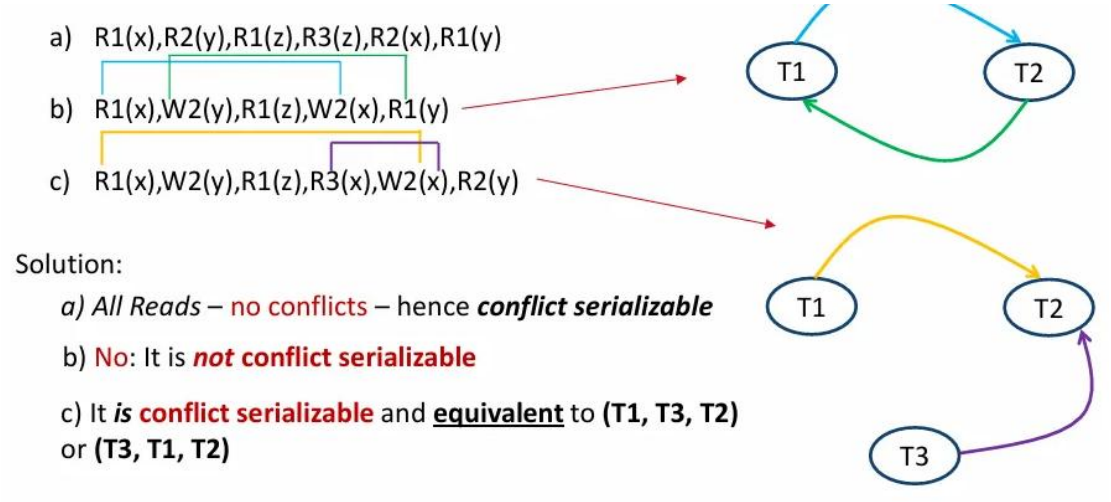
c) whether the execution is conflict serializable or not.



只用确定有 LOOP 就行，不需要识别出所有的 confilct pair

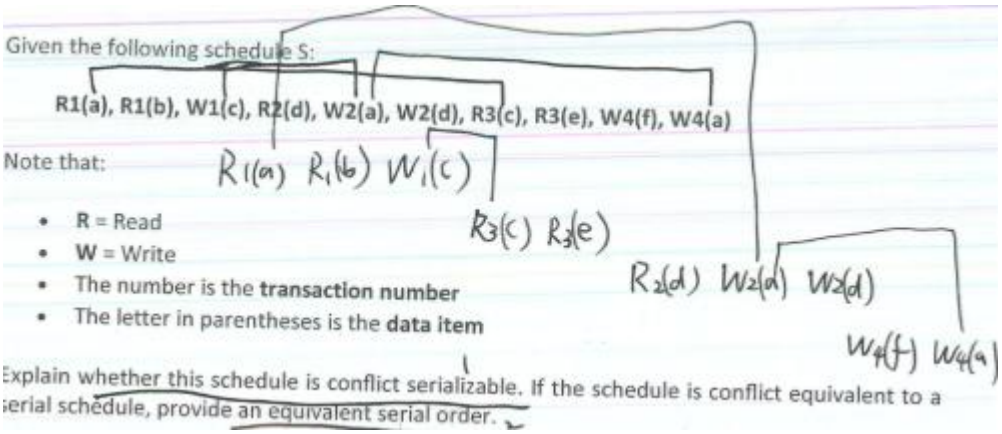
SELECT 只是 READ ROW

UPDATE，DELETE 是 READ+WRITE ROW

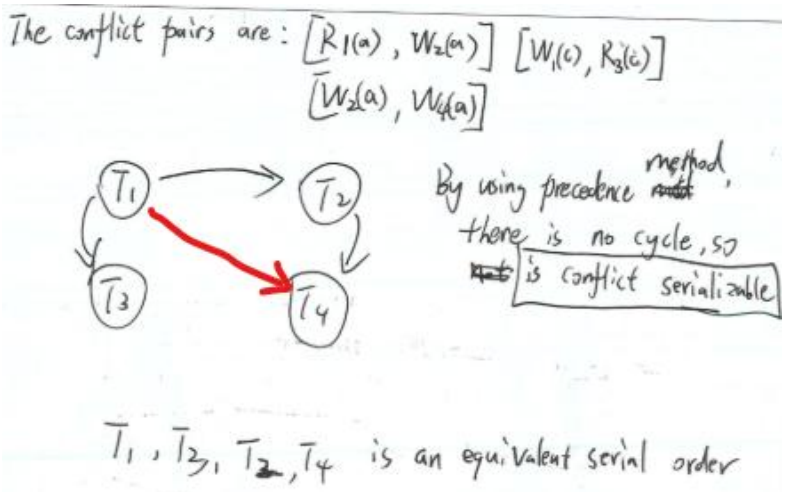


可以看到 conflict pair 对于 Reaction 的出现顺序有要求

T1 和 T3 都要在 T2 前面



这里 conflict pair 漏掉了 R1(a) and W4(a)



Example: swap 3&4 (both read), swap 5&6 (different item), then swap 4&5 (different item) →
Result if serial (T1, T2), conflict serializable

	Original		Step 1		Step 2		Step 3	
	T1	T2	T1	T2	T1	T2	T1	T2
1	R(A)		R(A)		R(A)		R(A)	
2	W(A)		W(A)		W(A)		W(A)	
3		R(A)	R(B)		R(B)		R(B)	
4	R(B)			R(A)		R(A)	W(B)	
5		W(A)		W(A)	W(B)			R(A)
6	W(B)		W(B)			W(A)		W(A)
7		R(B)		R(B)		R(B)		R(B)
8		W(B)		W(B)		W(B)		W(B)

Exercise 3. Transaction Isolation experiment

In this exercise, we are going to give you some intuition of transaction management. You will begin by opening pgAdmin in two windows.

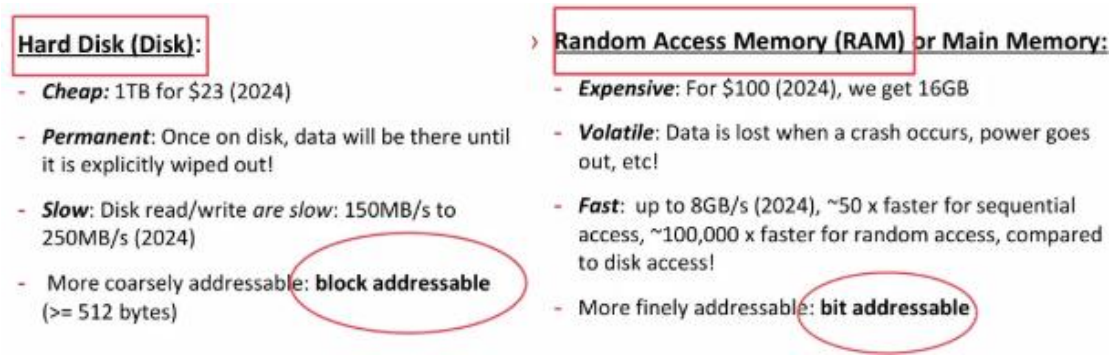
Execute the following two transactions line-by-line as seen in the following figure (press 'Execute' after each line) and write down the results of the SELECT COUNT() statements:

Window 1	**Window 2**	**Count**
BEGIN TRANSACTION;	BEGIN TRANSACTION; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	in this transaction Only see before this transaction
SELECT COUNT(*) FROM student;		? N
	SELECT COUNT(*) FROM student;	? N
INSERT INTO student VALUES (); //insert test values		
SELECT COUNT(*) FROM student;		? N+1
	SELECT COUNT(*) FROM student;	? N
COMMIT;		
SELECT COUNT(*) FROM student;		? N+1
	SELECT COUNT(*) FROM student;	? N
	COMMIT;	
	SELECT COUNT(*) FROM student;	? N+1

now can see outside

Week11

Physical storage medium



数据库最主要的性能瓶颈在于数据的存取，而不是 CPU 的计算，所以研究主要集中在 secondary storage

Permanent storage (external) — secondary storage

- nonvolatile or long-term
- relatively cheap
- 包含 HDD 和 SSD
  - Magnetic disk (Hard Disk Drive - **HDD**): HDD devices are slower, but they have a large storage capacity.
  - Solid-State Drive (**SSD**). SSD devices are faster, but they also cost much more, and life span is shorter

Transient storage (internal) — primary memory

- volatile or short-term
- relatively expensive
- 包含 Main Memory 和 Cache
  - **Main memory** is used to store data that is being used for on-going computations
  - **Cache** is used to store data in very fast computer memory chips to speed up computations
    - size is smaller than main memory size

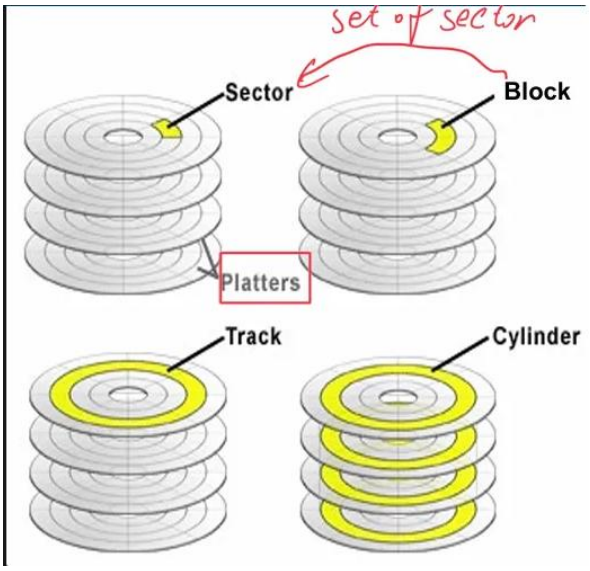
Tertiary storage

- Magnetic cartridge tapes (tape silos): only sequential access
- Optical disk (juke boxes): random access but much slower than HDD.

Disk (magnetic disk, secondary medium)

- 1KB = 1024 Bytes
- 1MB = 1024*1024 Bytes

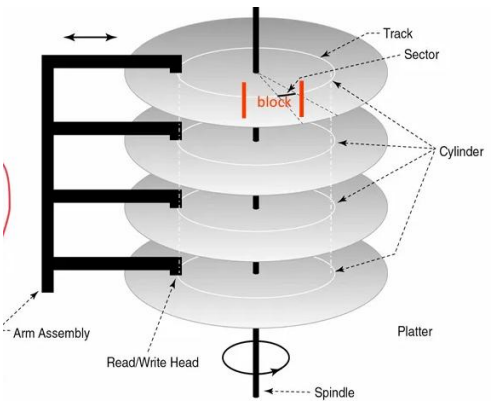
Structure



Block

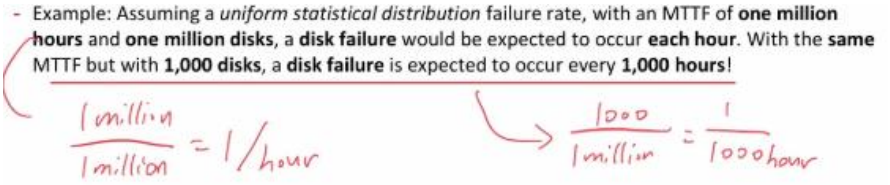
- size: set by the operating system (OS) when the disk is initialized;
  - 4096 bytes (4K), 8K
- usually spans more than one sector
- **Blocking factor**:
  - b = block size / record size
  - Record spanning: happens when b<1. Means "record size is bigger than the block size"; It is not good design when this happened

Time for disk read/write



- **Seek time**: the arm assembly is moved in or out to position a read/write head on a desired track；Go to the right track
- **Rotational delay**: The platters spin; wait for platters spin to the right place
- **Transfer time**: Reading/writing the data

MTTF (Mean Time to Failure)



MTTF is a statistical value that describes the average operating time until the disk fails. **helps to estimate how regularly failures could occur.**

Key approaches for physical disk-based transfer

Block transfer

- Data is always transferred by blocks to minimize transfer time
- A disk block is a set of contiguous sectors from a single track of one platter
- Block sizes typically range from 4k bytes to 32k
- 减少 transfer time

Cylinder-based

- data in relations is likely to be accessed together.
- Principle: store relation data on the same cylinder:
    - blocks in the same cylinder effectively involve only one seek and one rotation latency.因为指针的位置不用动
- Excellent strategy if access can be predicted
- 减少 seek time

Multiple disks — RAID

- disk drives continue to become smaller and cheaper.
- Idea: use multiple disks to support parallel access - also enhances reliability.
- 减少 transfer 和 seek time

RAID: Redundant Array of Inexpensive (Independent) Disks

- 通过把数据分布在多个廉价磁盘上，实现更高的性能（并行访问）和更好的可靠性（冗余备份）。
- Load balance multiple small accesses to increase throughput
- Parallelize large accesses so the response time is reduced
- RAID 级别高 ≠ 一定好。

RAID级别	条带化方式（Striping）	冗余方式	关键特点
RAID 0	数据分条写入多个盘（块/位级）	无冗余	仅提升性能，不提供容错能力
RAID 1	无条带化	镜像：2份数据拷贝	高可靠性，性能提升（读取）
RAID 2	位级条带化	多盘奇偶校验（内存式ECC）	罕见，复杂，适合高可靠性要求环境
RAID 3	位级条带化	单一奇偶校验盘	连续数据块传输，较适合大文件顺序访问
RAID 4	块级条带化	单一奇偶校验盘	适合并行读取，但写操作受限
RAID 5	块级条带化	奇偶校验分布在所有盘上	读写平衡，可靠性高，广泛使用
RAID 6	块级条带化	双重奇偶校验分布在所有盘上	可容忍2块硬盘同时故障，安全性最高

RAID级别	特点	适合场景
RAID 0	成本最低，没冗余	只看性能、临时数据、不重要的缓存
RAID 0+1	写入小文件时高效，双重镜像	小文件频繁写入，需性能又要容错
RAID 3	位级条带，单一奇偶校验，开销小	大文件顺序读写，不适合小文件随机访问
RAID 5	块级条带，分布式奇偶校验，读写均衡	一般企业存储、读多写少的大块数据
RAID 6	块级条带，双奇偶校验，最高可靠性	容错最高，适合关键业务或数据库

Disk scheduling — Elevator approach

- Principle: schedule **readings** of requested blocks **in the order** in which they appear under the disk head.
- does not work well when there are only a few requests
- 减少 seek time

Elevator approach

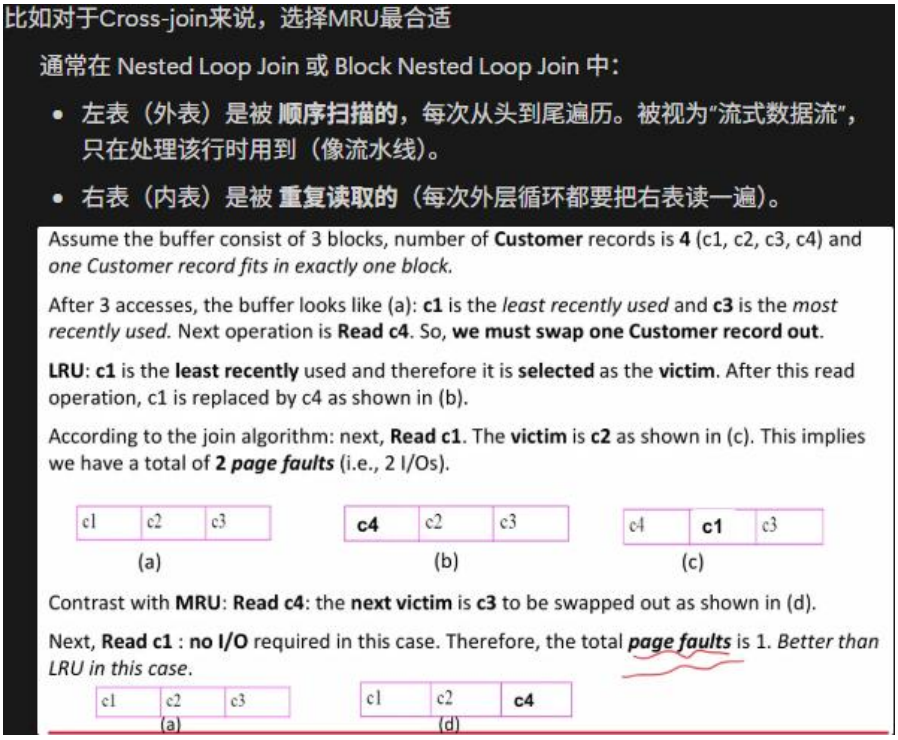
- The arm moves towards one direction serving all requests on the visited tracks;减少磁盘的寻道时间（seek time）
- 假设磁头像电梯一样：
- 先朝一个方向移动（比如从外向内），按顺序处理所有沿途的请求。
- 到达最里层（或最外层）后，再反向移动，继续处理请求

Prefetching/double buffering

- keep as many blocks in memory as possible to reduce disk access. speed-up access by pre-loading needed data.
- requires extra memory
- 减少 seek delay, Rotary delay

Buffer (RAM, Primary medium)

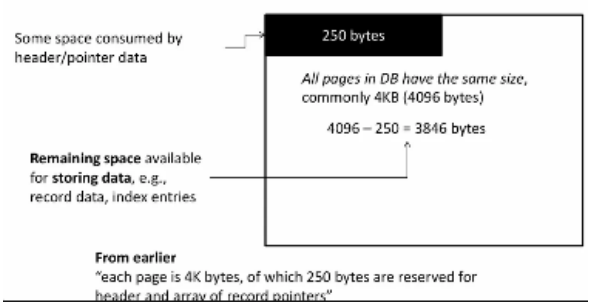
- Used to store database blocks
- provide virtual memory for operations to be performed in main memory
- Because I/Os are expensive, we try to keep as many blocks as possible in memory for later use
- **LRU (Least Recently Used)**: Replace the least recently used block in main memory when memory is full.
- **MRU (Most Recently Used)**: Replace the most recently used block in main memory when memory is full.



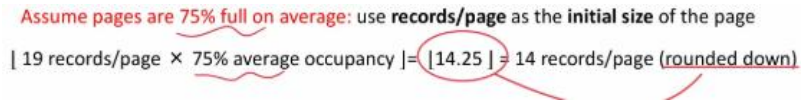
Calculating space and time

- A table → a file
- Each file is consists of one or more pages
- Each page is consists of one or more blocks

    - we will **assume** that a **page consists of a single block** unless otherwise indicated
- page is the smallest data unit in main memory
- 注意事项
    - 一般来说 header 会占用 250 bytes，但是也要根据题目进行变化



    - 注意题目允不允许 record spanning.
    - Page size 默认是 4K，但是会随题目变化
    - record per page 要用 floor
    - page for table 要用 ceil
    - occupancy rate 要在 records per page 进行乘，并且 round down



Data Access Method

Access method have no impact on sql semantics, but have impact on accessing time

- Unordered (Heap) Files — Linear Scan
- Sorted Files — Binary Search
- Indexes (B+ Tree) — Index Scan

Unordered (Heap) Files — Linear Scan $O(N)$

- a record can be placed anywhere in the file wherever there is space (random order)
- suitable when the typical access is a file scan retrieving all records.
- On average half of all pages in a file must be read, but in the worst case the whole file must be read.
- Range search need to scan whole file

Linear Scan (sequential search): For each page,

1. Load the page into main memory (cost = 1 I/O);
2. Check each record in the page for match;

Assume we need 140,351 pages to store our Table: How many page I/Os are needed to find records for:

- **SELECT** * **FROM** Relation **WHERE** tuplekey=715;

› For **equality search**, if tuplekey is **unique**, so can terminate on first match.

› If a matching record is present, it will on average look through half of all pages, so we require 70,176 I/Os

› For **zero matching records** or **non-unique** attribute, it needs to check every record, so we require 140,351 I/Os

How many page I/Os (out of 140,351 pages) are needed to find records for:

- **SELECT** * **FROM** Relation **WHERE** attribute1 **BETWEEN** 100 **AND** 119;

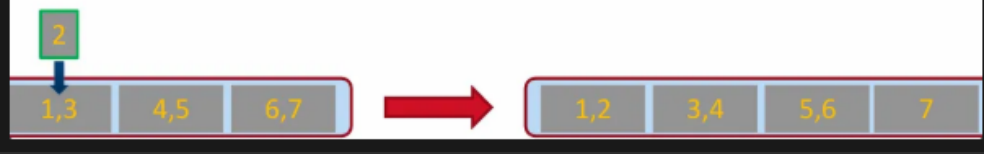
› For **range search** need to check each record, so 140,351 I/Os

可以看到没有 match 和 range query 都需要 scan 全部；如果有匹配的，则平均需要一半（也是 ceil）

Sorted Files — Binary Search $O(log_2 N)$

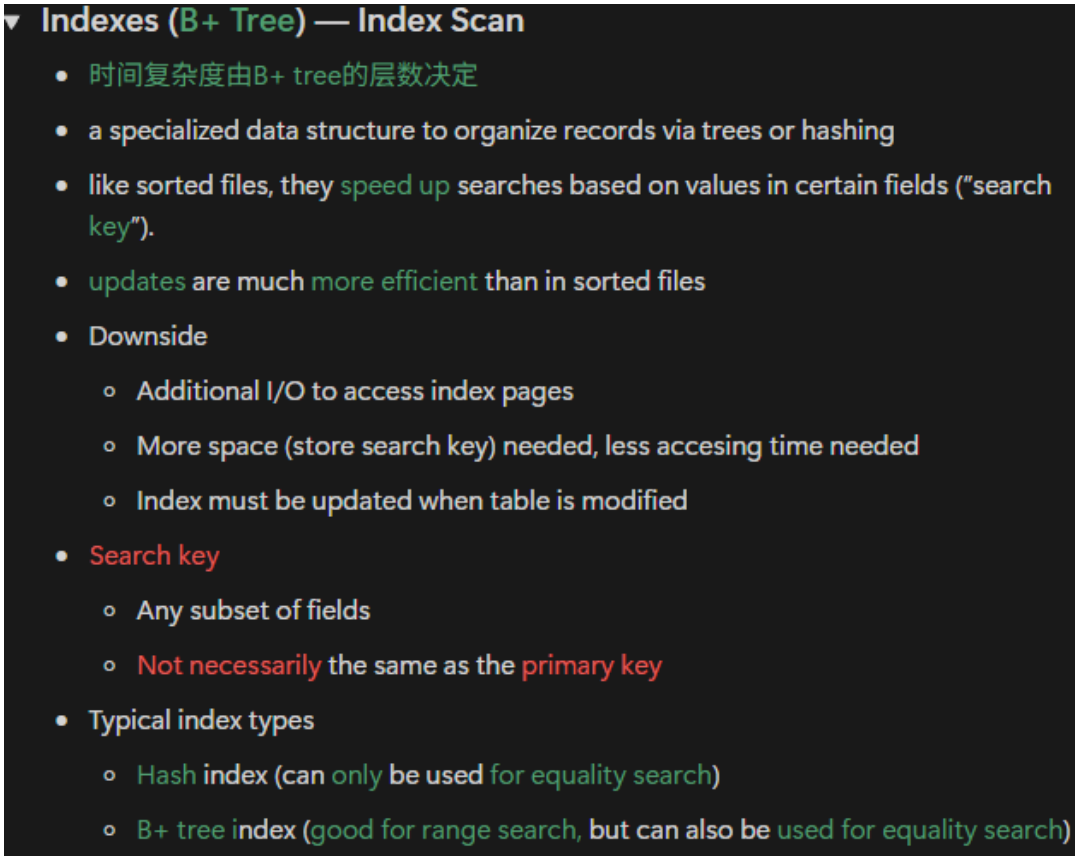
- store records in sorted order, based on the value of the search key of each record
- may speed up searches using binary search, however updates can be very expensive

- After the **correct position** for an **insert** has been determined, it **needs to shift** all **subsequent records** to **make space** for the **new record**



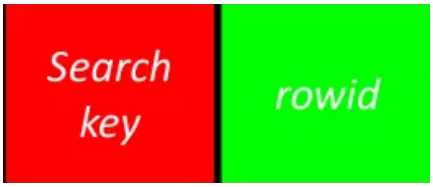
If B = 140,351 pages, the I/O cost for retrieving the page containing *first record* of *Table* is $\log_2 B = \log_2 140{,}351 = 18$

这里的 log 需要 ceil，并且是对 page 进行的操作



B+ Tree Index (A type of Index Scan Algorithm)

- Not physically sorted
  - cannot use binary search
- 题目一般要求“Pages are logically sorted”
- 每个 Entry 由 search key + rowid 组成



  - search key: 一般需要计算
  - rowid: 一般由题目给出
  - 2 者相加得到 Entry 的大小，再根据 page 的大小知道我们需要多少面来存储 Entry

- **我们对于每个 attribute 都要有自己的 index tree**，否则查询的时候还是无法实现快速查询

SELECT * FROM Relation **WHERE** tuplekey=715;

› For **equality search** on tuplekey, we can use the related index:

1) Load index root into main memory (cost 1 I/O);
2) Find location of matching page in next level;
3) Load matching next level page (cost 1 I/O);
4) Find location of matching page in following level;
5) Load matching page on following level (cost 1 I/O);
6) Find location of matching record page in leaf level;
7) Load matching record page (cost 1 I/O);
8) Check each record in the record page for match.

› Total of **4 I/Os** vs **70,176 I/Os** for heap file

different

SELECT * FROM Relation **WHERE** ***attribute1*** **BETWEEN** 100 **AND** 119;

› For **range search** on **attribute1**, the index above is of **no use. Why? We still need to check each record page sequentially**, so **140,351 I/Os**: Alternative: *build another index on attribute1*.

- **Composite search key**
  - Index with search key (age, salary) ≠ Index with search key (salary, age)

Given an **index** with search key **(age, salary)**

- Can we use the index to lookup search condition "age = 22"? YES!
- Can we use the index to lookup search condition "age >= 22"? YES!
- Can we use the index to lookup search condition "salary = 2000"? NO!
- Can we use the index to lookup search condition "salary >= 2000"? NO!
- Can we use the index to lookup search condition "age = 22 AND salary = 2000"? YES!
- Can we use the index to lookup search condition "age = 22 AND salary >= 2000"? YES!

	age	salary
r2	21	2000
r5	22	1000
r1	22	2000
r4	22	3000
r3	23	2000

Sorted order on search key (age, salary)

TUT11

Calculating space and time

Table size: assume there are **2,000,000 records** in the table **Relation** and each **record** is **200 bytes** long, including a *primary key* **tuplekey** of **4 bytes**, an *attribute* **attribute1** of **4 bytes**, along with **other attributes**.

General features: assume that each **page size** is **4K bytes**, of which **250 bytes** are reserved for *header* and *array of record pointers*.

Questions:

›*How many bytes are required for a record?*

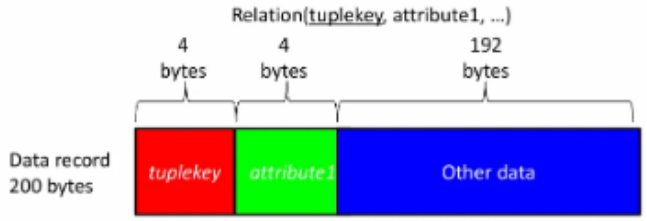
›*How many records can fill a page?*

-We assume that **no record is ever split** across pages, i.e., ***no record spanning***.

›*How many pages are required to store the table?*

Q1

"each **record** is **200 bytes** long, including a primary key called **tuplekey** of 4 bytes, an attribute called **attribute1** of 4 bytes, and also other attributes"



Q2

$\lfloor 3846 \text{ bytes/page} \div 200 \text{ bytes/record} \rfloor = \lfloor 19.23 \rfloor$ = **19** records/page **plus** 46 *remaining* bytes

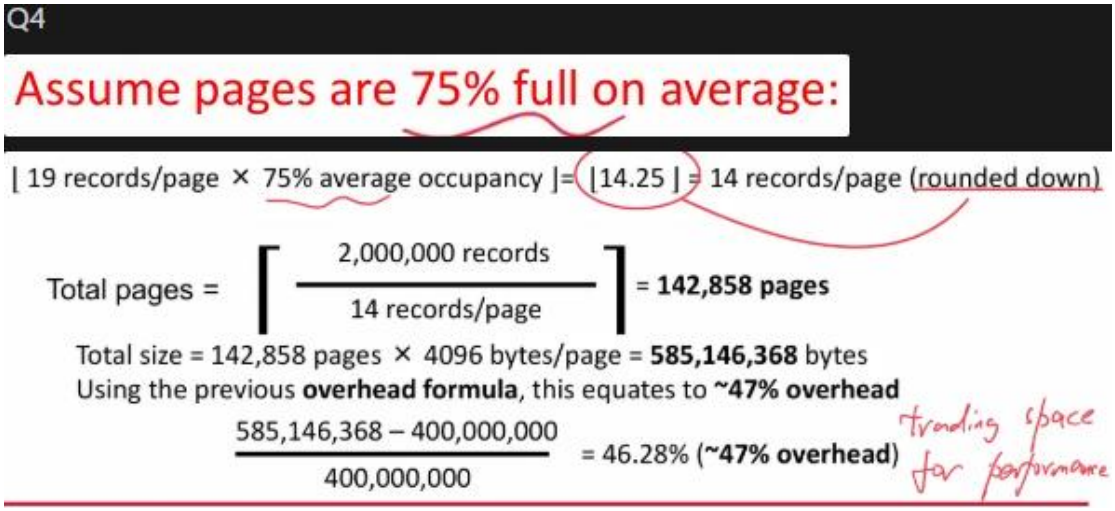
Q3

$$\left\lceil \frac{2{,}000{,}000 \text{ records}}{19 \text{ records/page}} \right\rceil$$ = **105,263** full pages **plus 1 page** containing **3** *remaining* **records = 105,264**

Total size of the required pages = **105,264** pages × **4096** bytes/page = **431,161,344** bytes

Actual **Table** size = 2,000,000 x 200 = **400,000,00**0 bytes

$$\textbf{Overhead} = \frac{431{,}161{,}344 - 400{,}000{,}000}{400{,}000{,}000}$$ = **7.79%** (~8% overhead)



Another Calculating space and time Example

Table has 2,000,000 records (rows), each record is 200 bytes, including a primary key tuple-key of 4 bytes, an attribute 1 of 4 bytes, along with other attributes

Each page is 4k bytes, of which 250 bytes are reserved for header and array of record pointers

Calculations:

- Calculate size of each record: 200 bytes
- Records per page (assuming no record spanning): assume 100% occupancy
  - Space available for data storage = 4 * 1024 – 250 = 3846 bytes
  - Number of records per page: $\left\lfloor \frac{3846}{200} \right\rfloor = \lfloor 19.23 \rfloor = 19$ records per page
  - Remaining space of each page: $3846 - 19 * 200 = 46$ bytes
- Number of pages for the table
  - Number of pages: $\left\lceil \frac{2{,}000{,}000}{19} \right\rceil = \lceil 105{,}263.16 \rceil = 105{,}264$
  - Space: $105{,}264\ pages * 4KB = 421{,}056\ KB \approx 411\ MB$
  - Number of records on last page: $2{,}000{,}000 - 105{,}263 * 19 = 3$
- Overhead:
  - Total size of required pages: $105{,}264 * 4 * 1024 = 431{,}161{,}34$ bytes
  - Actual table size: $2{,}000{,}000 * 200 = 400{,}000{,}000$ bytes
  - Overhead = $\frac{(431{,}161{,}344 - 400{,}000{,}000)}{400{,}000{,}000} = 7.79\%$
- Pages with a given fill factor: 75% on average
  - Updated record per page: $\left\lfloor \frac{3846}{200} * 0.75 \right\rfloor = \lfloor 14.42 \rfloor = 14$ records per page
  - Total pages: $\left\lceil \frac{2{,}000{,}000}{14} \right\rceil = \lceil 142{,}857.14 \rceil = 142{,}858$ pages
  - Total size: $142{,}858 * 4 * 1024 = 585{,}146{,}368$ bytes
  - Overhead: $\frac{585{,}146{,}368 - 400{,}000{,}000}{400{,}000{,}000} = 46.28\%$

B+Tree

给出

- logically sorted data file consisting of 140,351 pages
  - 也就是说leaf node有140,351个
- pointer由4bytes search key 和 4bytes rowid组成
- 给出每个页面只有75%被用到

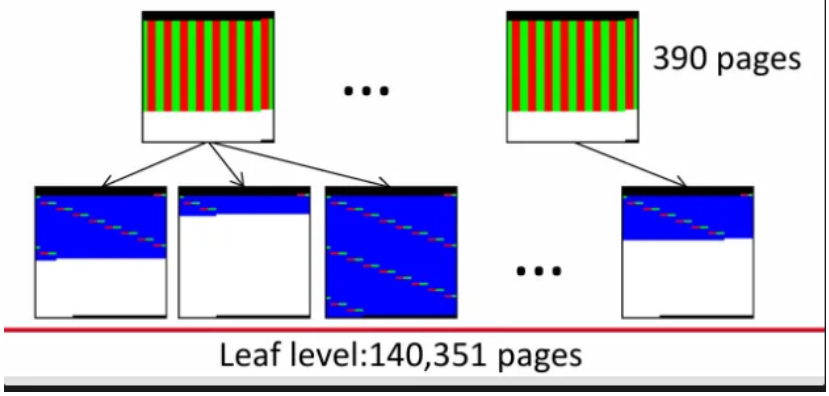
⌊3846 bytes/page ÷ 8 bytes/entry⌋= 480 entries/page

The average number of index entries in an index page is 480 × 75% = 360

因为每个page可以存储360 bytes，所以总共要用

The **number** of **index pages** is ⌈140,351/360⌉ = 390 (rounded up)

个page存储index，即



接下来计算剩下的layer

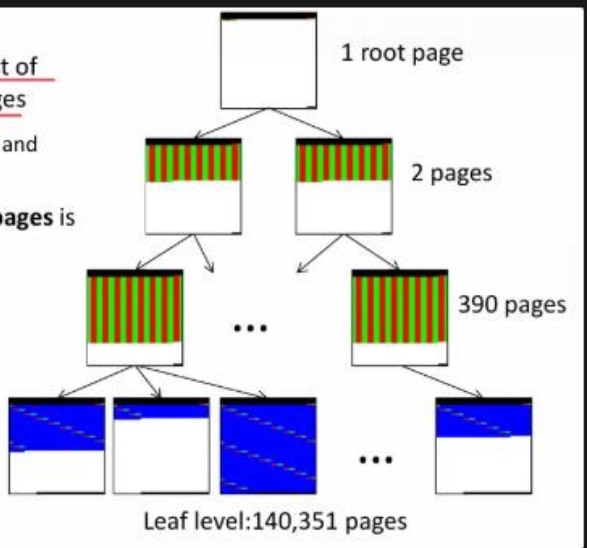
an index page is 480 × 75% = 360

The next-level index would then consist of 390 index entries, and 390/360 = 2 pages

- The **root level** consists of 2 index entries, and 2/360 = 1 page

Therefore, the **total number** of **index pages** is 390 + 2 + 1 = **393 index pages**

393/140,351 = **0.2% increase**



相比于不使用index，这里的space增加量为0.2%

可以说是 this index has 3 levels, including the root page and excluding the leaf (record) pages.

Another B+ TREE Example

B+ Tree construction: file has 140,351 pages

- Create a logically sorted file: All records within a page are physically sorted on the search key
- Create one index entry for each page: <search key, rowid>
  - Size of entry: size of search key 4 bytes + size of rowid 4 bytes = 8 bytes
- Fit index entries into pages:
  - Number of index entries in each page: $\left\lfloor \frac{3846}{8} \right\rfloor = 480$ entries per page
  - Number of entries with fill factor (75%): $480 * 0.75 = 360$ entries
  - Number of index pages: $\frac{number\ of\ pages}{360} = \left\lceil \frac{140{,}351}{360} \right\rceil = 390$ pages

Build B+ Tree index:

- Pages required for 2nd level: $\frac{number\ of\ entries\ in\ level\ down}{360} = \lceil \frac{390}{360} \rceil = 2$
- Pages required for 3rd level: $\left\lceil \frac{2}{360} \right\rceil = 1$ → reached 1, this is the root level
- Total number of index pages: $390 + 2 + 1 = 393$
- Increase: $\frac{393}{140{,}351} = 0.2\ \%$ increase in the number of pages required

Final tree:

- Root index: 1 page storing 2 index pointers
- 1st level index: 2 pages storing 390 index pointers
- 2nd level index: 390 pages storing 140,351 pointers to pages
- 3rd level data: 140,351 pages of data

综合起来

Exercise 1. Calculating space and time

4X1024=4096

Suppose we have a table Rel1(A, B, C). Each A field occupies 4 bytes, each B field occupies 12 bytes, each C field occupies 8 bytes. Rel1 contains 100,000 records. There are 100 different values for A represented in the database, 1000 different values for B, and 50,000 different values for C. Rel1 is stored with a primary B+ tree index on the composite key consisting of the pair of attributes (A,B); assume this index has 2 levels, including the root page and excluding the leaf (record) pages. Assume that in this database, each page is 4K bytes, of which 250 bytes are taken for header information. Record locations within the index use a 4-byte *rowid*. Assume that reading a page into memory takes 150 msec, and that the time needed for any query can be approximated by the time spent doing disk I/O.

1a) Calculate the space needed for the records/data of Rel1.
Hint: How much space is used by a single record? How many records per page? How many pages for the table? Then convert this back to space. [Note that each record is not split across pages, and each page has a header.]

Each page has 4096 – 250 = 3846 bytes available to store records. As each record needs 4+12+8 = 24 bytes, we can fit ⌊3846/24⌋ = 160 records in a page (note that we round down to an integer, as one doesn't store just part of a record in a page). Since Rel1 has 100,000 records, we need ⌈100,000/160=625⌉ pages to store the table; measured in bytes it is approximately 2.4 MB. (Note that our simplifying assumption here is 100% occupancy and you should always state your occupancy assumption).

1b) Calculate the time taken to perform a table scan (i.e., linear scan) through the table Rel1.
Hint: How many pages are needed to make up the space occupied by the table? How many pages will be read from disk during a table scan?

To do a table scan we must fetch each of the 625 pages of the table; measured in seconds, this takes 625*150 = 93750 ms = 93.75 s or approximately 1.5 minutes.

这里如果是 log 的话，则应该为 ceil(log_2(625))*150

1c) Calculate the time taken, using the primary index, to execute the following query. Assume the selectivity of range condition on B is 10%.

```
SELECT C
FROM Rel1
WHERE A = 'AQG' AND (B BETWEEN 'WPQ' AND 'XYZ');
```

some guesses. We know that there are 100 different A values, so we can guess that 100,000/100 = 1000 data records have the A value 'AQG'. As the selectivity of range condition on B (i.e., the fraction

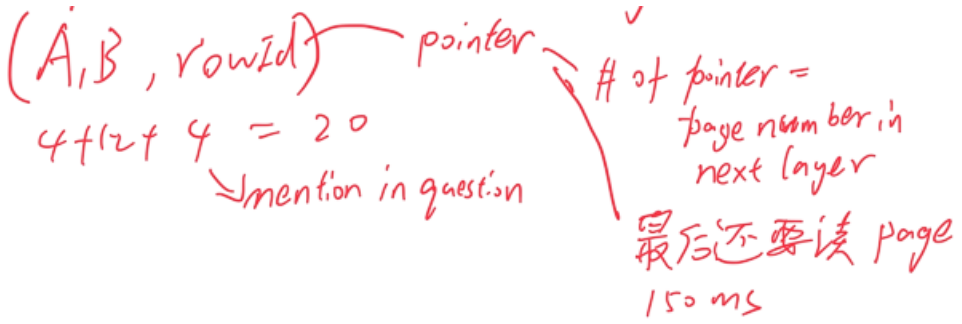
这里关于 data page，因为我们只用搜寻 100 个 record，而一个 data page 最多可以存储 160records，所以我们的 data 最多跨 2 页；但是如果我们要搜寻 162 个 record，则我们的 data 最多可以跨 3 页

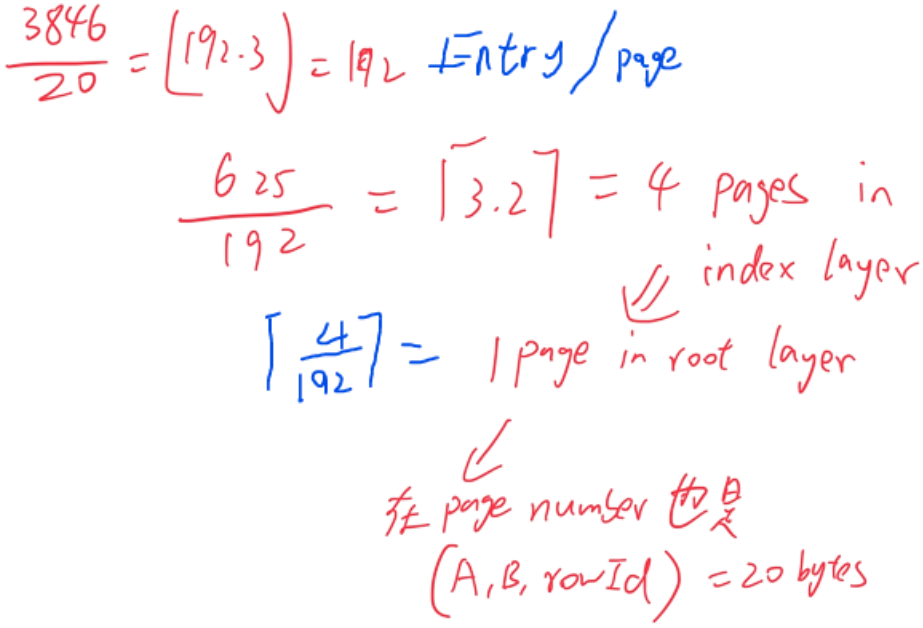
of records satisfying the range condition on B) is 10%, we can guess that 1000*10%=100 records will satisfy the matching condition, and be fetched from the data pages. Since each page stores 160 records, we probably only need to fetch 1 or perhaps 2 pages of data to get all the matching records; let's guess that we only fetch 1 data page. Overall, we will need to fetch 2 index pages, and 1 data page; that is, we read 3 pages from disk, taking 3*150=450ms.

考试还可能不会给出 B+tree 的 level

这里之所以用 A，B 作为 rowid 是因为题目说了，而不是因为他们是 PK

下面看看怎么计算 B+ tree 的 level，用来判断多少个 idx page 需要 retrieve





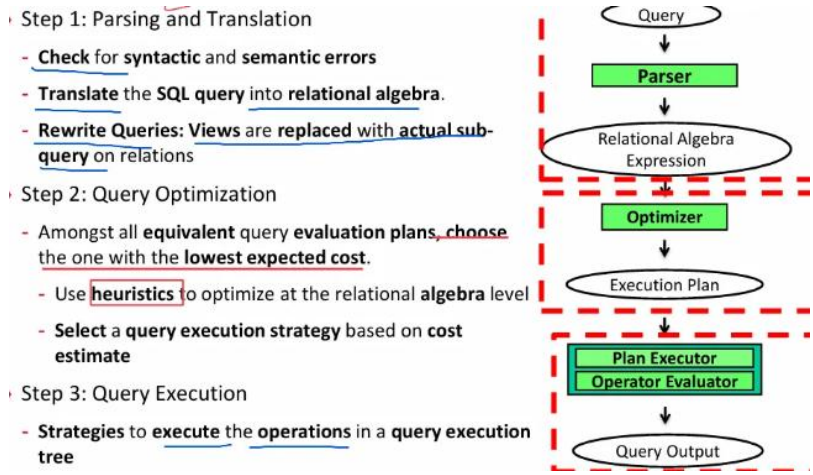
I/O 次数计算

I/Os needed for different access path: if total pages in a file = 140,351 and page size = 4KB

- Linear scan on heap:
  - Equality search:
    - If search key is unique and exists: $\left\lceil \frac{140,351}{2} \right\rceil = 70,176$ I/Os
    - If non-unique or no match: 140,351 I/Os
  - Range search: 140,351 I/Os
- Binary search on sorted files:
  - Height $\log 140,351$ + additional pages containing retrieved records
- Index scan on B+ Tree:
  - Equality search: height of B+ Tree (index pages) + additional pages containing retrieved records
  - Range search if first item is search key: height of B+ Tree + records matched with subsequent attributes – additional pages containing retrieved records
    - Table R(A,B,C) has 100,000 records, A has 100 different values, B has 1000, C has 50,000
    - B+ Tree index on composite key (A,B), 2 levels of index + record pages
    - Fits 160 records per page, file has 625 pages, reading a page takes 150 msec
    - Calculate time to run: SELECT C FROM R WHERE A = "x" AND (B BETWEEN 'y' AND 'z'); selectivity of range condition on B is 10%
    - Time to locate the data: go through one page at each index page level: 2 pages
    - Page containing the record:
      - 100,000 records & 100 different values of A: assume $\frac{100,000}{100} = 1000$ records = 'x'
      - Of the 1000 pages, 10% matches the range search on B: $1000 * 10\% = 100$ records
      - The 100 records fit onto: $\left\lceil \frac{100}{160} \right\rceil = 1$ page
    - Total page to read: 2 + 1 = 3 pages
    - Total time = 3 pages * 150 msec = 450 msec

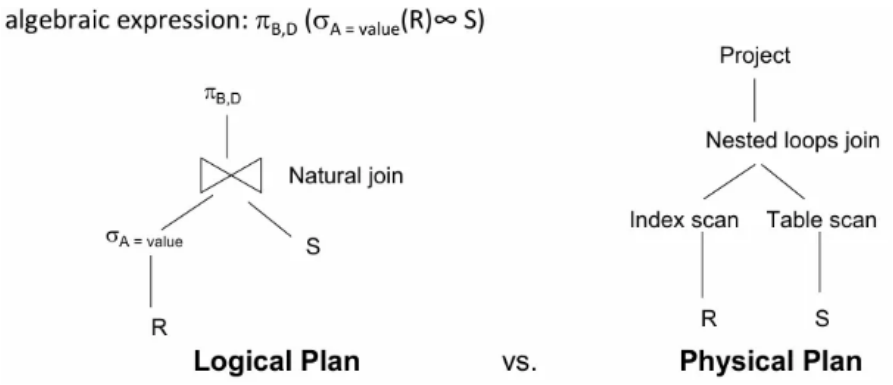
Week12

Query Processing



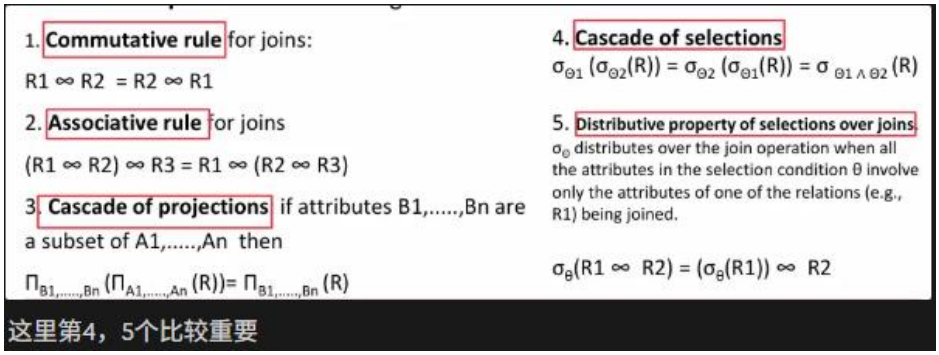
- **Parser**
  - Check for syntactic and semantic errors
  - Translate the SQL query into relational algebra.
    - relational algebra Focus of queries is mostly on the how.
    - **Leafs correspond to (base) tables**.
  - Rewrite Queries: Views are replaced with actual sub query on relations
- **Optimizer**
  - Select a query execution strategy based on cost estimate
  - choose the one with the lowest expected cost
  - 2 thrusts
    - Heuristic rules: rearranging operations in a query tree:
      - output is an efficient logical query plan
      - Heuristics: minimize the size of intermediate results (relations) using equivalent algebraic expressions.
    - Cost estimate: estimate of different execution strategies/plans to select the one with minimal
      - cost: output is an efficient physical query plan
      - minimizing the number of disk I/Os
- **Plan Executor & Operator Evaluator**

Query Optimization



Heuristic-based Optimization — Logical

Algebraic rules



Selection optimization

具体思路就是只包含必要的 attribute，即 Whenever possible, do a selection as soon as possible.

Projection optimization

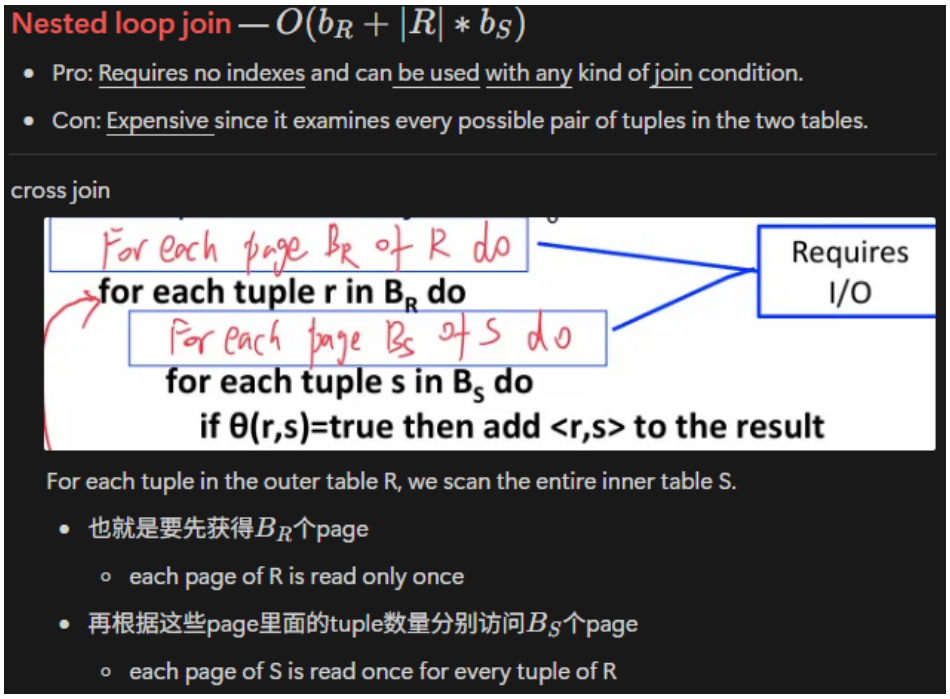
do a projection as soon as possible

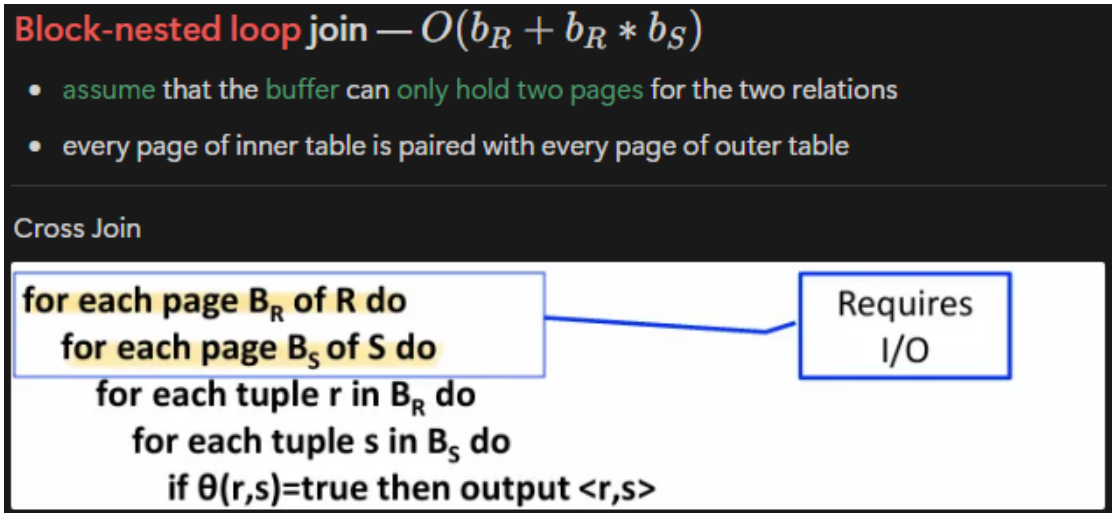
Cost Estimate Optimization — Physical

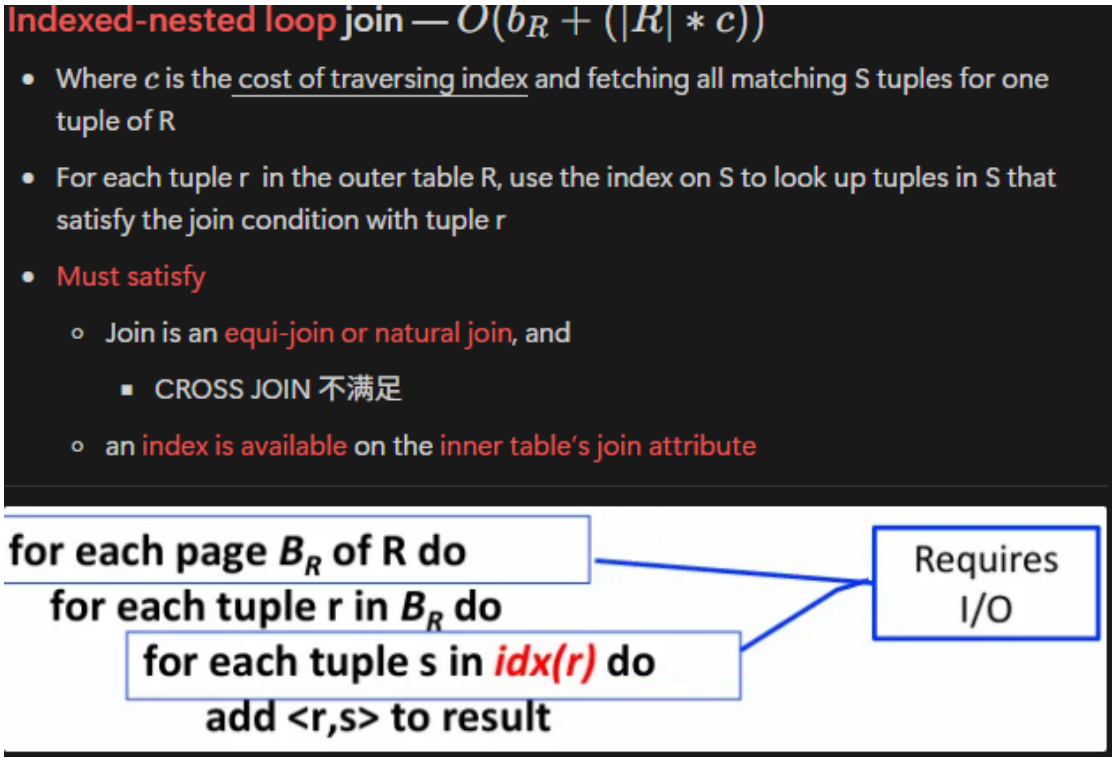
- Find an optimal plan among a set of all equivalent plans
  - minimizing the number of disk I/Os
- 一个 logical 的 access 可能对应多个不同的 physical access
- **Output** of the cost estimate optimization: **Efficient physical query plan**

Join implementation

- Based on cost estimate (i.e. I/Os) to choose implementation method
- We will use the following statistics:
    - |R|: number of tuples in R, stored in b_R pages
    - |S|: number of tuples in S, stored in b_S pages







Sort-Merge Join (Merge-Join)

Key Idea

first sort the relations by the join attribute, so that linear scans can match the corresponding values at the same time

Then how to sort?

- small dataset: quicksort
- big dataset: External Merge-Sort

External Merge-Sort

- **B** denote the buffer size (in pages).
- **N** is the size (in pages) of the file (table).
- 每个 step1 或 step2 都叫做一个 pass
    - 总的 pass 数为：$\lceil log_{(B-1)}(N/B) \rceil + 1$
    - 这里加 1 是因为第一步是一定要做的，且只做一次
    - if m >= B , then the number of passes in this step (step 2) will be greater than 1.

Steps

1. **Create** sorted ***runs***. (A **run** is the name of a **sorted subset** of the **file records**)

   Let *i* be 0 initially. **Repeatedly** do the following till the end of the file:
   (a) Read *B* pages of records from disk into buffer
   (b) Sort the in-buffer pages
   (c) Write the sorted data to run R_i; increment *i* by 1.
   Let the final value of *i* be $m = \lceil N / B \rceil$; there are ***m* sorted runs**.

2. **Merge** each contiguous group of *B-1* runs into *1* run: (*B-1*)-way merge.

3. After each merge pass, the number of runs is reduced by a factor of B-1.
   Let $m = \lceil N / B \rceil$, if ***m* > *B***, **several merge passes are required**. The **number of passes** (including the initial sorting pass) for the multiway merging is $\lceil \log_{(B-1)} (N/B) \rceil + 1$

1 Create Sorted Runs

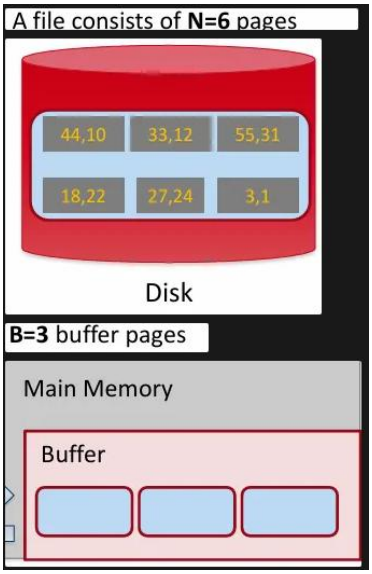
- 将大文件（数据集）**分段**，每次读取 **B 页**（这里B是缓冲区大小，页数）到内存
- 内存中做 **内部排序**（比如快速排序）
- 排序后写到磁盘，得到一个 **已排序的** run（子文件）
- 这样一直分批做下去，直到把整个文件都处理完
- 最终得到 `m = [N/B]` 个已排序的 run

2 Merge Passes

- 一次归并最多能处理 `B-1` 个 run（因为需要留 1 页做输出缓冲）
- 把 **连续的 B-1 个 run** 合并成一个更大的 run
- 每次归并 pass，run 的数量都会缩小为原来的 1/(B-1)
- 不断做归并 pass，直到只剩下一个最终的 run

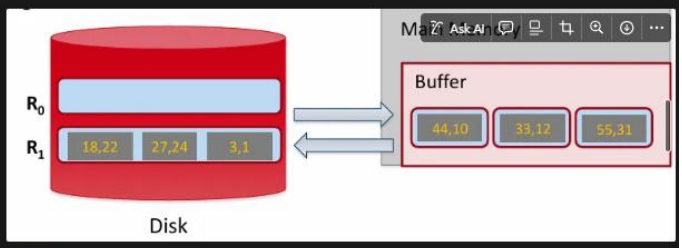
3 最终结果

- 当只剩下 1 个 run，表示整个数据集已经整体排好序
- 这个 run 就是最终的有序文件

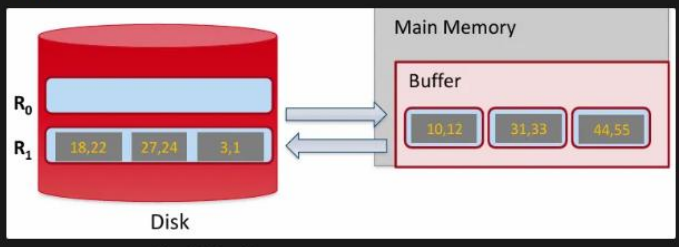


Step 1 Create Sorted Runs

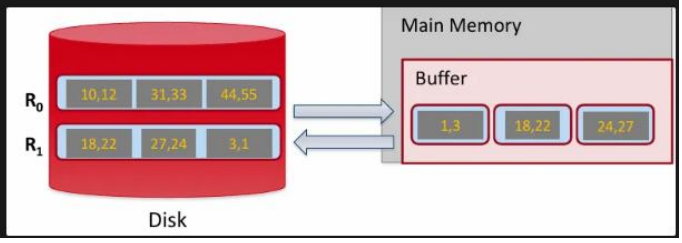
Read R0



sortR0



Read R1 + sort R1 (这里合并了步骤)

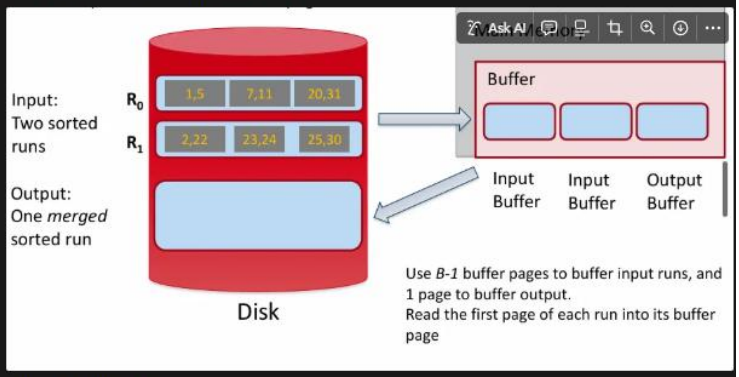


现在我们有2个sorted runs

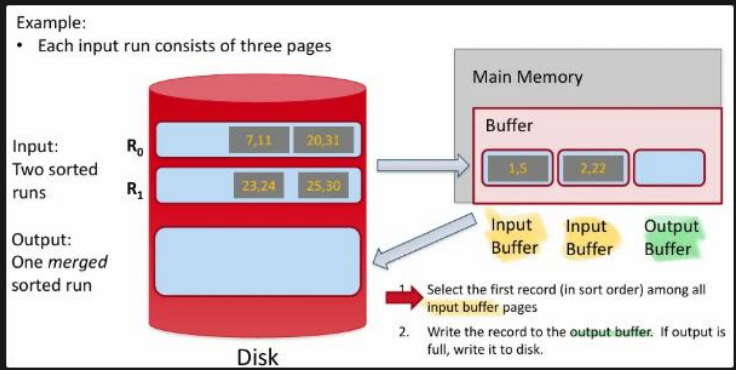
Step 2 Merge Passes

这里因为ppt出了问题，所以数字不一样

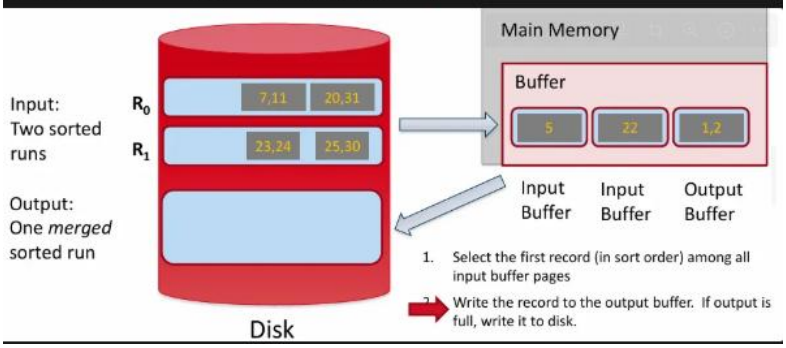
buffer保留一位作为output buffer



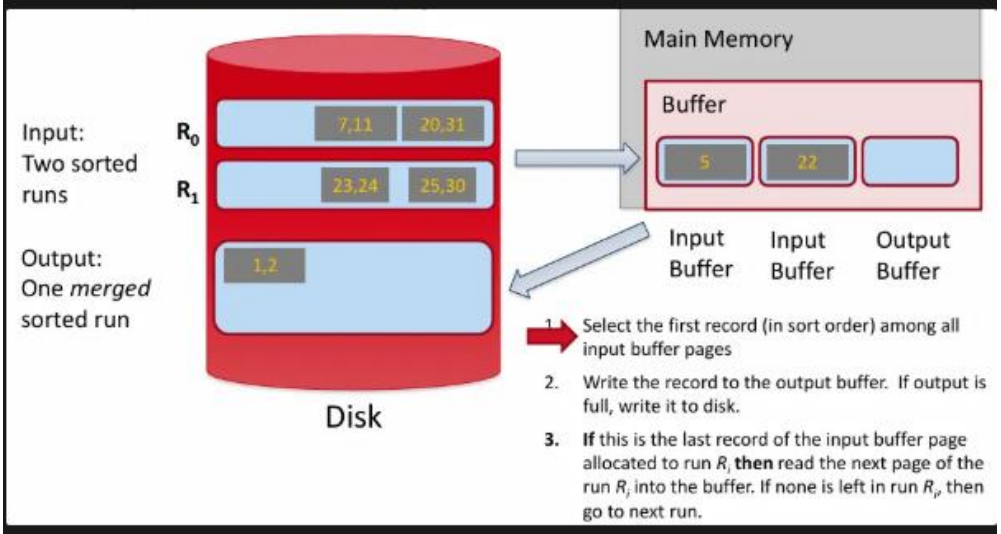
把input buffer读满

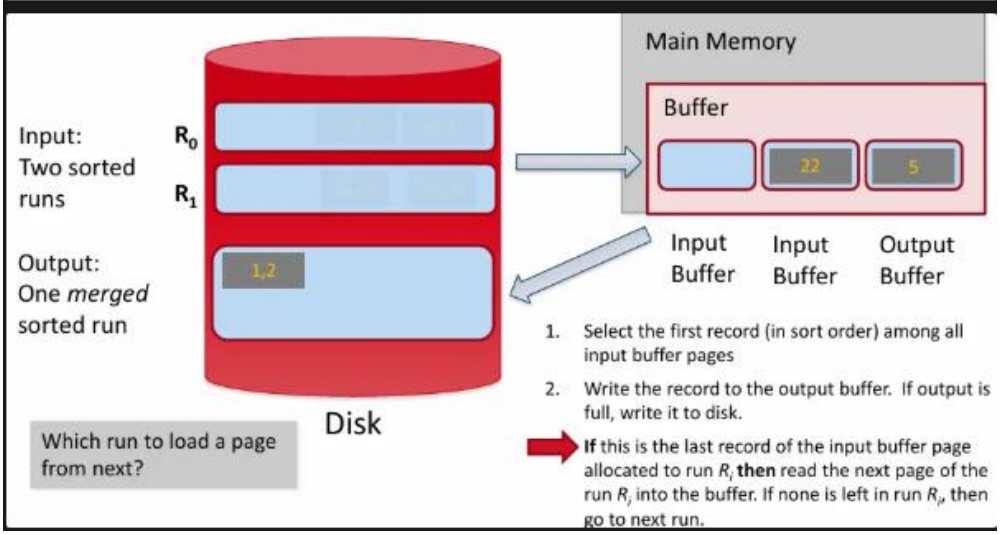


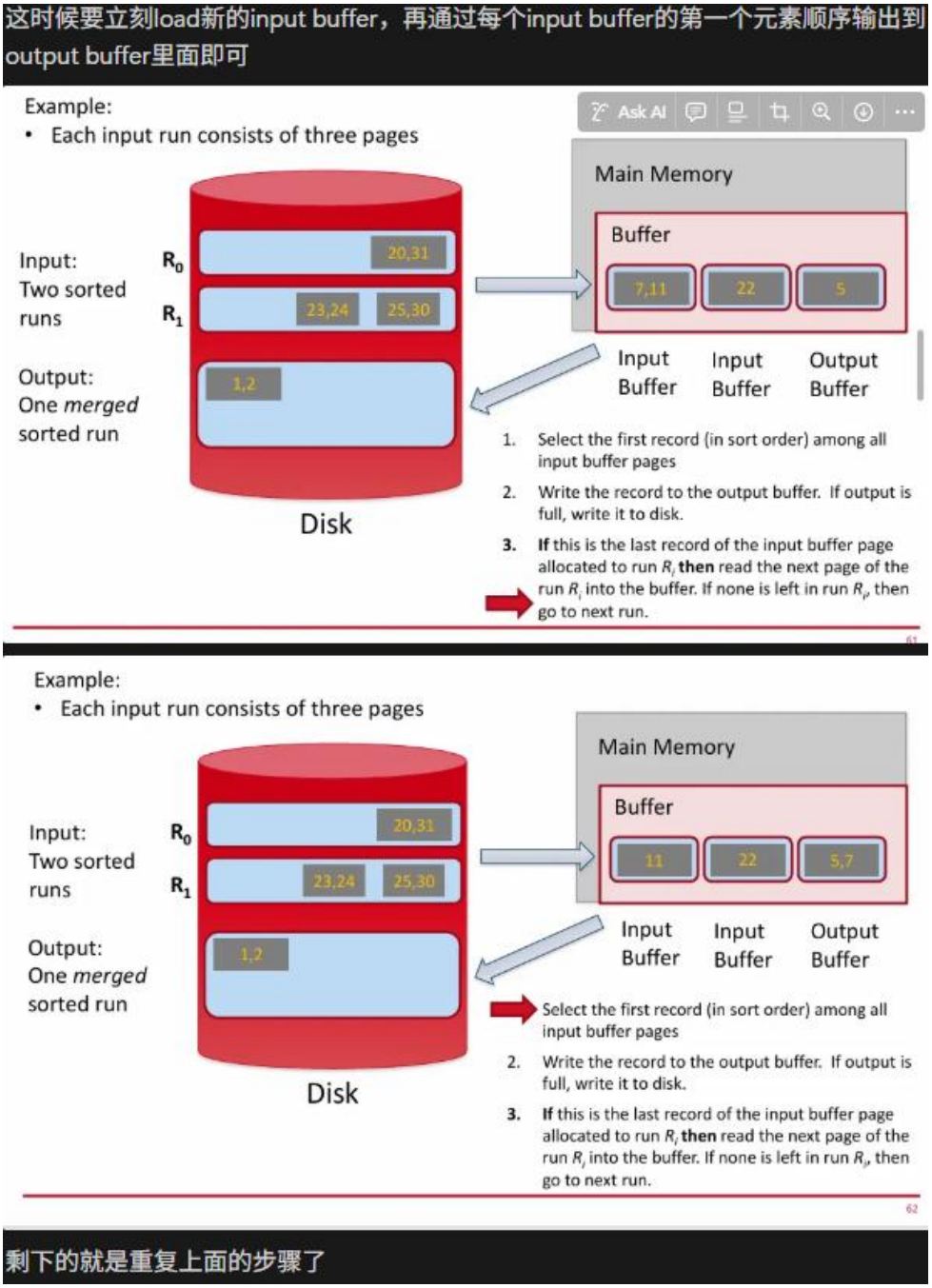
选择每个input buffer的第一位，按顺序输入output buffer



重复以上步骤，直到某个input buffer空了







Query Execution

Materialization

Def

- Output of one operator written to disk and the next operator will read it from the disk.

- starting at the lowest-level.
- Produces sorted output.

Advantage:

Can always apply materialized evaluation

Disadvantages:

Costs can be quite high

Pipelining

Def

- Output of one operator is directly input to next operator
- evaluate several operations **concurrently**
- unsorted output

Advantage

Much cheaper than materialization:

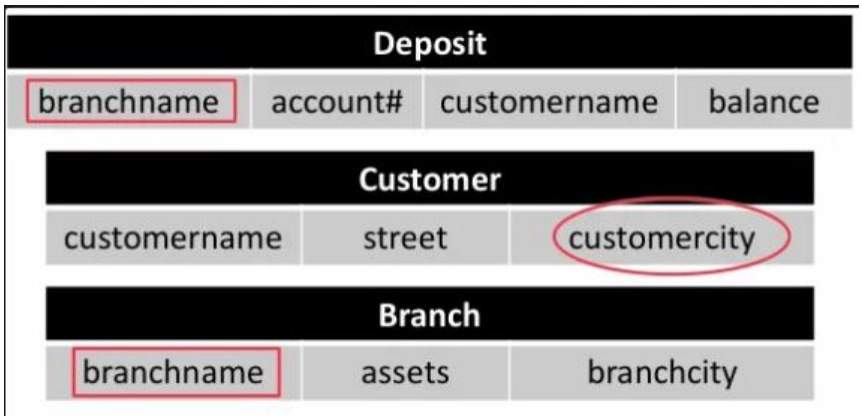
- no need to store a temporary table

Issues:

If algorithm requires sorted output, pipelining may not work well if data is not already sorted.

TUT12

Algebraic rules



- branch没有customercity和balance，所以可以用rule5进行分割

Using our previous query: *"find the assets and names of all banks which have depositors living in Sydney and have a balance of more than $500".*

This query is equivalent to the following algebraic expression:

$\Pi_{\text{Branchname, Assets}}(\sigma_{\text{Customercity=Sydney} \wedge \text{Balance} > 500}(\text{Customer} \infty \text{Deposit} \infty \text{Branch}))$

Use **Rule #5**: $\sigma_\theta(R1 \infty R2) = (\sigma_\theta(R1)) \infty R2$, to do the selection *after* doing a join on *Customer* and *Deposit*. The resulting expression is therefore:

$\Pi_{\text{Branchname, Assets}}(\sigma_{\text{Customercity=Sydney} \wedge \text{Balance} > 500}(\text{Customer} \infty \text{Deposit}) \infty \text{Branch})$.

Using **Rule #4:** $\sigma_{\Theta 1}(\sigma_{\Theta 2}(R)) = \sigma_{\Theta 2}(\sigma_{\Theta 1}(R)) = \sigma_{\Theta 1 \wedge \Theta 2}(R)$, to break up the selection condition into two selections and we get:

$\Pi_{\text{Branchname, Assets}}(\sigma_{\text{Customercity=Sydney}}(\sigma_{\text{Balance} > 500}(\text{Customer} \infty \text{Deposit})) \infty \text{Branch})$

Use **Rule #5 twice:** $\sigma_\theta(R1 \infty R2) = (\sigma_\theta(R1)) \infty R2$, to move the first and second selections to their respective relations:

$\Pi_{\text{Branchname, Assets}}(\sigma_{\text{Customercity=Sydney}}(\text{Customer}) \infty \sigma_{\text{Balance} > 500}(\text{Deposit})) \infty \text{Branch})$

Selection optimization

Break up the **selection condition** into **two selections** and we get:

$\Pi_{\text{Branchname, Assets}}(\sigma_{\text{Customercity=Sydney}}(\sigma_{\text{Balance} > 500}(\text{Customer} \infty \text{Deposit})) \infty \text{Branch})$

And then **move the second selection** past the **first join**:

$\Pi_{\text{Branchname, Assets}}(\sigma_{\text{Customercity=Sydney}}(\text{Customer}) \infty \sigma_{\text{Balance} > 500}(\text{Deposit})) \infty \text{Branch})$

Projection optimization

$\Pi_{\text{Branchname, Assets}}((\sigma_{\text{Customercity=Sydney}}(\text{Customer}) \infty \text{Deposit}) \infty \text{Branch})$

- *Branchname* is the only attribute we need in the first join

$\Pi_{\text{Branchname}}(\sigma_{\text{Customercity=Sydney}}(\text{Customer}) \infty \text{Deposit}) \infty \Pi_{\text{Branchname, Assets}}(\text{Branch})$

Nested loop join

Assume

Number of tuples of
Students ($|R|$): 1,000
Enrolled ($|S|$): 10,000

Number of pages of
Students (b_R): 100
Enrolled (b_S): 400

Student

sid	name	gender	country
1001	Ian	M	AUS
1002	Ha Tschi	F	ROK
1003	Grant	M	AUS

Enrolled

sid	uos_code	semester
1001	COMP5138	2020-S2
1002	COMP5702	2020-S2
1003	COMP5138	2020-S2
1006	COMP5318	2020-S2

students **(R) as outer table**: 100 + 1000 * 400 = **400,100 disk I/Os**

enrolled **(S) as outer table**: 400 + 10,000 * 100 = **1,000,400 disk I/Os** $(b_S + |S| * b_R)$

Block-nested loop join

Assume

Number of tuples of
Students (|R|): 1,000
Enrolled (|S|): 10,000

Number of pages of
Students (b_R): 100
Enrolled (b_S): 400

Student

sid	name	gender	country
1001	Ian	M	AUS
1002	Ha Tschi	F	ROK
1003	Grant	M	AUS

Enrolled

sid	uos_code	semester
1001	COMP5138	2020-S2
1002	COMP5702	2020-S2
1003	COMP5138	2020-S2
1006	COMP5318	2020-S2

students **(R) as outer table**: 100 + 100 * 400 = **40,100 disk I/Os**
enrolled **(S) as outer table**: 400 + 400 * 100 = **40,400 disk I/Os** ($b_S + b_S * b_R$)

Indexed-nested loop join

Assume

Number of tuples of
Students (|R|): 1,000
Enrolled (|S|): 10,000

Number of pages of
Students (b_R): 100
Enrolled (b_S): 400

Student

sid	name	gender	country
1001	Ian	M	AUS
1002	Ha Tschi	F	ROK
1003	Grant	M	AUS

Enrolled

sid	uos_code	semester
1001	COMP5138	2020-S2
1002	COMP5702	2020-S2
1003	COMP5138	2020-S2
1006	COMP5318	2020-S2

Example: if $c_1 = 4$ for the relation **S** and $c_2 = 3$ for the relation **R**

- If index on **S** is available: cost is $b_R + (|R| * c_1)$
- 100 +1,000 * 4 = **4,400 disk I/Os**

- If index on **R** is available: cost is $b_S + (|S| * c_2)$
- 400 + 10,000 * 3 = **30,400 disk I/Os**

比较不同的 Join implementation，看看哪个的 I/O 最小

Suppose we have a schema Rel1(A, B, C) and Rel2(C, D). Each A field occupies 4 bytes, each B field occupies 12 bytes, each C field occupies 8 bytes, each D field occupies 8 bytes. Rel1 contains 100,000 records, and Rel2 contains 50,000 records. There are 100 different values for A represented in the database, 1000 different values for B, 50,000 different values for C, and 10,000 different values for D. Rel1 is stored with a primary (sparse, clustered) B+ tree index on the pair of attributes (A,B); assume this index has 2 levels, including the root page and excluding the leaf (record) pages. Rel2 is stored with a primary (sparse, clustered) B+ tree index on C; assume this index has 2 levels, including the root page and excluding the leaf (record) pages.

General features: assume that each page is 4K bytes, of which 250 bytes are taken for header and array of record pointers. Assume that no record is ever split across several pages. Assume that index entries in any index use the format of (search key, rowid), where rowid uses 4 bytes.

这里我们假设 75%的使用率

Exercise 1. Block-Nested Loops Join

Consider the following query:

```
SELECT Rel1.A, Rel1.B, Rel1.C
FROM Rel1, Rel2
WHERE Rel1.C = Rel2.C AND Rel2.D = 16;
```

and consider the query plan which calculates this as follows:

Form the equi-join of Rel1 and Rel2 by using a block-nested loops join with Rel1 as the outer relation, and then filter each tuple of the join to see if the value of D is 16; if so, output the values of A, B and C from that tuple of the join.

How many page I/Os are needed to compute this plan (assume that we have only the minimal space, say 2 pages worth, for buffering in memory)?

1. Calculate the # of pages (we use 75% here)

Rel①. Rel(A,B,C) = 4+12+8 = 24 — 100,000 records

Rel②. Rel(C,D) = 8+8 = 16 — 50,000 records

- Each page is 4K, so 4×1024 = 4096 bytes
- 250 bytes for header
- So each page can have 4096 − 250 = 3846 bytes

Rel① ⌊3846/24⌋ × 0.75 = ⌊160 × 0.75⌋ = 120

Rel② ⌊3846/16⌋ × 0.75 = 0.75 × 240.375 = 180

① → ⌈100000/120⌉ = 834

② → ⌈50000/180⌉ = 278

Now we can think about the costs of the query processing: to do the block-nested loops join, we read each data page of the outer relation once, and we read each data page of the inner relation as many times as there are pages in the outer relation (because we scan the inner relation for every

page of the outer one). Note that this processing does not use or examine the higher levels of the index; only the leaves which contain the data records are scanned. Thus, the block-nested loops join takes 834+834*278=232,686 I/Os. The filtering and output then take place in memory, incurring no extra I/O cost.

使用 block-nested loops join 的公式

Exercise 2. Index-Nested Loops Join

Reconsider the query in the above Exercise. But, now consider a different query plan, where the join is processed as an index-nested loops join (using the primary index on Rel2.C), and then each tuple of the join is filtered to check the value of D. How does this affect the cost?

Ex2:

· Assume index was not given 考试类似

· Firstly let's calculate the # of layers, and index

For Rel① $\frac{3846}{4+12+4} = \lfloor 192.3 \rfloor = 192 \times 0.75 = 144$ ↳ assumption

A B rowID

$\frac{834}{144} = \lceil 5.79 \rceil = 6$ # lowest level of Tree, we have 6 pages

6<144 所以只有2层

Tree 1

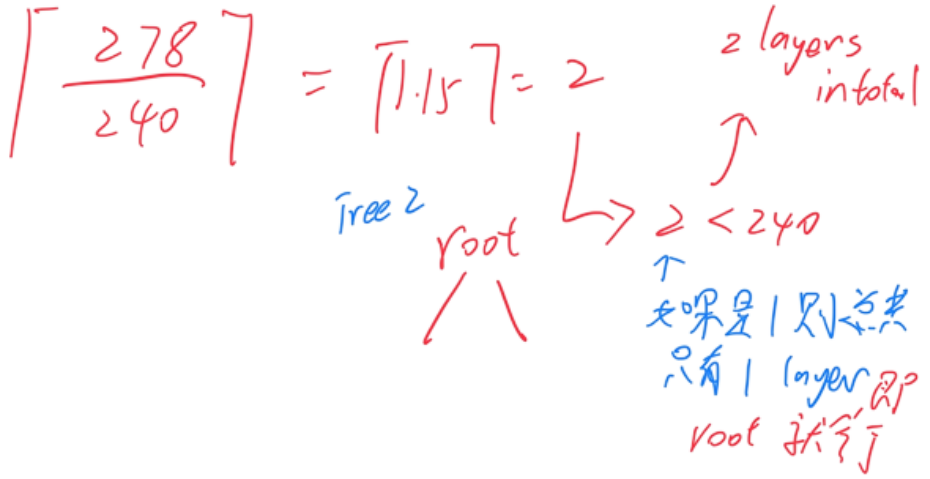
root

index 2

data

Rel② $\left\lfloor \frac{3846}{8+4} \times 0.75 \right\rfloor = 240$

only c as PK



· Now use B+ trees structure, we have

First scan whole rel1: 834, set unique C then

Second Using C: go through tree1
· go throug 3 pages (3 nodes) by using idx

· 100,000 × 3 = 300,000

Total 834 + 300000 =

注意这里的 Inner index 用的是 PK

不能用 Rel2 As outer relation because
因为 Rel2 不包含 A,B

External Merge-sort I/O Times

Calculate I/O for external merge-sort: N=108 pages, B = 5

- Number of passes: $\left\lceil \log_{B-1} \frac{N}{B} \right\rceil + 1 = \left\lceil \log_4 \frac{108}{5} \right\rceil + 1 = \lceil 2.21 \rceil + 1 = 4$
- Pass 0: create sorted run
  - Size of sorted runs: 5 = size of buffer
  - Number of sorted runs: $\left\lceil \frac{N}{B} \right\rceil = \left\lceil \frac{108}{5} \right\rceil = \lceil 21.6 \rceil = 22$ (last run is 3 pages)
- Pass 1: B-1 four-way merge to produce sorted runs of bigger size
  - Size of sorted run: 4 input slots * 5 pages in each sorted run = 20 pages
  - Number of sorted runs: $\left\lceil \frac{N}{B-1} \right\rceil = \left\lceil \frac{22}{4} \right\rceil = \lceil 5.5 \rceil = 6$ (last run is 8 pages)
- Pass 2: B-1 four-way merge to produce sorted runs of bigger size
  - Size of sorted run: 4 input slots * 20 pages in each sorted run = 80 pages
  - Number of sorted runs: $\left\lceil \frac{N}{B-1} \right\rceil = \left\lceil \frac{6}{4} \right\rceil = \lceil 1.5 \rceil = 2$ (last run is 28 pages)
- Pass 3: last merge to produce the sorted file
  - Number of sorted runs: $\left\lceil \frac{N}{B-1} \right\rceil = \left\lceil \frac{2}{4} \right\rceil = \lceil 0.5 \rceil = 1$
- Total costs: 2 (read & write) * number of passes * N (number of pages read and write in each pass) = 2 * 4 * 108 = 864 I/Os

Pass 0：把原始 108 页，按 5 页为单位划成小的 sorted run

- 每个 run 大小：5 页（因为 buffer 只能装 5 页）
- 所以有：

$$\lceil \frac{108}{5} \rceil = \lceil 21.6 \rceil = 22 \text{ 个 runs}$$

Pass 1：4-way 归并这些 runs（每次最多归并 4 个）

- 22 个 runs ⇒ 用 4 路归并，每次归并 4 个
- 有：

$$\lceil \frac{22}{4} \rceil = 6 \text{ 个新的 runs}$$

- 每个新 run 大小 $\approx 4 \times 5 = 20$ 页（最后一个较小）

📘 Pass 2：再次 4-way merge

- 6 个 runs ⇒ 每次归并 4 个
- 有：

$$\lceil \frac{6}{4} \rceil = 2 \text{ 个新的 runs}$$

- 每个新 run 大小 ≈ $4 \times 20 = 80$ 页（最后一个 28 页）

📘 Pass 3：再归并一次 ⇒ 得到最终的 sorted file

- 2 个 runs ⇒ merge 得到最终结果

📦 I/O 成本怎么算？

> 每一轮（每一 pass）你都要：
> ✅ 读一次全部数据
> ✅ 写一次全部数据

所以每一轮的 I/O 成本是：

$$2 \times N = 2 \times 108 = 216 \text{ I/Os}$$

总共 4 个 passes，所以总 I/O 是：

$$2 \times 108 \times 4 = \boxed{864 \text{ I/Os}}$$

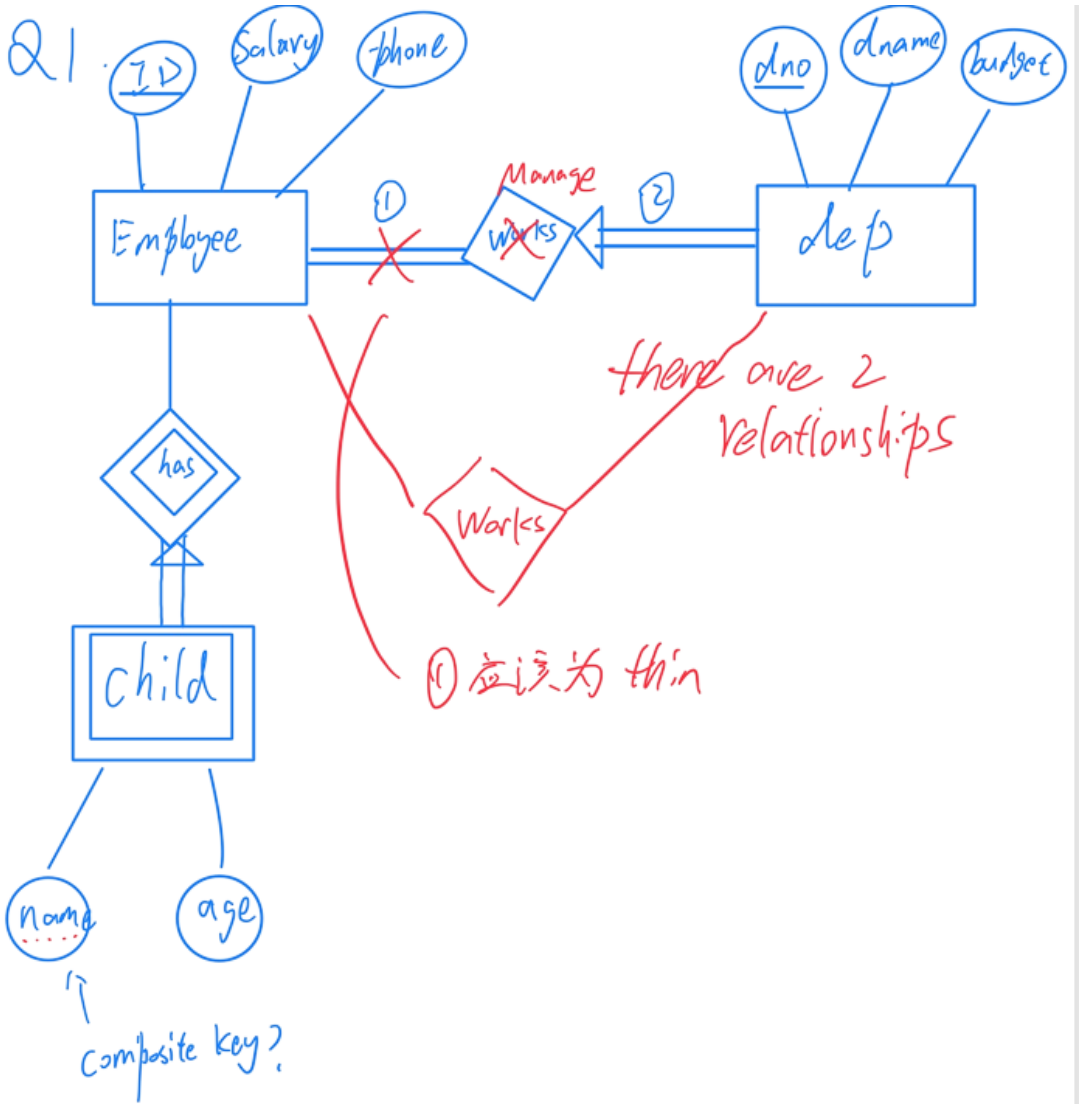
TUT13

Q1: Drawing ER diagram

A company database needs to store information about employees (identified by employeeID, with salary and phone as attributes), departments (identified by dno, with dname and budget as attributes), and children of employees (with name and age as attributes).

Employees work in departments; each department is managed by an employee; a child must be identified uniquely by name when the parent (who is an employee; assume that only one parent works for the company) is known. We are not interested in information about a child once the parent leaves the company.

Draw an ER diagram that captures this information.



Q2: Normalization

A only appear in left side, so it is a part of key

Suppose you are given a relation R with four attributes ABCD. For the given sets of FDs,

A → B, BC → D, A → C

do the following:

(a) Identify the candidate key(s) for R. A

(b) Identify the best normal form that R satisfies (1NF, 2NF, 3NF, or BCNF). 2NF

(c) If R is not in BCNF, decompose it into a set of BCNF relations that preserve the dependencies. (A, B) (A, C) (BC, D) left side is not a key

Q3: Unfinished Business

Attention in SQL syntax

- 英文
  - chronologically 按时间顺序
  - ordinal ranking: 每个条目都有独立编号，即使有相同值，也不会并列；ROW_NUMBER()
  - dense ranking：如果有相同值，会并列排名，下一个排名不会跳过；DENSE_RANK()
  - Standard ranking（或叫 sparse ranking）：有相同值时会并列，下一个排名会跳过；RANK()
- REFERENCE KEY MUST EITHER BE CANDIDATE KEY OR UNIQUE
- 不是每个表都必须有主键，但是绝大部分正规设计的表都会有主键。
- BETWEEN .. AND... 是 inclusive 的，而且可以对 String 使用
- Int / Int 返回的还是 INT，并且是直接截断小数部位，不会四舍五入
- LIKE 和 LOWER 连用可以解决忽略大小写的问题
- NULL 用 IS 判断
- 需要注意的是，别名需要用双引号 或者 不带引号，不能用单引号
- 用 YEAR 作为别名的时候要用到 AS
- 关于 SELECT 中别名的引用
  - WHERE 里面不能用
  - HAVING 里面不能用
  - GROUP BY 可以
- natural join 不用关心表的顺序，它会自动找出其中相匹配的 col 进行连接，并且会合并同名列
- The set operators require that all set relations have the /same schema/.
- INFINITY 和 -INFINITY 和 NaN 都是 FLOAT 类型，并且 INFINITY = INFINITY
- Format String
  - TO_CHAR(5, 'FM00') 会返回 05，即使数字是单一数字，FM 也会强制将它变成两位数字。
  - TO_CHAR(5, 'FM00x') 会返回 05x，因为 x 并不会被识别成填充符号。
  - TO_CHAR(-5, 'S00') 会返回 -05
- **只有除数或被除数是浮点数，结果才是浮点数**
  - 比如 5/3.0 得到 1.6666666666666667
  - 5/3 截断 1 后面的小数位
- 用 nested query 达成

Finding highest/lowest: especially where there are ties (otherwise LIMIT 1 would work)

```
WHERE <condition1> AND col1 = (SELECT MAX(col1) FROM table WHERE <condition1>)
```

Finding above average:

```
WHERE col1 > (SELECT AVG(col1) FROM table)
```

Check existence: each row in the outer query that doesn't satisfy inner query is removed from the result

```
WHERE EXISTS (SELECT 1 FROM t2 WHERE t2.col1 = t1.col1 AND t2.col2 = 'x')
```

-

SQL Syntax 集合

View

CREATE VIEW view_name AS … ;

CREATE VIEW student_enrollment AS SELECT sid, name, title, semester FROM student NATURAL JOIN Enrolled NATURAL JOIN unitofstudy;

Domain

CREATE DOMAIN GradeDomain AS CHAR(1) DEFAULT 'P' CHECK (VALUE IN ('F', 'P', 'C', 'D', 'H'));

CREATE TABLE name**(…);**

DROP TABLE name **CASCADE**

ALTER TABLE

- **添加列**: ALTER TABLE Flight ADD test1 INTEGER, ADD test2 INTEGER; 不能同时添加 constraint
- **删除列**: ALTER TABLE Flight DROP test1;
- **添加 Constraint**: 只能先删除旧的外键约束，再添加一个新的外键约束
  - ALTER TABLE Orders ADD FOREIGN KEY (customer_id) REFERENCES Customers(customer_id);
  - ALTER TABLE Orders ADD UNIQUE (column_name);
- 删除 Constraint: 除了 **NOT NULL** 和 **DEFAULT** 以外，别的都需要对应的 **Constraint 名字**来删除
  - ALTER TABLE table_name DROP CONSTRAINT constraint_name;
  - ALTER TABLE table_name ALTER COLUMN column_name DROP NOT NULL;
- **重命名列**: ALTER TABLE table_name RENAME COLUMN old_column_name TO new_column_name;
- **重命名表**: ALTER TABLE old_table_name RENAME TO new_table_name;
- 修改 Default Value 或者 data type
  - ALTER TABLE table_name ALTER COLUMN column_name SET DEFAULT 7.77;
  - ALTER TABLE table_name ALTER COLUMN column_name TYPE new_data_type;

INSERT INTO

```
INSERT INTO <table_name> [(<col1, col2…>)] VALUES
(attr1a, attr2a,…), [(attr1b, attr2b,…),…];
```

```
INSERT INTO Actor VALUES (4711, 'Arnold', 'Schwarzenegger', 'AT');
```

DELETE FROM

```
DELETE FROM Language WHERE name = 'German';
```

UPDATE

```
UPDATE Film
SET rental_rate = rental_rate *1.2
WHERE title = 'ANGELS LIFE';
```

ASSERTION

CREATE ASSERTION assertion-name CHECK (condition)

CREATE ASSERTION smallclub CHECK ((SELECT COUNT(*) FROM Sailors) + (SELECT COUNT(*) FROM Boats) < 10);

```
create or replace function maxcountsb() returns boolean as $maxc$
begin
if ((SELECT COUNT(sailors.sid) FROM Sailors)
        + (SELECT COUNT(boats.bid) FROM boats) < 9)
    then return true;
    else
    return false;
    end if;
end;
$maxc$ language plpgsql;
```

```
CREATE TABLE Boats (
            bid    INTEGER,
            PRIMARY KEY (bid),
            color  CHAR(10)--,
            --CHECK (maxcountsb())
            );
```

Trigger

语句级触发器

```SQL
CREATE FUNCTION function_name()
RETURNS trigger AS $$
BEGIN
  -- 触发时执行的操作
  RETURN NULL; -- 对于 AFTER 触发器可以返回 NULL,
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_name
AFTER INSERT OR UPDATE OR DELETE ON table_name
FOR EACH STATEMENT
EXECUTE FUNCTION function_name();
```

行级触发器

```SQL
CREATE FUNCTION function_name()
RETURNS trigger AS $$
BEGIN
  -- 可以访问 NEW.column_name 或 OLD.column_name
  RETURN NEW;  -- BEFORE 触发器必须返回 NEW
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_name
BEFORE INSERT OR UPDATE OR DELETE ON table_name
FOR EACH ROW
EXECUTE FUNCTION function_name();
```

```
-- 定义一个触发函数：把对视图的 INSERT 转换为对 employee 表的 INSERT
CREATE FUNCTION insert_employee_into_view()
RETURNS trigger AS $$
BEGIN
  -- 把插入视图的行为改为插入真实表
  INSERT INTO employee(name, dept_id)
  VALUES (NEW.name, (SELECT id FROM department WHERE dept_name = NE
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;

-- 定义 INSTEAD OF 触发器
CREATE TRIGGER employee_view_insert
INSTEAD OF INSERT ON employee_view
FOR EACH ROW
EXECUTE FUNCTION insert_employee_into_view();
```

Procedure

```
CREATE OR REPLACE FUNCTION check_login(login VARCHAR(10), password_param VARCHAR(20))
RETURNS TABLE(username VARCHAR(10), firstname VARCHAR(50), lastname VARCHAR(50)) AS $$
BEGIN
    RETURN QUERY
    SELECT s.username, s.firstname, s.lastname
    FROM salesperson s
    WHERE LOWER(s.username) = LOWER(login) AND s.password = password_param;
END;
$$ LANGUAGE plpgsql;
```

Set Operations

Union: (query1) UNION (query2)
Intersection: (query1) INTERSECT (query2)
Difference: (query1) EXCEPT (query2)

```
SELECT name FROM tableA
UNION
SELECT name FROM tableB;
```

Correlated Subquery

[NOT] EXISTS

```sql
-- 查询所有选修任何一门课程的学生
SELECT name
FROM Student s
WHERE EXISTS (
    SELECT 1
    FROM Enrolled e
    WHERE e.student_id = s.id
);
```

ALL

```sql
-- 查询比所有 "COMP5138" 选课学生的 GPA 都高的学生
SELECT name
FROM Student
WHERE gpa > ALL (
    SELECT s2.gpa
    FROM Student s2
    JOIN Enrolled e2 ON s2.id = e2.student_id
    WHERE e2.uos_code = 'COMP5138'
);
```

SOME

```sql
-- 查询 GPA 高于部分（至少一个） "COMP5138" 学生的学生
SELECT name
FROM Student
WHERE gpa > SOME (
    SELECT s2.gpa
    FROM Student s2
    JOIN Enrolled e2 ON s2.id = e2.student_id
    WHERE e2.uos_code = 'COMP5138'
);
```

Transaction Syntax

```
BEGIN;
    SQL;
COMMIT;
ROLLBACK/ABORT;
```

4 Isolation Level

```
SET TRANSACTION ISOLATION LEVEL
{ SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }


-- Example
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- 这里开始写查询或更新语句
SELECT * FROM my_table;

COMMIT;
```

Delay

- NOT DEFERRABLE (默认的)，表明检测方式不能修改
- DEFERRABLE INITIALLY IMMEDIATE，立刻检测
  - FOREIGN KEY (lecturer_id) REFERENCES Lecturer(lecturer_id) DEFERRABLE INITIALLY IMMEDIATE
- DEFERRABLE INITIALLY DEFERRED，等 transaction 结束再检查
- 修改上面 2 种的语法
  - SET CONSTRAINTS name IMMEDIATE
  - SET CONSTRAINTS name DEFERRED

FOREIGN KEY

```
FOREIGN KEY (department_id) REFERENCES departments(department_id),
```

SET DEFAULT，如果对应的 col 没有 default constraint 会报错
ON DELETE/UPDATE CASCADE / SET NULL / SET DEFAULT
两个可以同时存在，比如 ON DELETE .... ON UPDATE ....

PRIMARY KEY

```
PRIMARY KEY (employee_id),
```

LIKE

`LIKE 'str%'` (match words starting with str)
`LIKE '%str%'` (match words containing str)
`LIKE 'str_'` (match a single character)
`LIKE 'str___'` (matches three characters after str)

use `LOWER(col) LIKE 'str'` for case-insensitive matching

JOIN

- Implicit join is Cartesian join/CROSS join
  - 类似于 FROM A, B
  - 如果 A 或 B 为空，则 CROSS 也为空
- Inner join，特殊写法 using 替代 on；省略 Inner
  - Theta Join（θ-Join）
  - Equi-Join 等值连接
  - Natural Join
    - Natural Join Caveat

Natural Join Caveat：如果两个表中没有相同的列名，则 `NATURAL JOIN` 会返回笛卡尔积（不推荐）。

- $R \bowtie S = \pi_{unique\_attributes}(\sigma_{equality\_of\_common\_attributes}(R \times S))$

- Outer join
  - LEFF JOIN
    - NATURAL LEFT OUTER JOIN: 不需要 ON 语句，它会自动匹配两个表中相同名称的列，并以左表 (Employee) 为主表. 结果可能和 LEFT JOIN 不同，因为 NATURAL JOIN 可能匹配多个同名列，而 LEFT JOIN 只会匹配 ON 指定的列。
  - RIGHT JOIN
  - FULL JOIN: 不会返回重复的行。在右表没有孤儿行的情况下等同于 LEFT JOIN

ON: keep duplicated columns
USING: remove duplicate columns

Data and time

DATE	'2024-11-05'
TIME	'05:20:35.331369+00'
TIMESTAMP	'2024-11-05 05:20:35.331369+00'
INTERVAL	'3 hours 15 minutes'

- DATE – DATE 得到的是相差的天数，为 INT
- DATE – INTERVAL 得到的还是 DATE 类型

Constants: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`

Extract individual time fields: `EXTRACT(<component> FROM <value>)`

- Date-related component (DATE and TIMESTAMP): `DAY`, `MONTH`, `YEAR`
- Time-related component (TIME and TIMESTAMP): `HOUR`, `MINUTE`, `SECOND`, `TIMEZONE`
- Day of the year: `DOY`, Day of the week (Mon): `ISODOW`, Week of the year: `WEEK`

```
SELECT EXTRACT(DOY FROM DATE '2025-06-12');  -- 返回 163
```

Make date using year, month and day: `MAKE_DATE(year, month, date)`

```
SELECT MAKE_DATE(2025, 6, 12);
```

Convert from string to date: `TO_DATE(<string>, <format of string>)`

```
SELECT TO_DATE('2025-06-12', 'YYYY-MM-DD');
```

```
SELECT * FROM Orders
WHERE order_date >= DATE '2015-01-01';
```

COALESCE()

```
SELECT film_id, title, release_year, COALESCE(rating, 'TBD') rating
FROM Film
ORDER BY release_year DESC, title;
```

如果是 null 就返回 TBD

NULLIF()

```
-- 注意是NULLIF 不是 IFNULL，并且他是把输出转换成NULL的

SELECT film_id, title, release_year, NULLIF(rating, 'TBD') rating
FROM Film
ORDER BY release_year DESC, title ASC;
```

如果是 TBD 就返回 NULL

CASE WHEN ··· THEN ··· ELSE ··· END

```
SELECT film_id, title, release_year, rating,
    CASE
        WHEN rating = 'G' THEN 'General Audiences. All Ages Admitted.'
        WHEN rating = 'PG' THEN 'Parental Guidance Suggested. Some Material May Not Be Suitable For Children.'
        WHEN rating = 'PG-13' THEN 'Parents Strongly Cautioned. Some Material May Be Inappropriate For Children U
        WHEN rating = 'R' THEN 'Restricted. Children Under 17 Require Accompanying Parent or Adult Guardian.'
        WHEN rating = 'NC-17' THEN 'No One 17 and Under Admitted.'
    END rating_def
FROM Film
WHERE release_year BETWEEN 2002 AND 2006
```

Aggregation

- COUNT, SUM, AVG, MAX, MIN, e.g. COUNT(col)

PAST EXAM

Insert Anomaly

想插入一条合法的信息，但由于设计不当，必须插入其他不相关的信息才能成功。

特点就是有些 FD 必须要符合

$$\text{ProvNo} \rightarrow \text{ProvSpeciality}$$

Insert anomaly: We cannot insert a new ProvNo to describe a doctor's speciality unless we also include a VisitID (i.e., we can't record details about a doctor unless they've made a visit).

Delete Anomaly

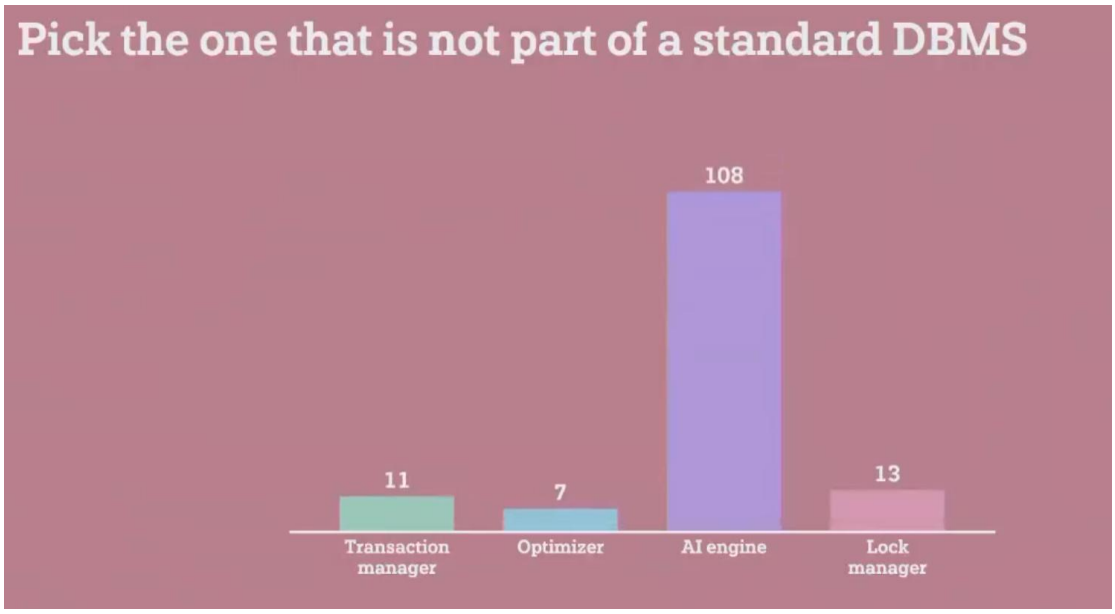
删除一条信息时，意外地丢失了其他有用的信息。

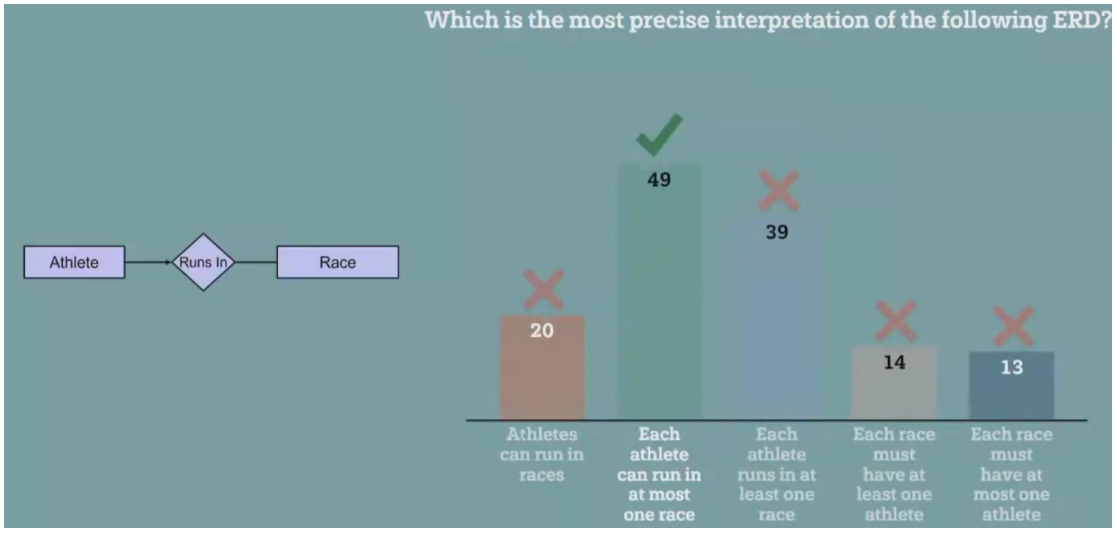
Update Anomaly

修改某一项信息时，由于冗余存在多个副本，必须更新多个地方，否则数据不一致。

Menti

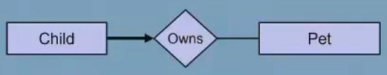
Week1


Pick the one that is not part of a standard DBMS
11
7
108
13
Transaction manager
Optimizer
AI engine
Lock manager

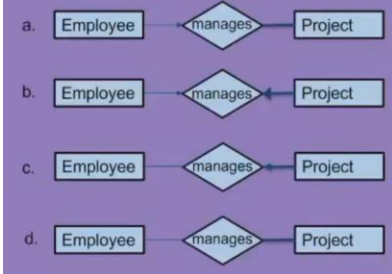

Which is the most precise interpretation of the following ERD?
Athlete
Runs In
Race
20
49
39
14
13
Athletes can run in races
Each athlete can run in at most one race
Each athlete runs in at least one race
Each race must have at least one athlete
Each race must have at most one athlete


Which is the most precise interpretation of the following ERD?
Person
Likes
Colour
21
53
9
47
Each person must like exactly one colour
Each colour must be liked by at least one person
Each person must like a different colour
Each person must like at least one colour

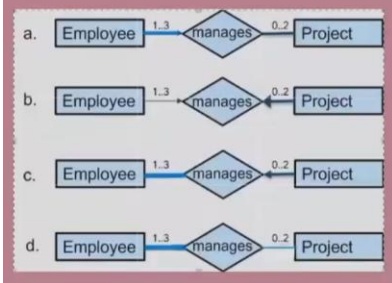
Which is the most precise interpretation of the following ERD?

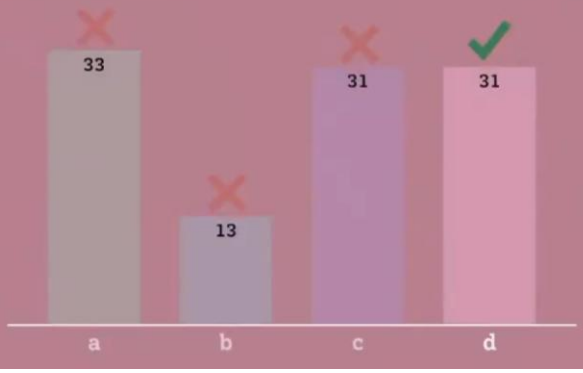


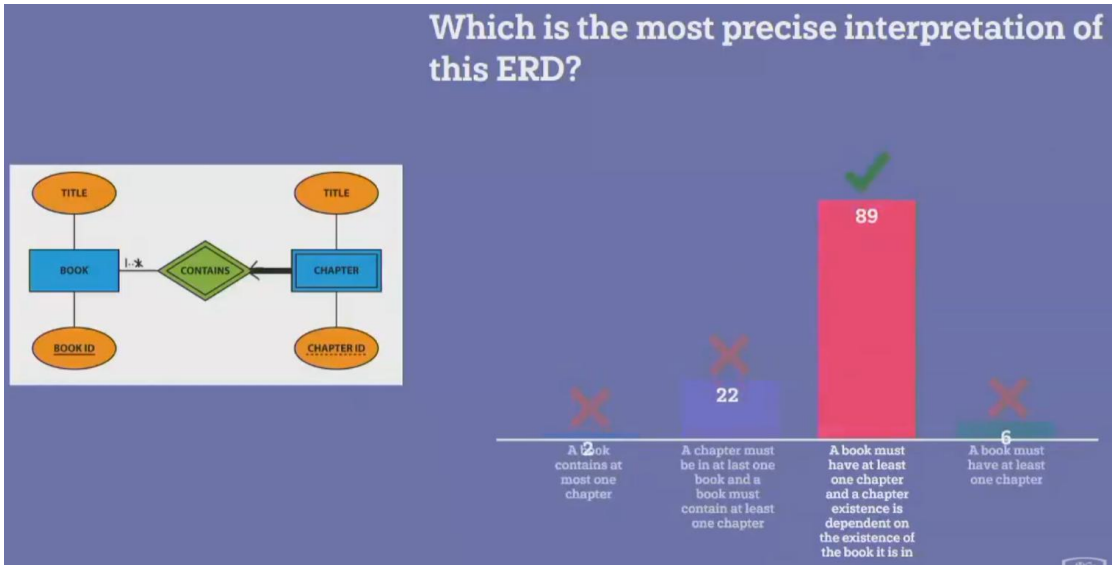
Which is the correct model for "One employee manages at most one project. Every project must be managed by at least one employee"?



Which is the correct model for "Each employee manages between 1 and 3 projects. Each project has either 0, 1, or 2 managers "?

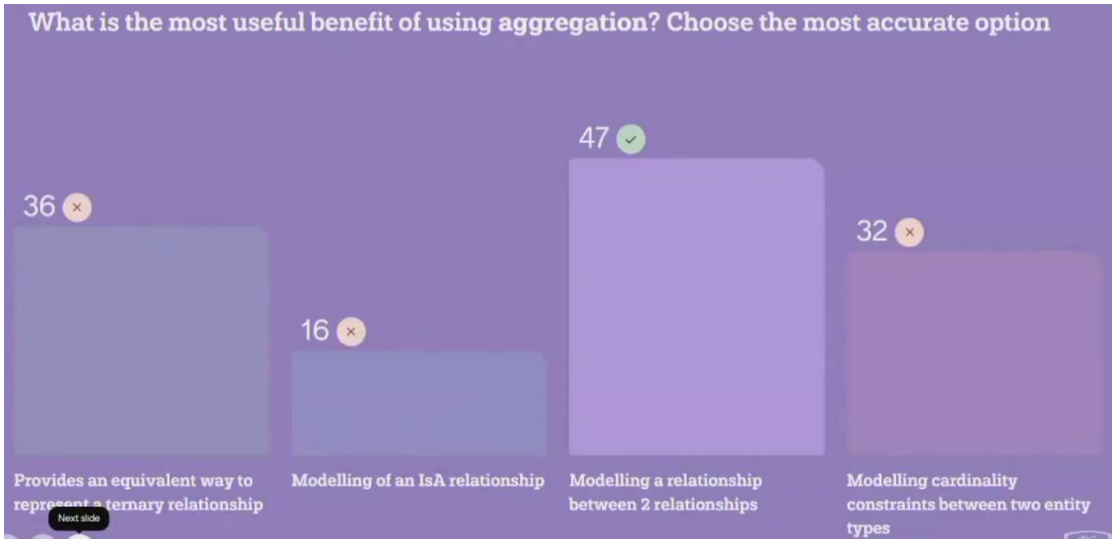


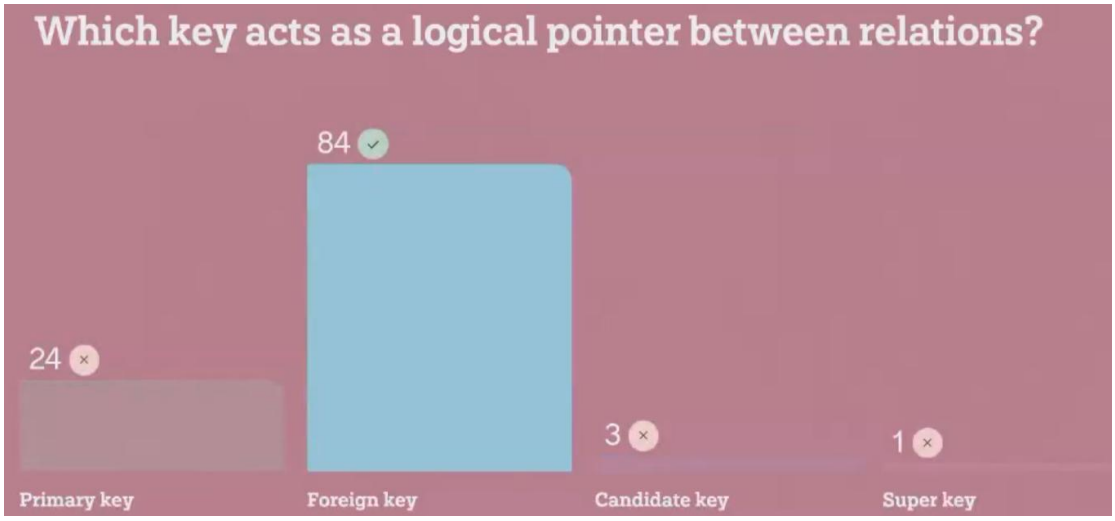


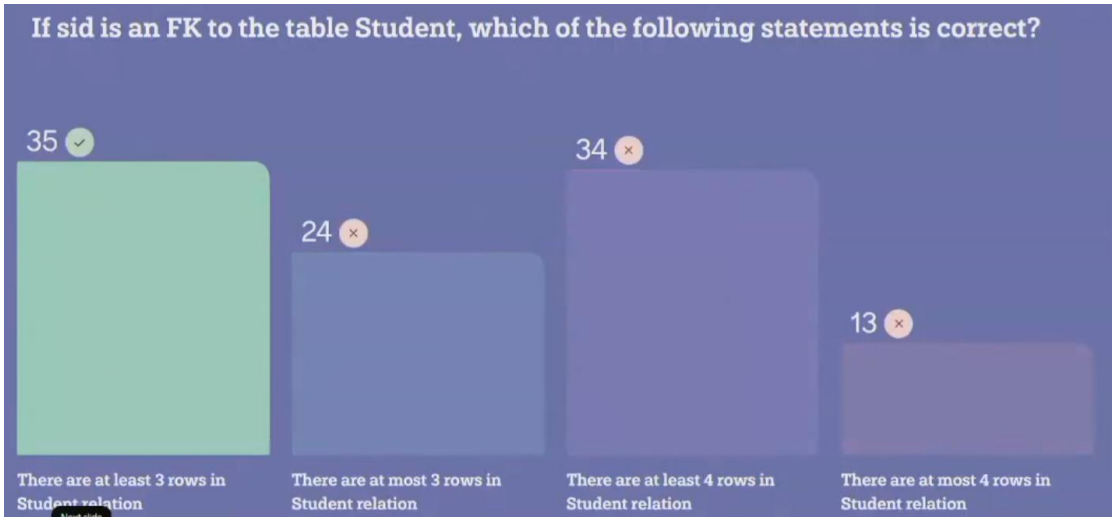

Which is the most precise interpretation of this ERD?
TITLE
BOOK
CONTAINS
CHAPTER
TITLE
BOOK ID
CHAPTER ID
2
22
89
6
A book contains at most one chapter
A chapter must be in at last one book and a book must contain at least one chapter
A book must have at least one chapter and a chapter existence is dependent on the existence of the book it is in
A book must have at least one chapter

Week 2

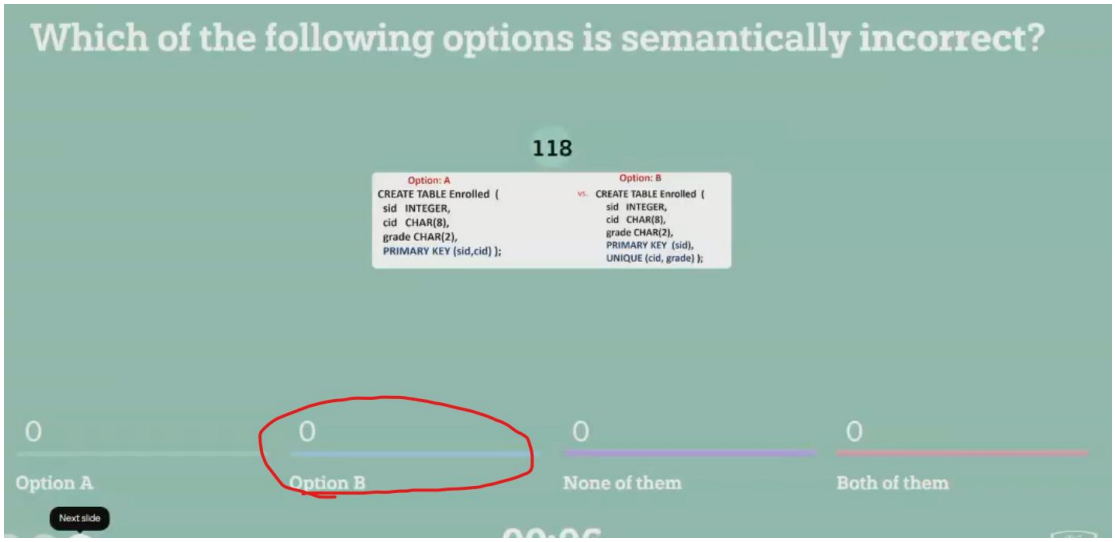

Choose the most accurate definition of a total disjoint Is-A relationship
42
33
36
35
An entity of the superclass can be in one or more subclasses
An entity of a subclass may or may not be in the superclass
Each entity of the superclass must be in one or more subclasses
Each entity of the superclass should be in exactly one subclass


What is the most useful benefit of using aggregation? Choose the most accurate option
36
16
47
32
Provides an equivalent way to represent a ternary relationship
Modelling of an IsA relationship
Modelling a relationship between 2 relationships
Modelling cardinality constraints between two entity types
Next slide

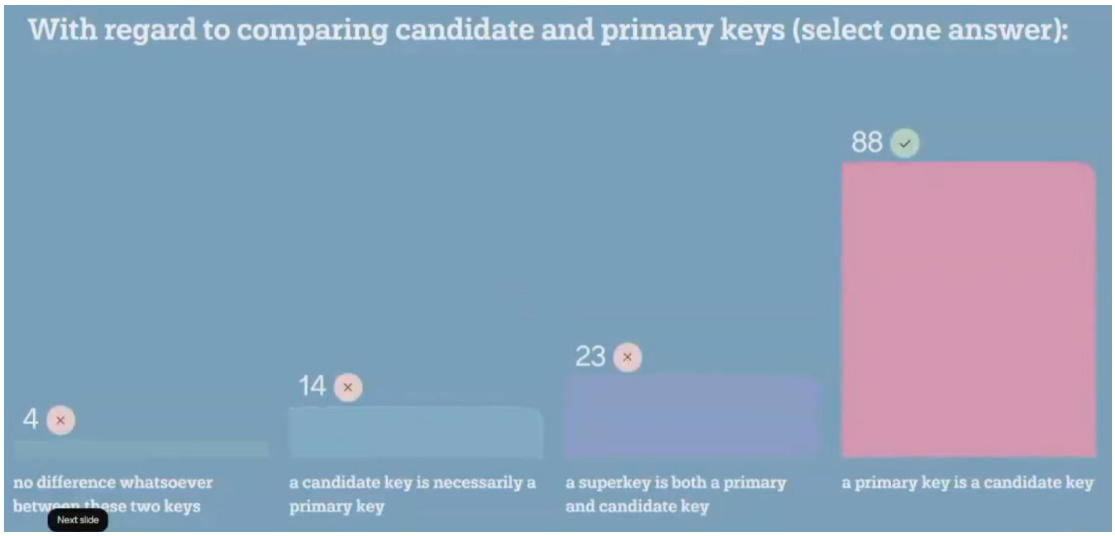




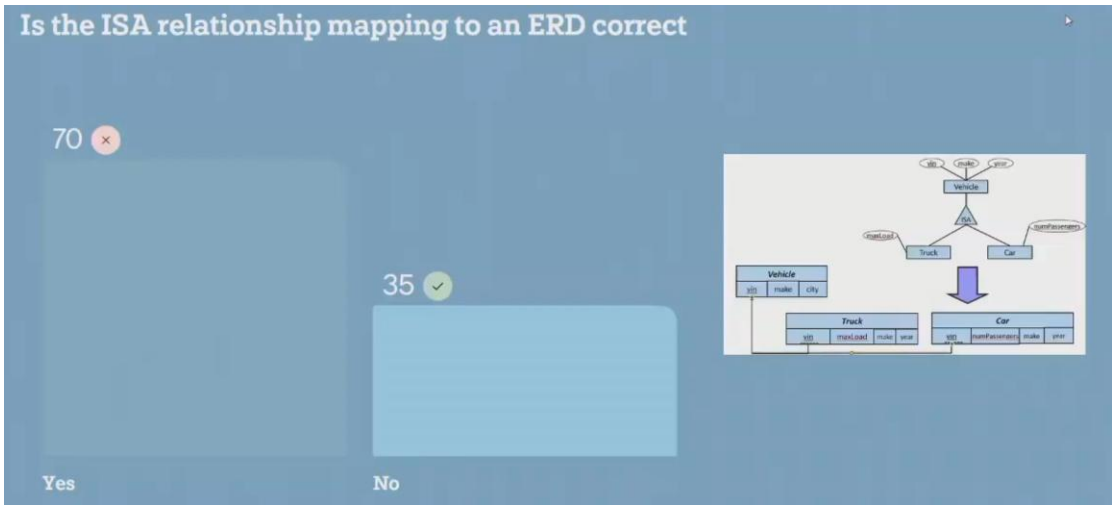
下面这个比较重要



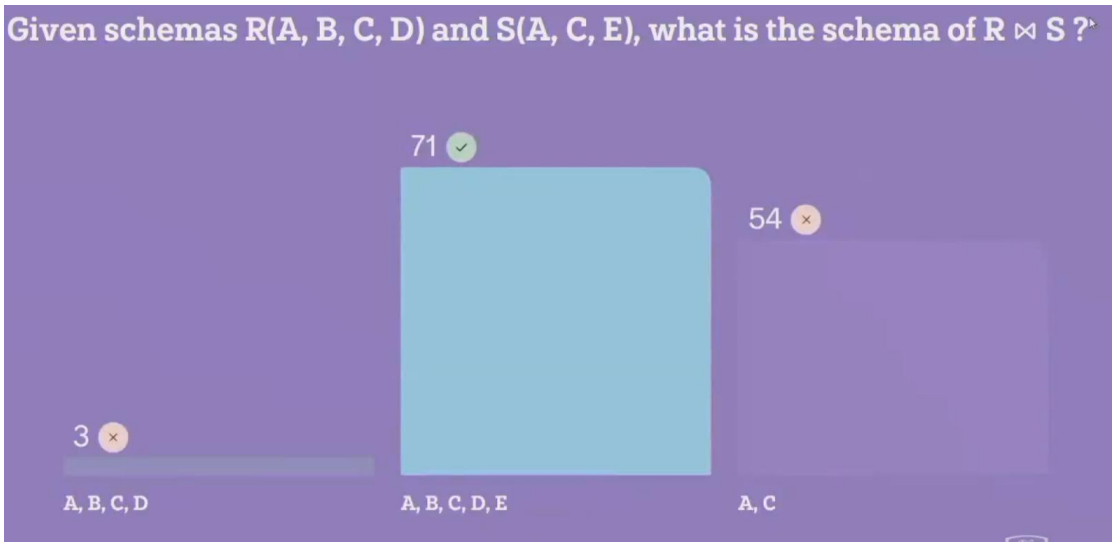
不对是因为 enroll 表必须用 studentid 和 courseid 作为主键才符合 semantical def

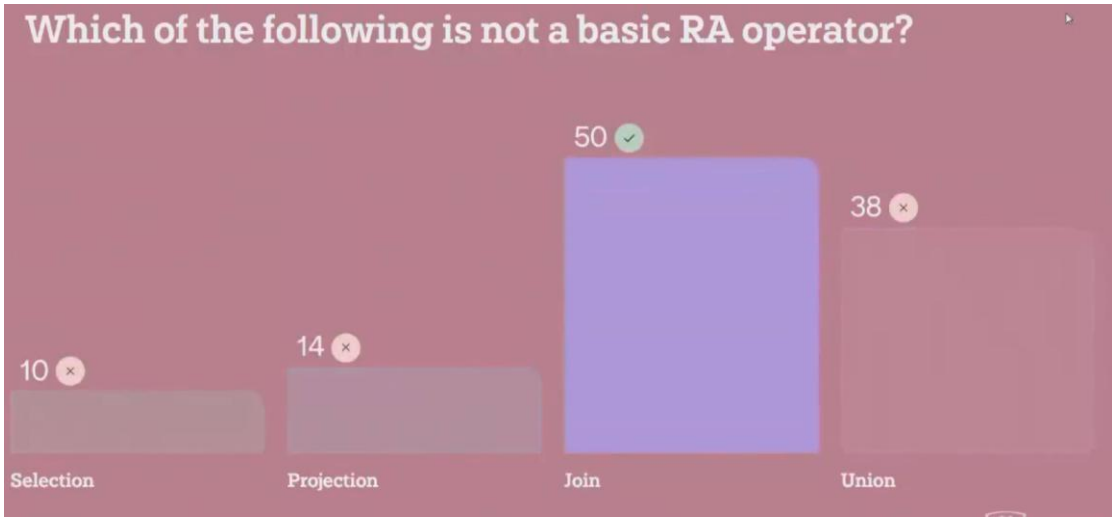


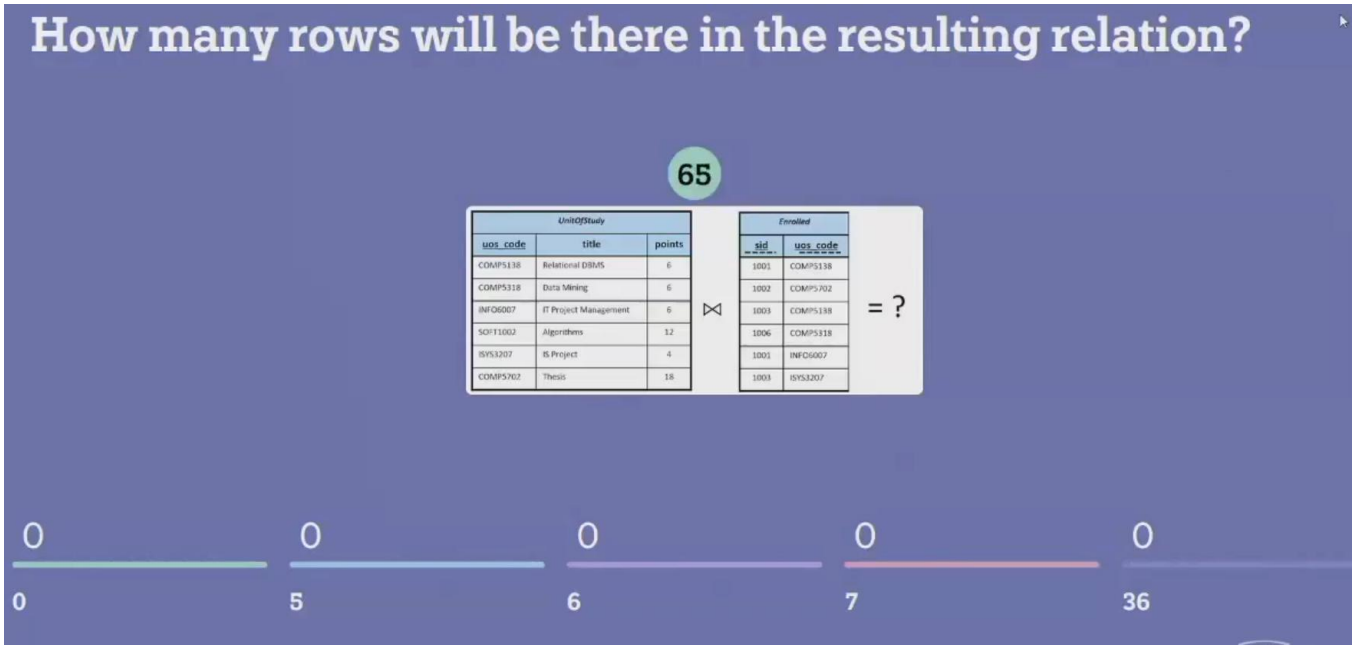
Week3



这里因为 child 包含了 parent 的 attributes 所以不对







这里我们用 uos_code 作为 natural join。Natural join 返回的是个 set，不是 table，因此不需要有 PK

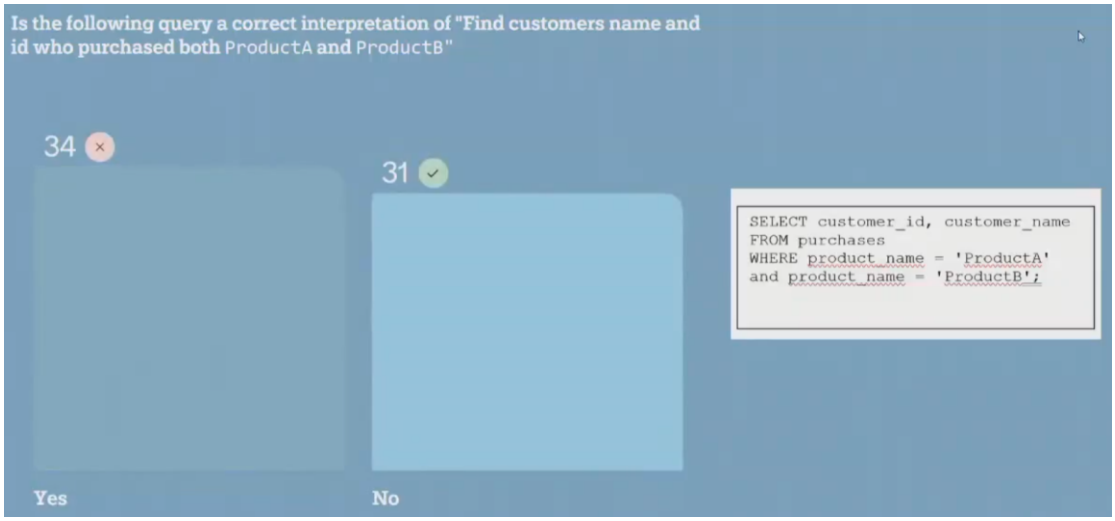
左边的 soft1002 没有对应的所以不显示

Comp5138 对应的有 2 个，所以显示两个

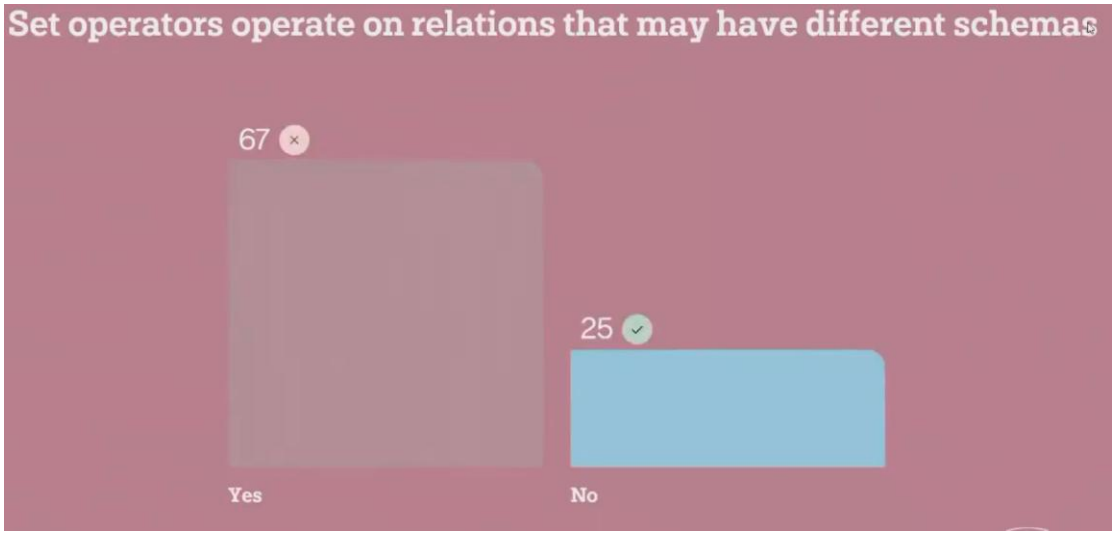
别的都是一一对应

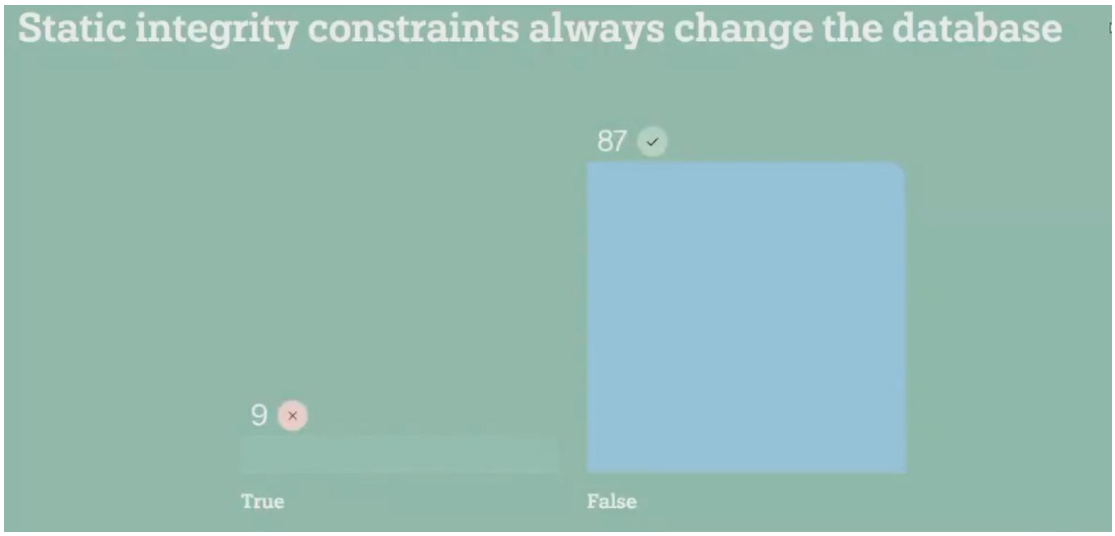
总共应该显示 6 行

Week5

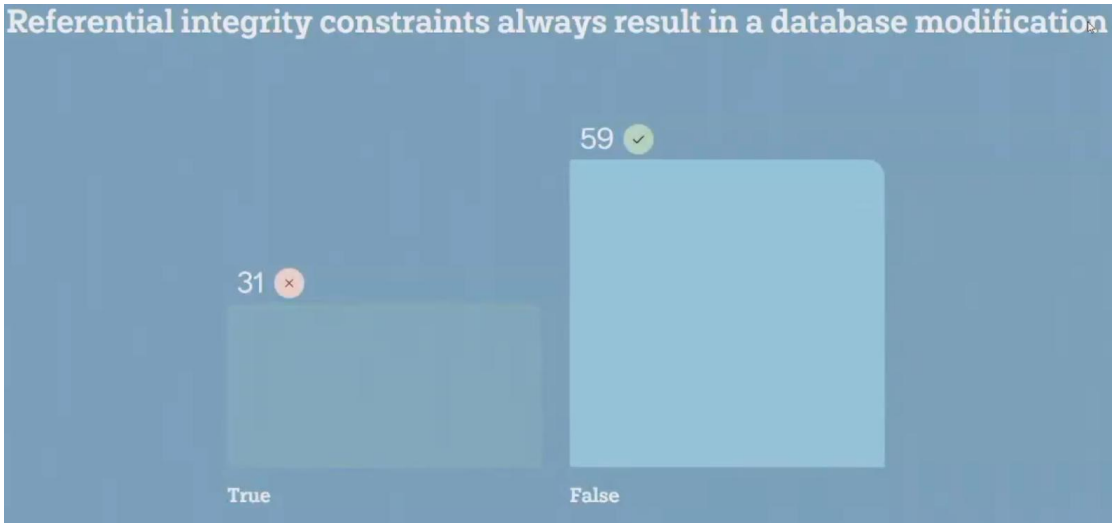


上面的思路是用 intersect 或者 subquery，参考 TUT6



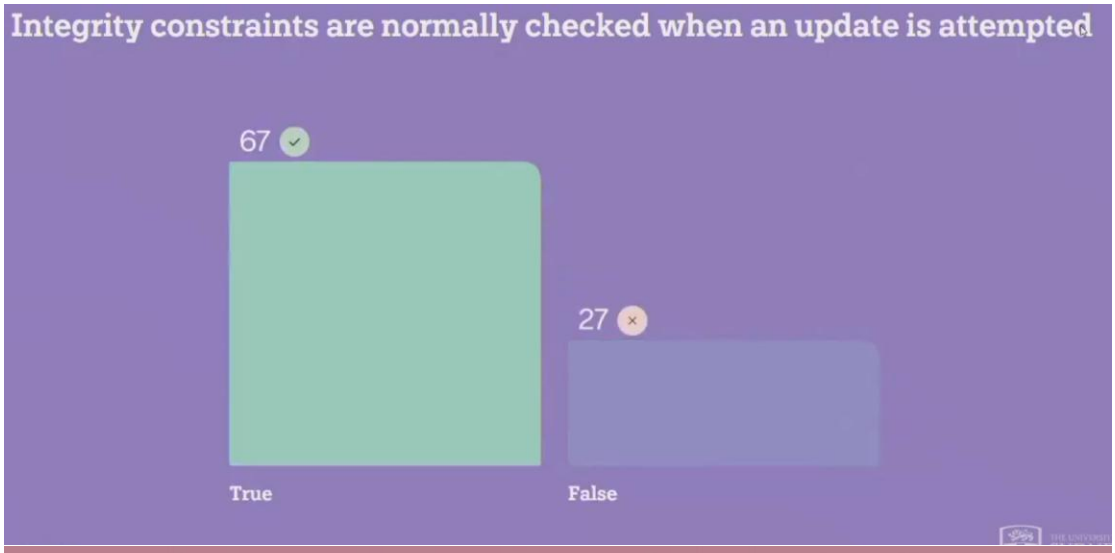


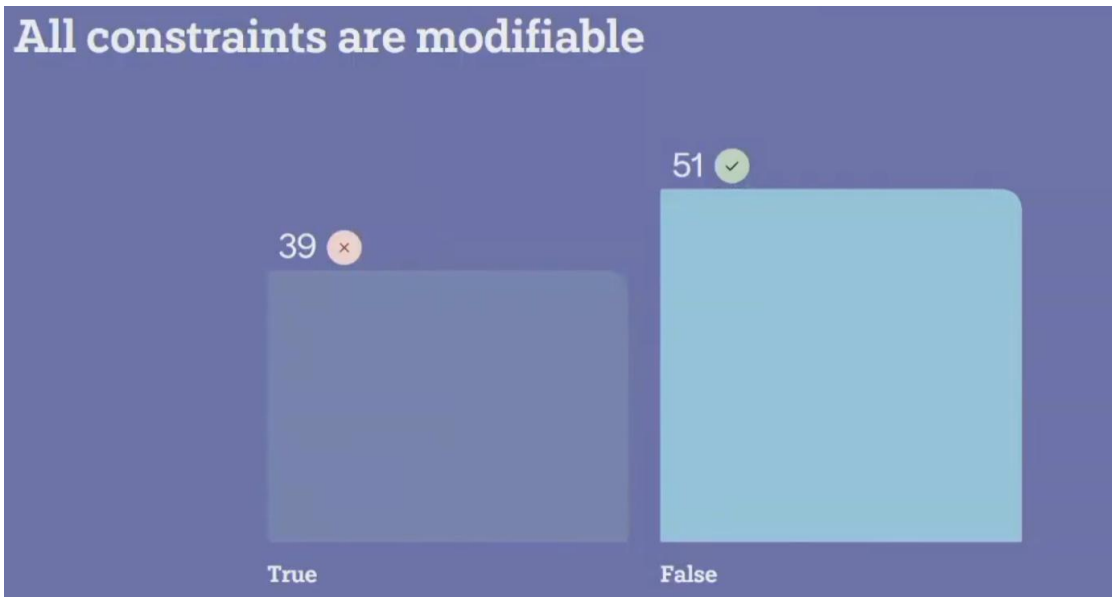
即使是 dynamic 也不应该 always，而只有在 trigger 中包含 update 的才会改变 DB



Referential Integrity：

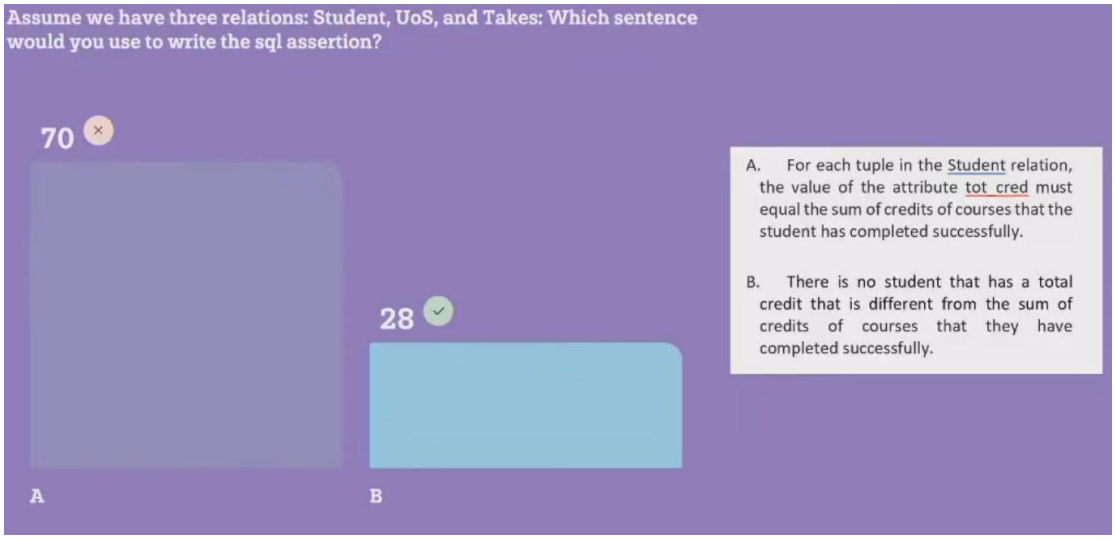
只是表达 references，对于 CASCADE, SET NULL, SET DEFAULT 是额外选项，因此不是 always





Only Domain constraints are modifiable

Week6



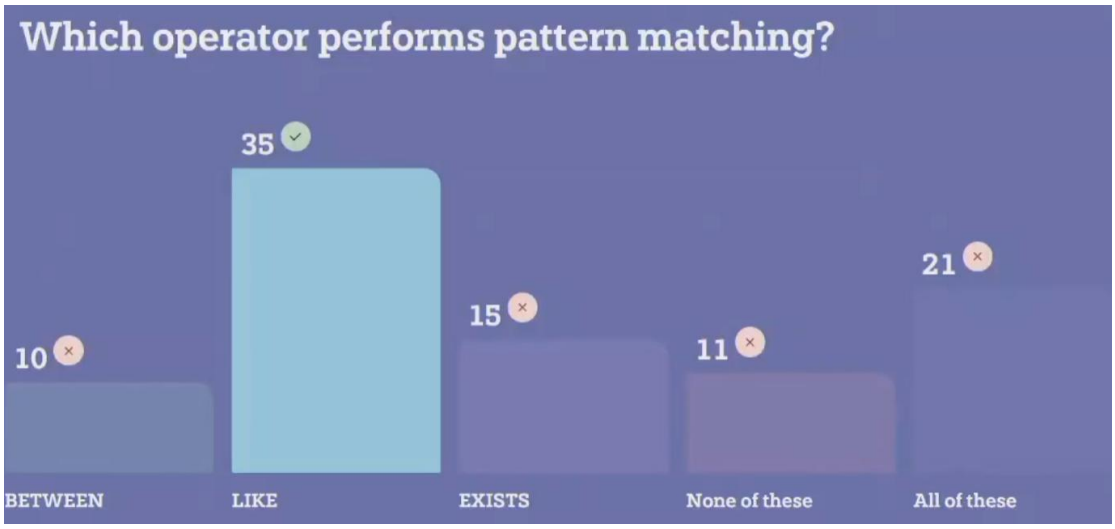
A 句 是针对**每一个单独元组（学生）的属性值和其他数据之间的等价关系。它表达的是元组级别的条件**（局部约束）。

→ 这类约束更适合用**触发器（Trigger）**或通过程序逻辑来实现，因为它关注的是单条记录的内部一致性。

B 句 是针对**整个数据库**的一个普遍命题，说明**不存在**一个学生的总学分与其完成课程学分和不相等。

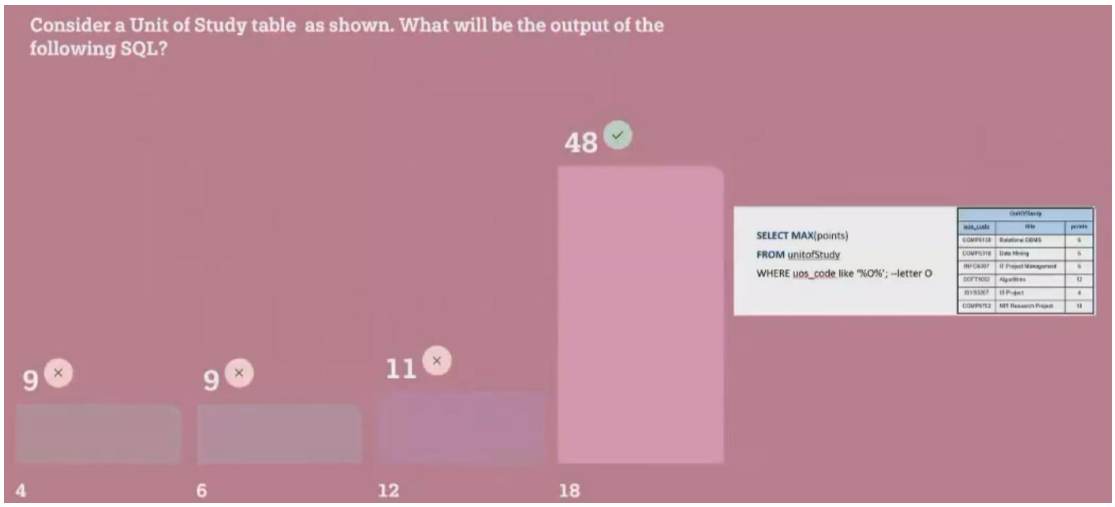
→ 这是典型的**全局约束**，适合用 **Assertion** 来表达，因为它是针对数据库整体状态的。


Row and Statement Triggers are executed exactly the same way.
82
9
True
False


Which operator performs pattern matching?
35
10
15
11
21
BETWEEN
LIKE
EXISTS
None of these
All of these

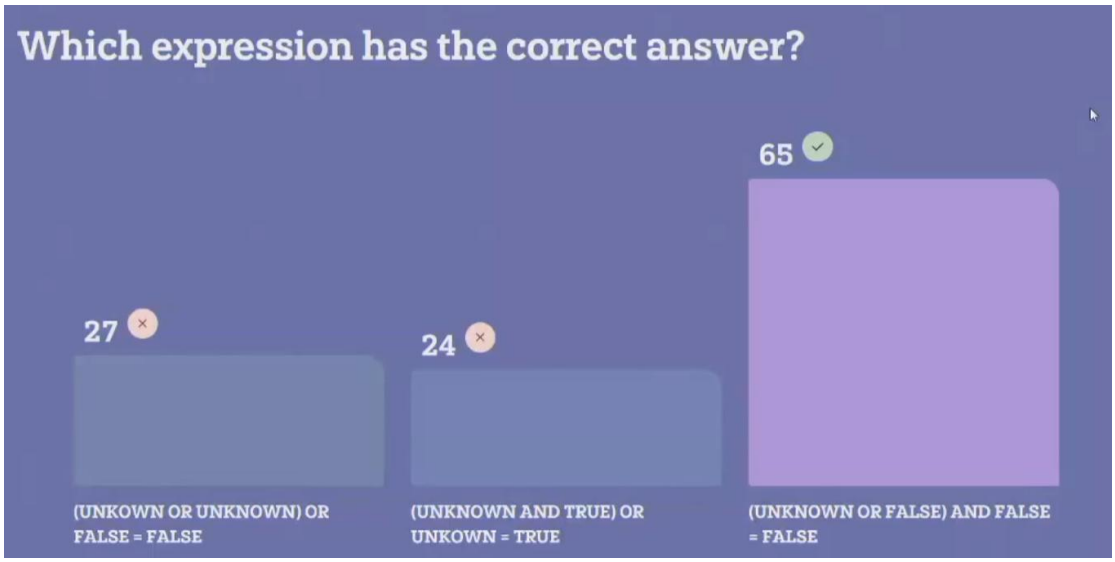

Given a USERS table, which of the following SQL syntax is correct?
42
27
9
7
SELECT usename and password FROM users
SELECT usename, password WHERE username = 'user1'
SELECT usename, password FROM users
SELECT usename, password FROM table = users

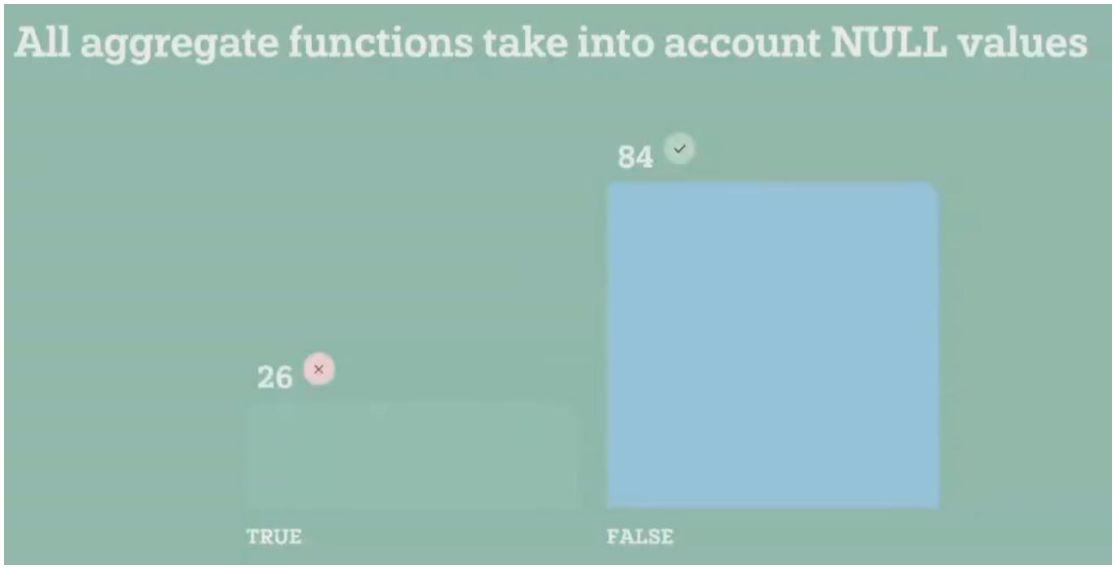

Consider an Employee table that has 10 records. It has a non-null SALARY column which is UNIQUE. What will be the output of the following SQL?
38
19
15
20
1
SELECT COUNT(*)
FROM Employee
WHERE salary > ANY (SELECT salary
FROM Employee)
10
9
5
0
1


Consider a Unit of Study table as shown. What will be the output of the following SQL?
48
9
9
11
SELECT MAX(points)
FROM unitofStudy
WHERE uos_code like '%0%'; --letter O
4
6
12
18

Week7

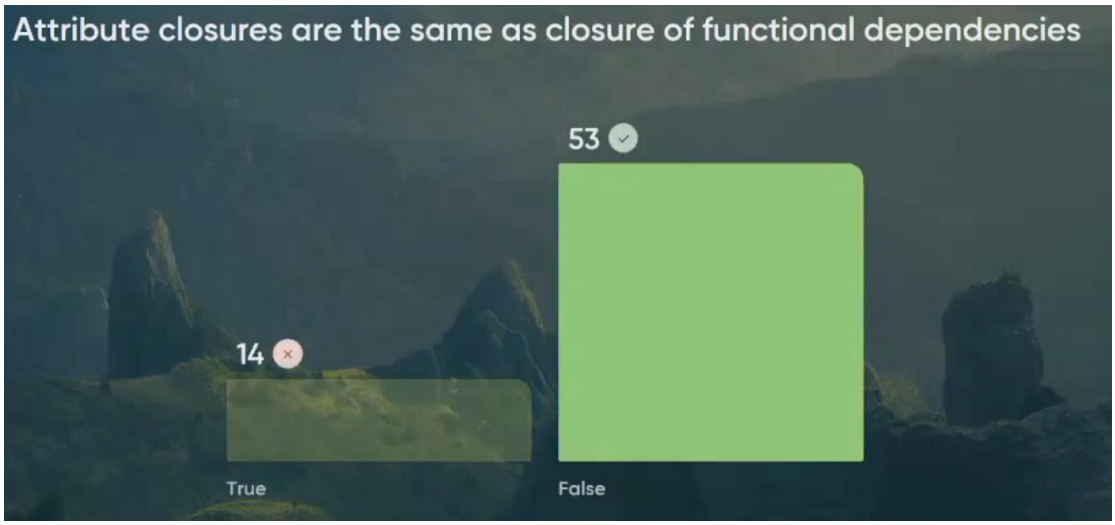

Which comparison results are correct?
47
38
31
(0 <= NULL) returns Unknown
(NULL = NULL) returns True
(NULL <= NULL) returns Unknown

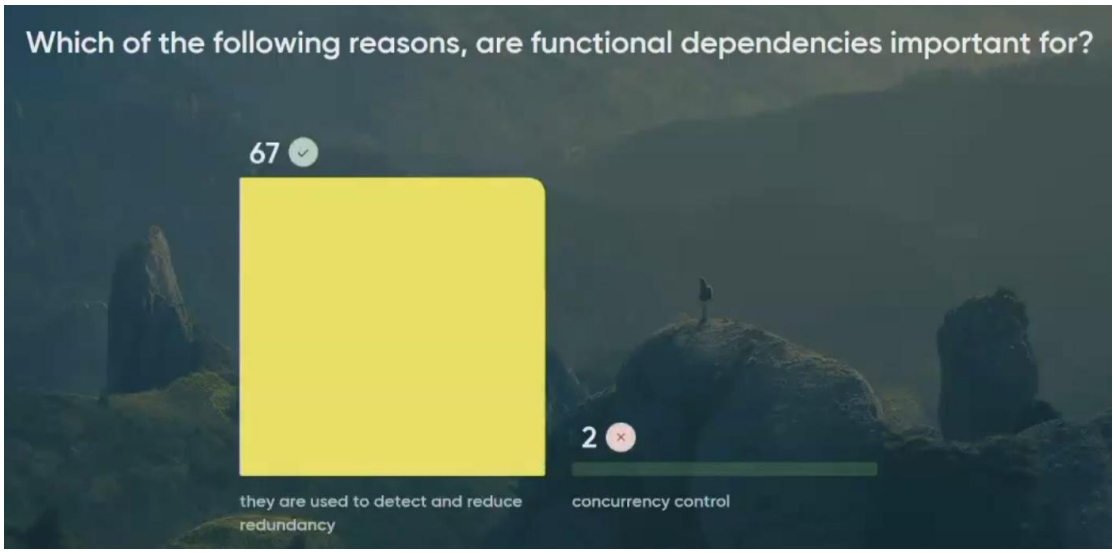

Which expression has the correct answer?
65
27
24
(UNKOWN OR UNKNOWN) OR FALSE = FALSE
(UNKNOWN AND TRUE) OR UNKOWN = TRUE
(UNKNOWN OR FALSE) AND FALSE = FALSE


All aggregate functions take into account NULL values
84
26
TRUE
FALSE

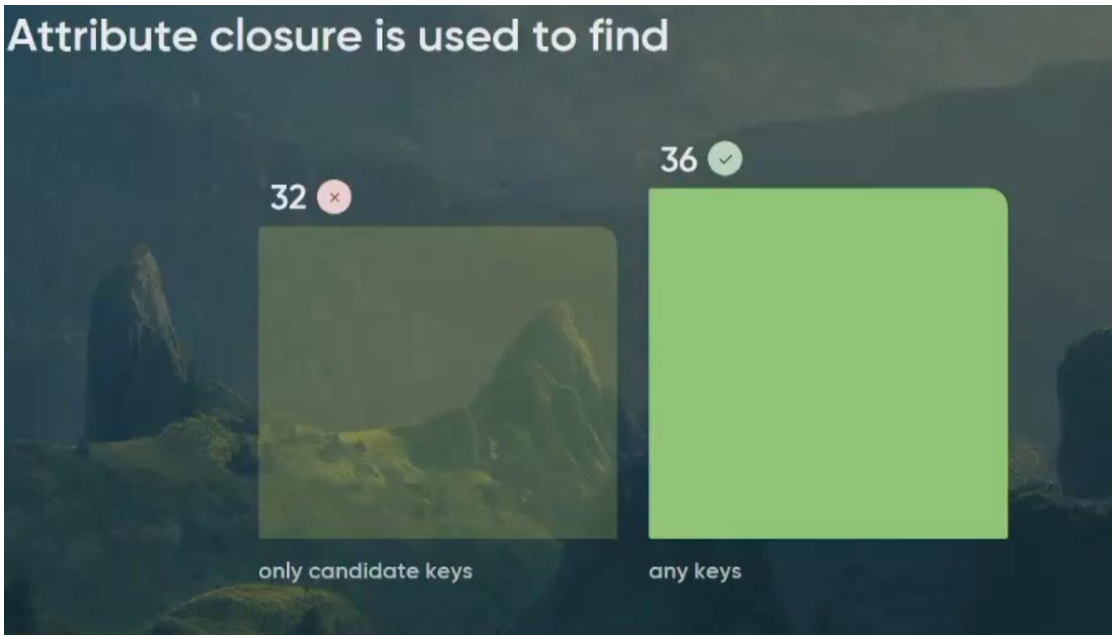
Week7


Functional dependencies
59
8
define an N-M relationship between two sets of attributes
define an N-1 relationship between two sets of attributes





Isolation level 正是数据库中用于确保 **并发控制（Concurrency Control）** 的关键机制。



Week9


1NF is NOT the main reason for the need of 4NF
43
51
True
False

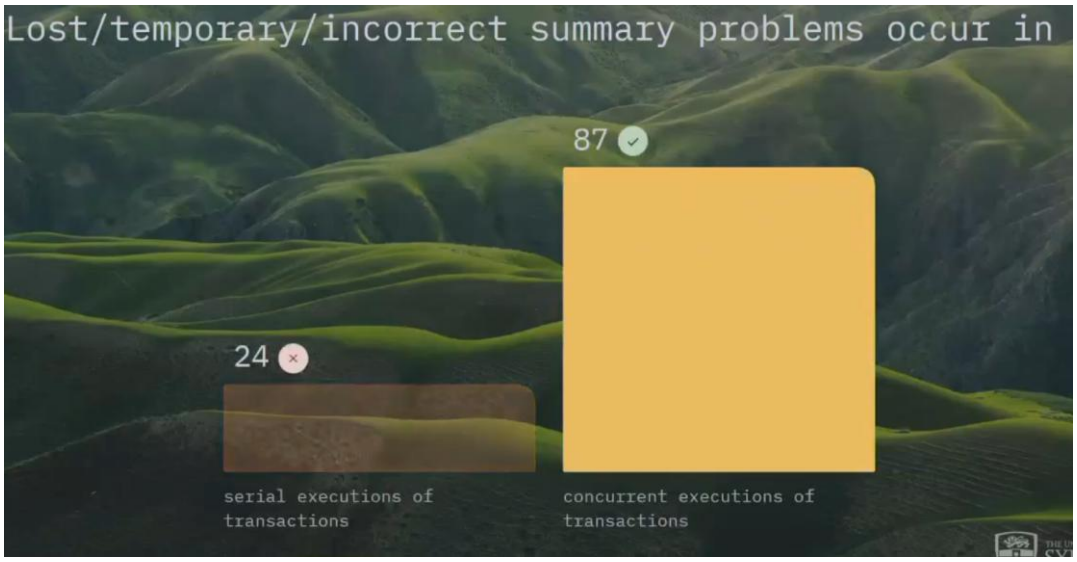

Multivalued dependencies introduce duplication
77
32
True
False

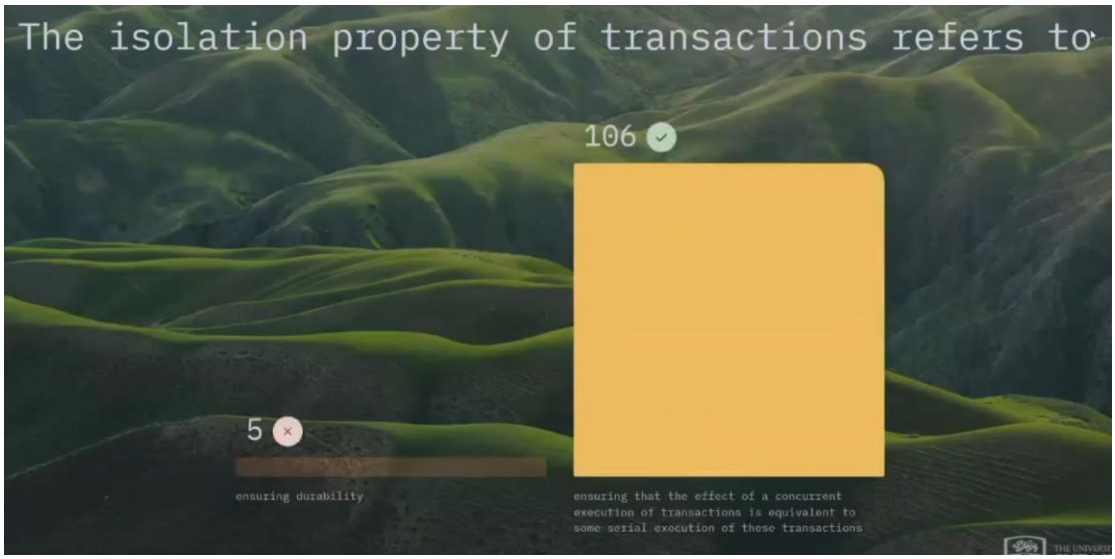

For a decomposition to be lossless-join
78
31
it needs to preserve the functional dependencies
joining the resulting decomposed relations would yield the original relation


Multivalued dependencies are needed to
67
to represent a relationship between any two set-valued attributes
41
establish independence between two relationships


Transaction consistency is important because
108
it defines the rules, including the integrity constraints, that need to be adhered to for the database to be in a correct state
3
the database design depends on it

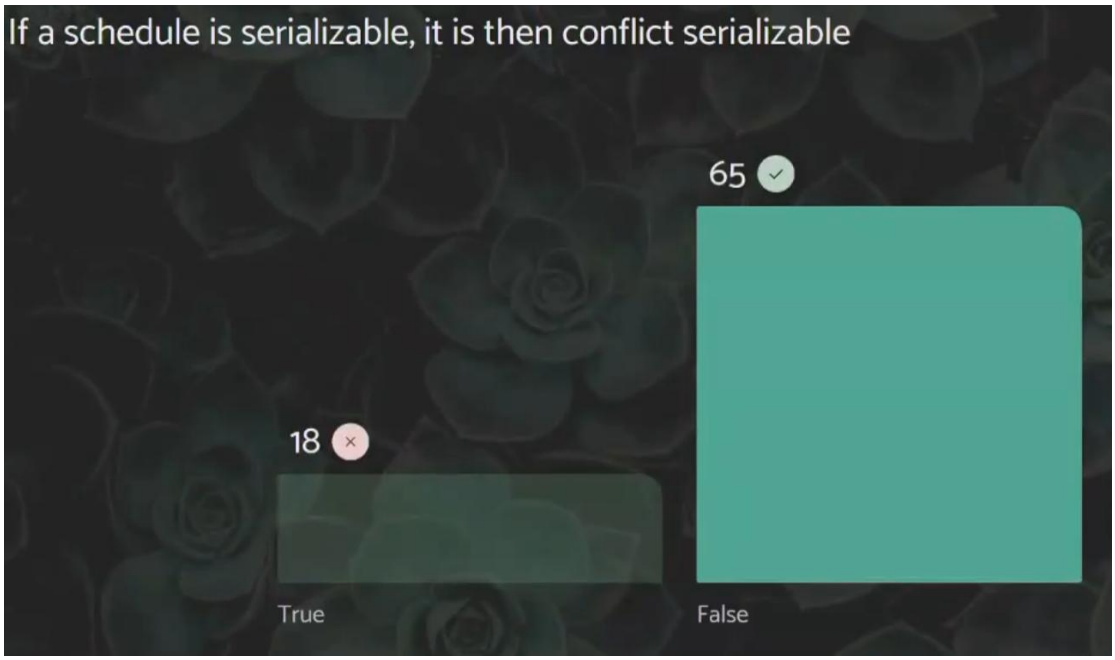

Transaction logs are used for
94
ensuring durability
14
auditing purposes


Lost/temporary/incorrect summary problems occur in
87
24
serial executions of transactions
concurrent executions of transactions

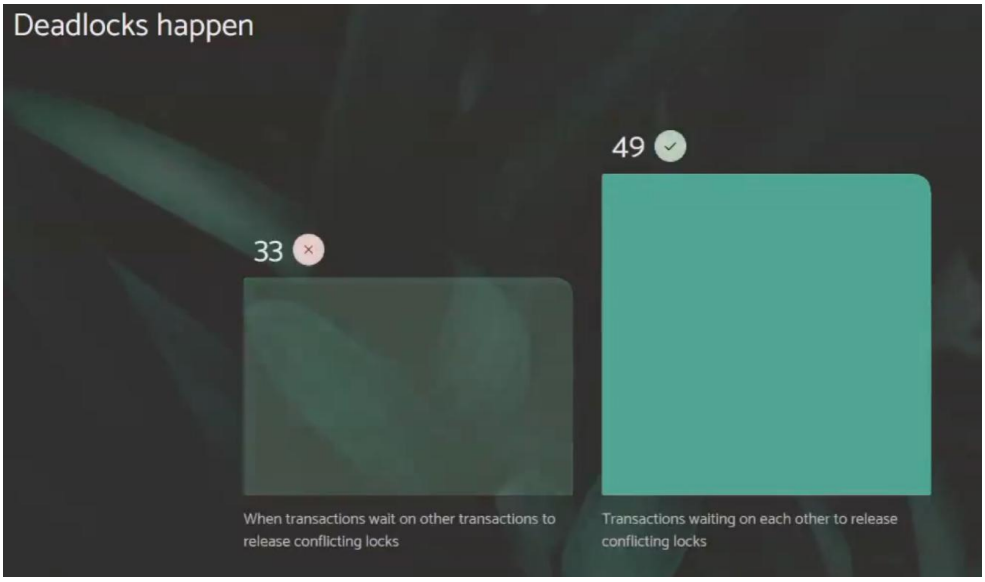

The isolation property of transactions refers to
106
5
ensuring durability
ensuring that the effect of a concurrent execution of transactions is equivalent to some serial execution of these transactions

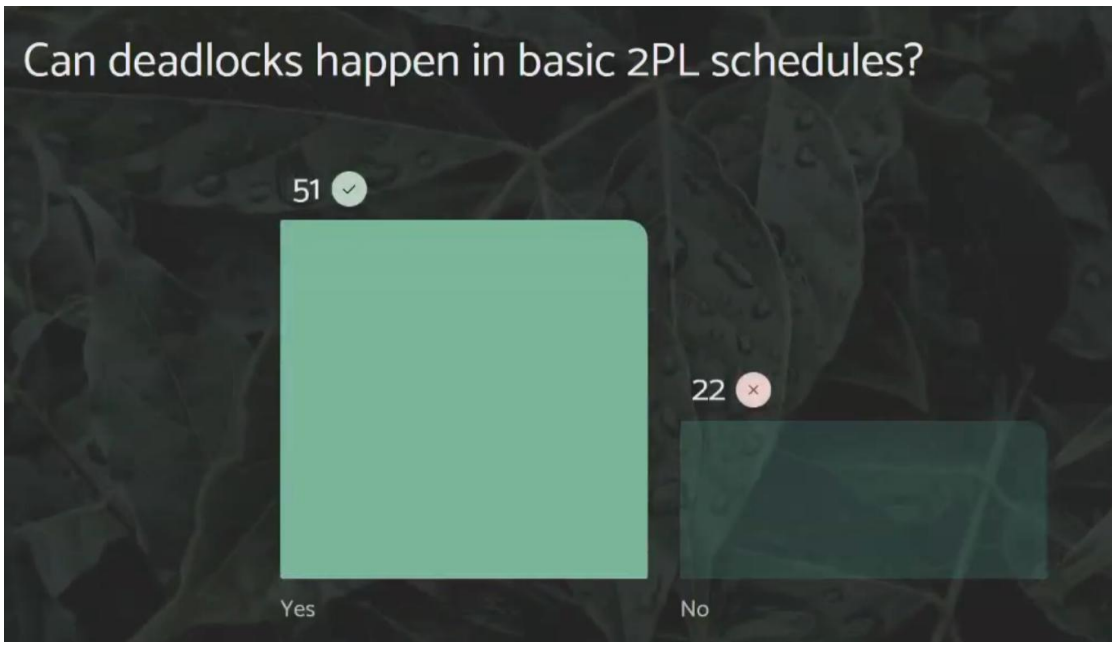
Week11

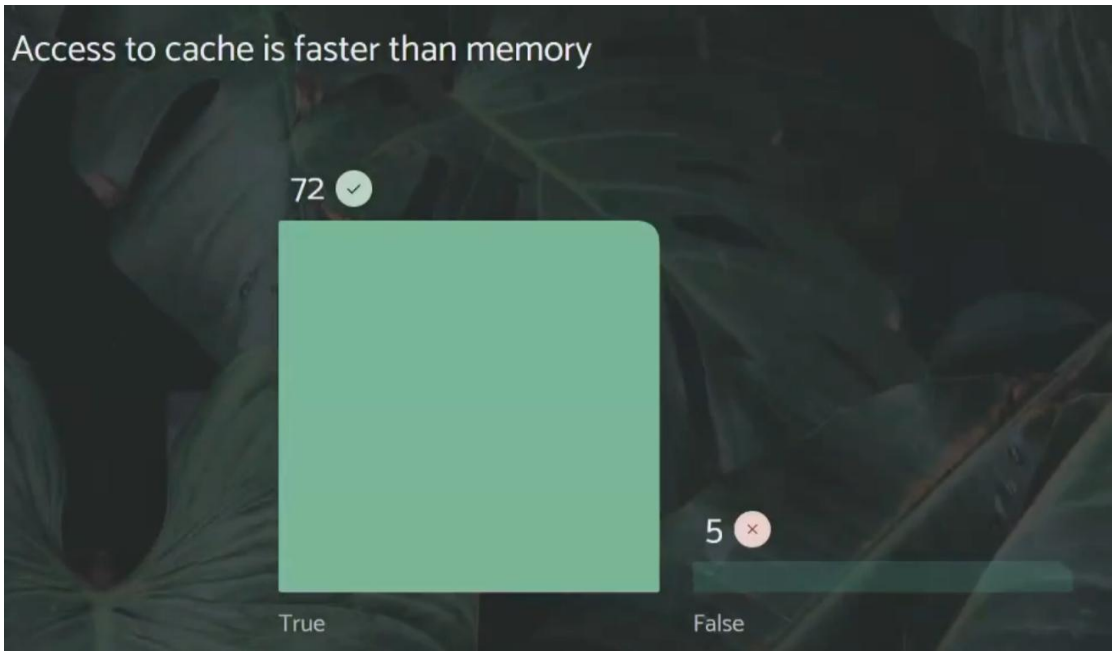

2-Phase Locking main aim is to guarantee
39
38
No deadlocks would occur
Serializabiity

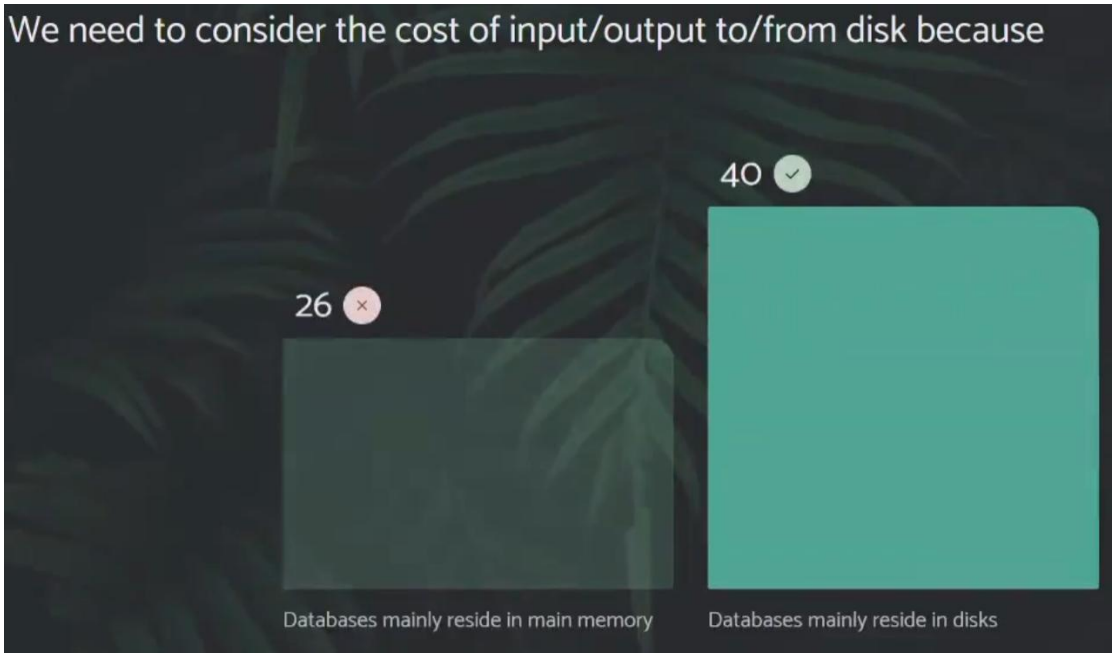

If a schedule is serializable, it is then conflict serializable
65
18
True
False

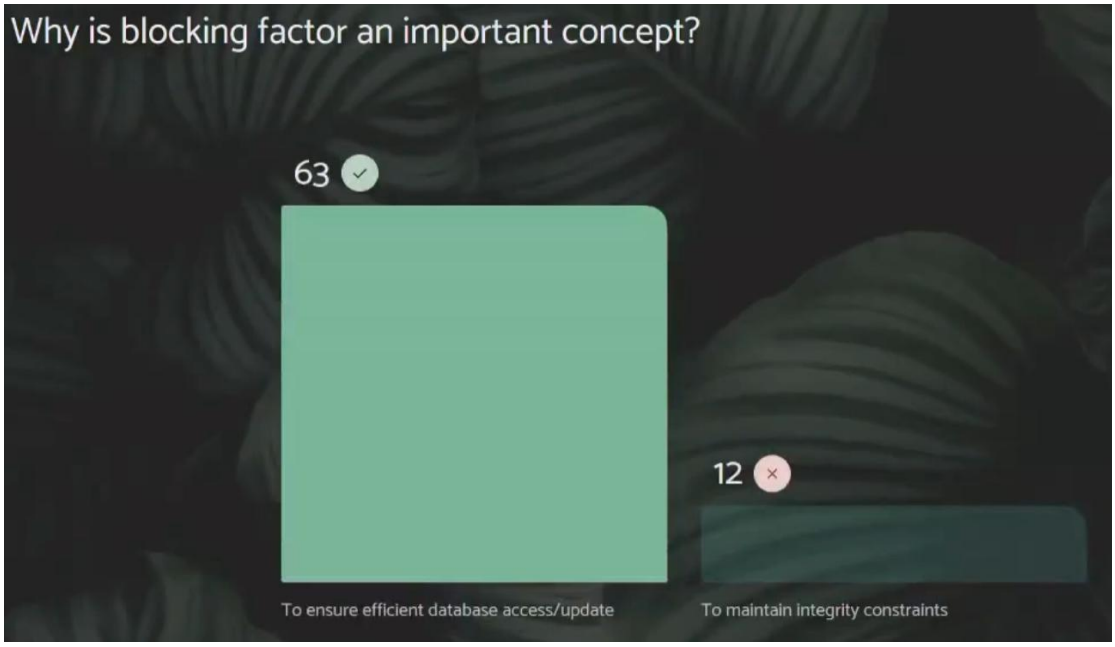

Conflict swappings are used in
53
16
14
Detecting deadlocks
Conflict serializability
Neither of the above


Deadlocks happen
49
33
When transactions wait on other transactions to release conflicting locks
Transactions waiting on each other to release conflicting locks

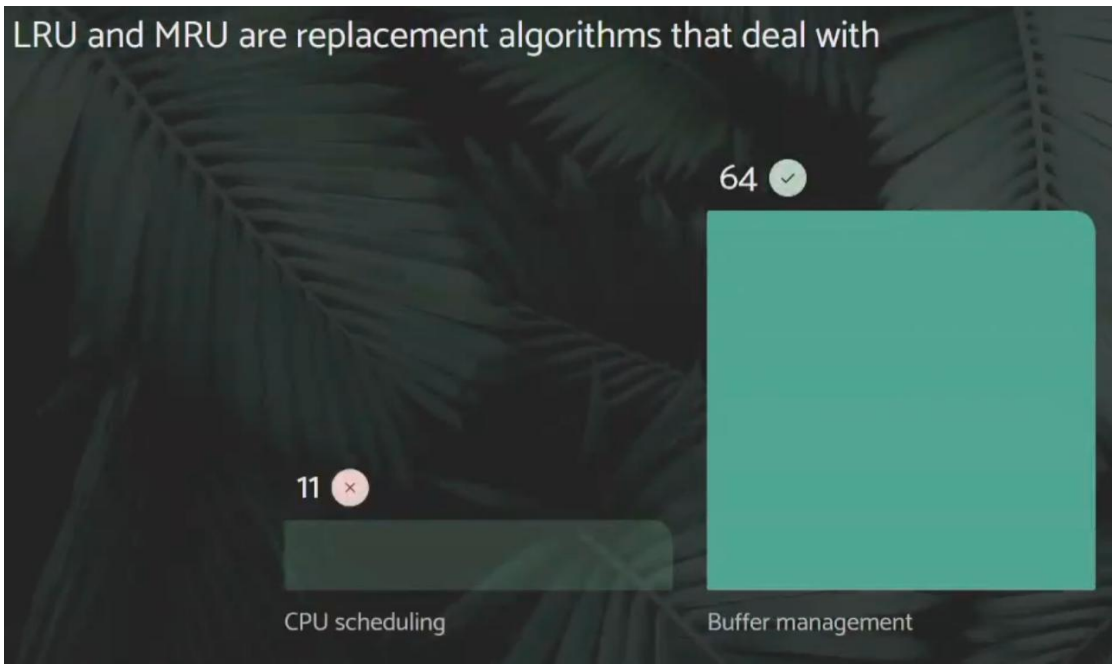

Can deadlocks happen in basic 2PL schedules?
51
22
Yes
No


Access to cache is faster than memory
72
5
True
False

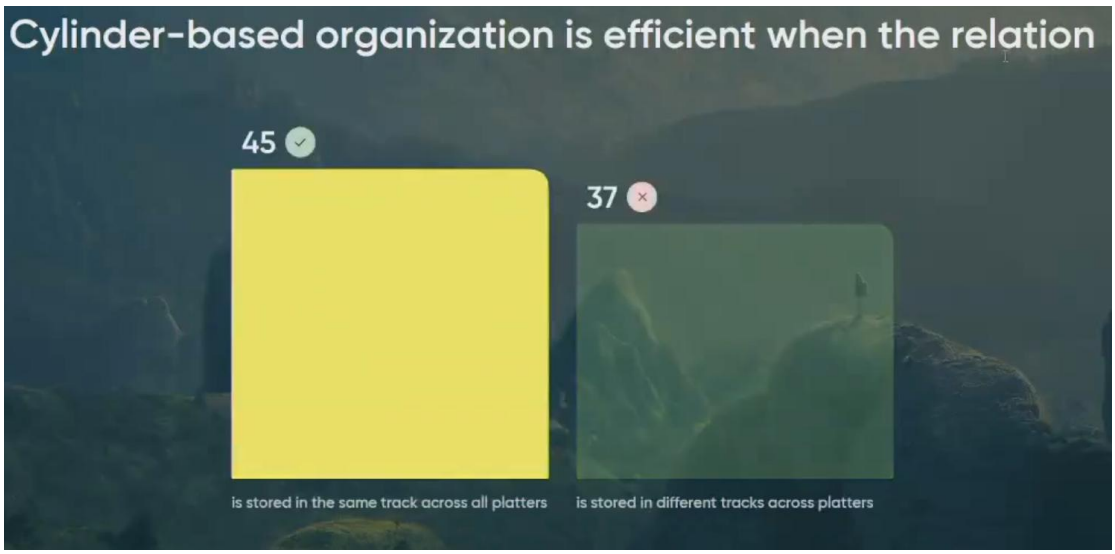

We need to consider the cost of input/output to/from disk because
40
26
Databases mainly reside in main memory
Databases mainly reside in disks

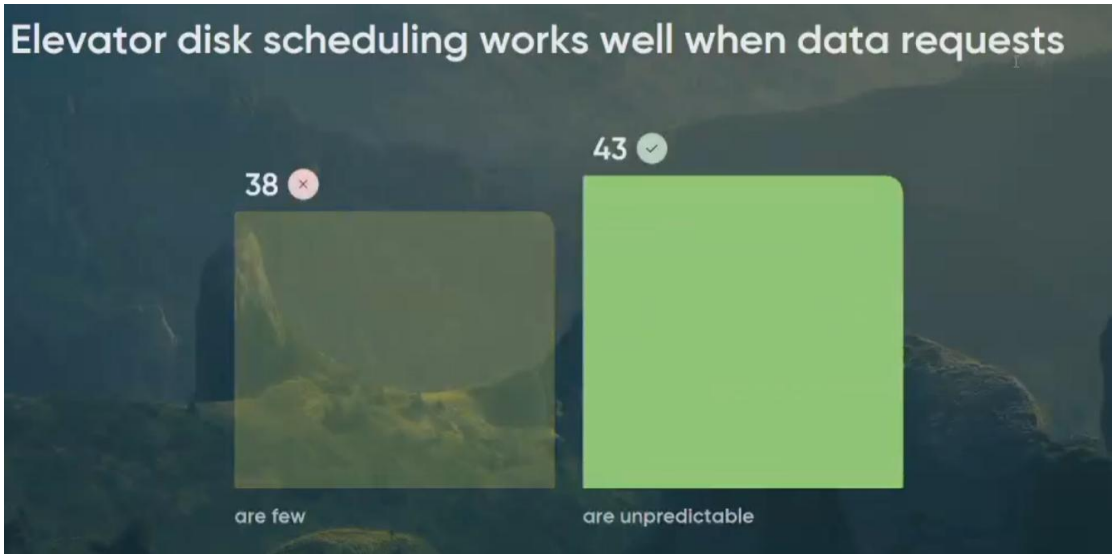

Why is blocking factor an important concept?
63
12
To ensure efficient database access/update
To maintain integrity constraints

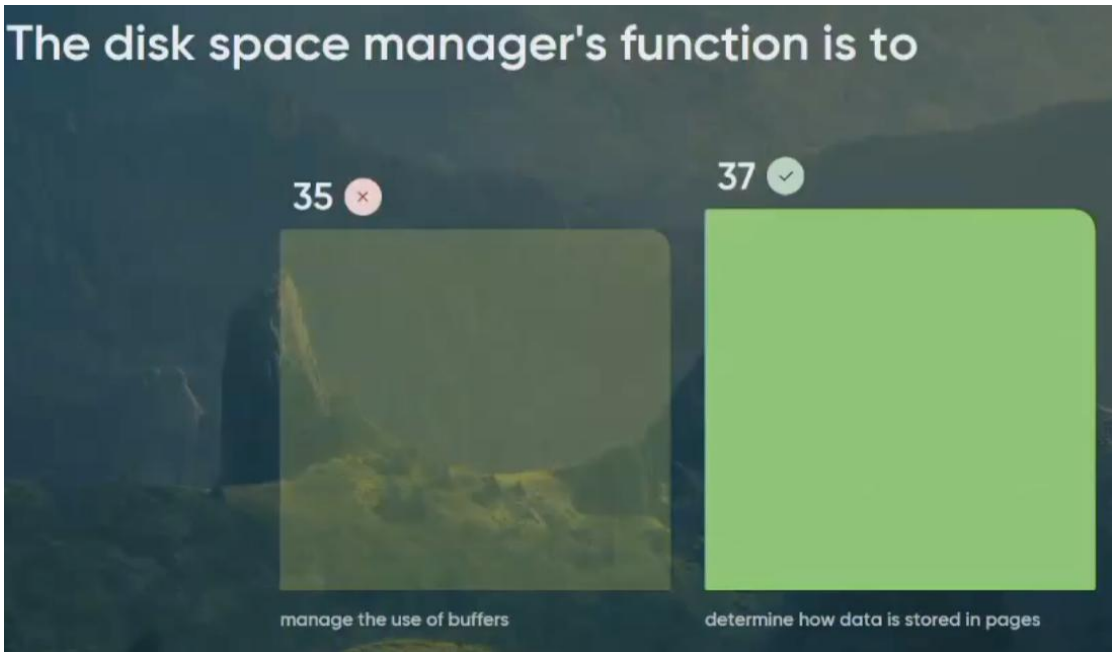

Buffer management provides
38
39
Virtual memory for in memory operations
An efficient framework to distribute blocks on disks


LRU and MRU are replacement algorithms that deal with
64
11
CPU scheduling
Buffer management

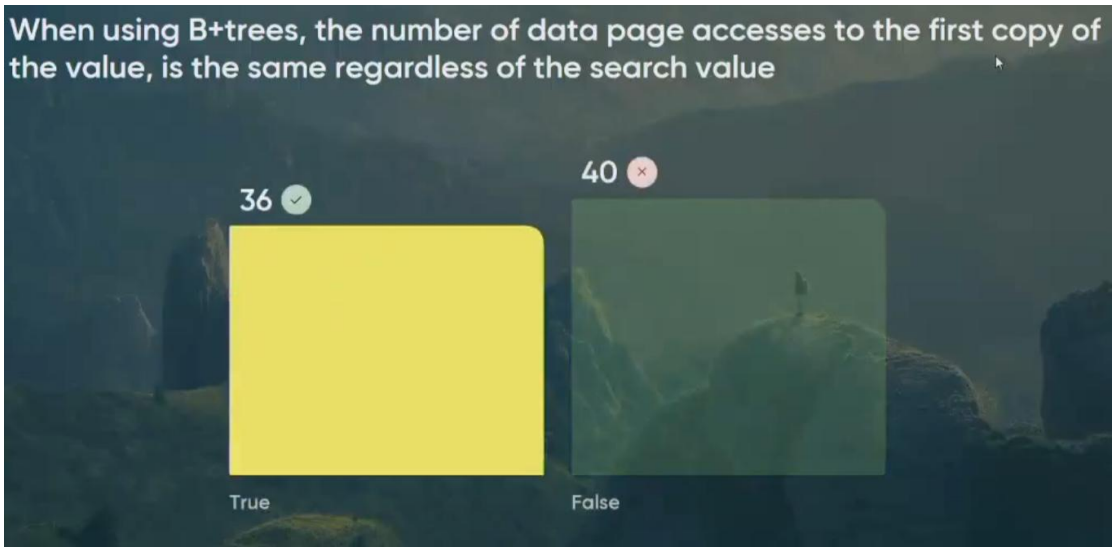
Week12

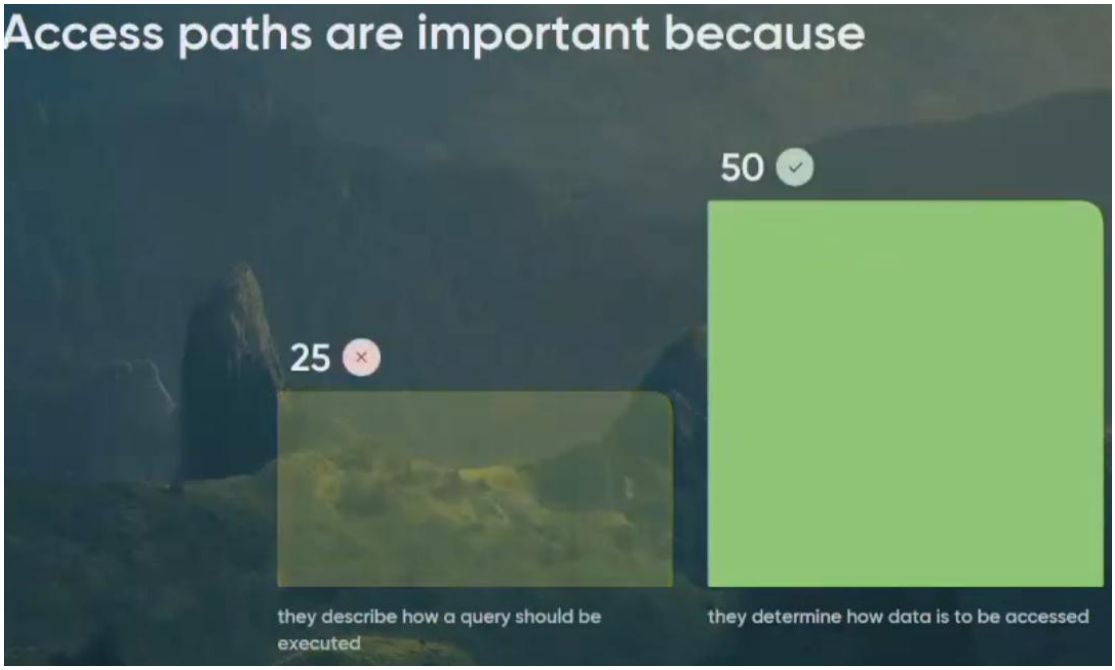

Cylinder-based organization is efficient when the relation
45
37
is stored in the same track across all platters
is stored in different tracks across platters


Elevator disk scheduling works well when data requests
38
43
are few
are unpredictable

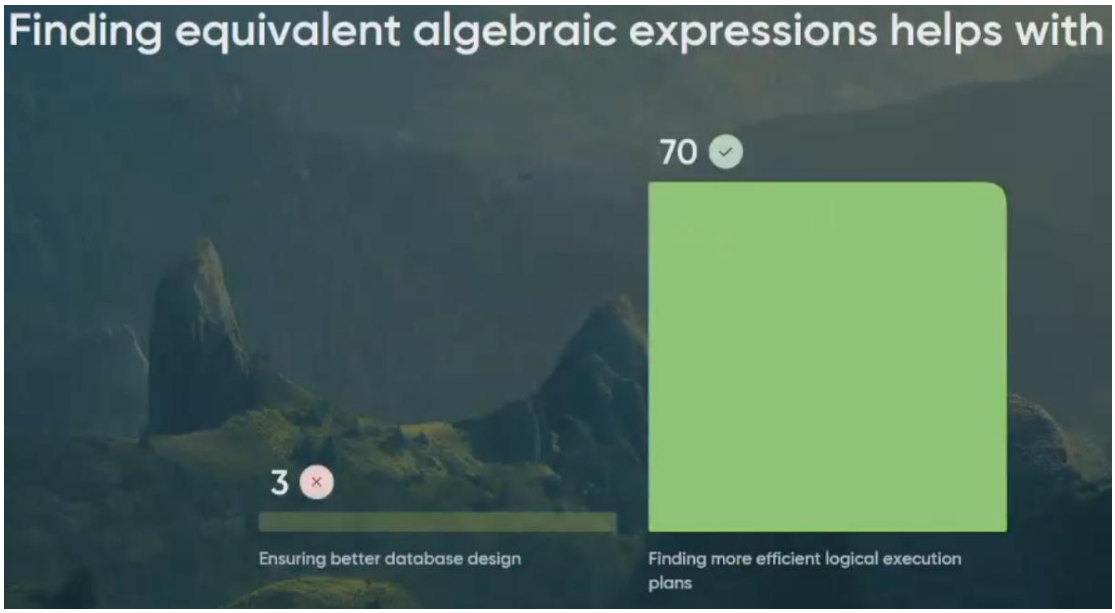

The disk space manager's function is to
35
37
manage the use of buffers
determine how data is stored in pages

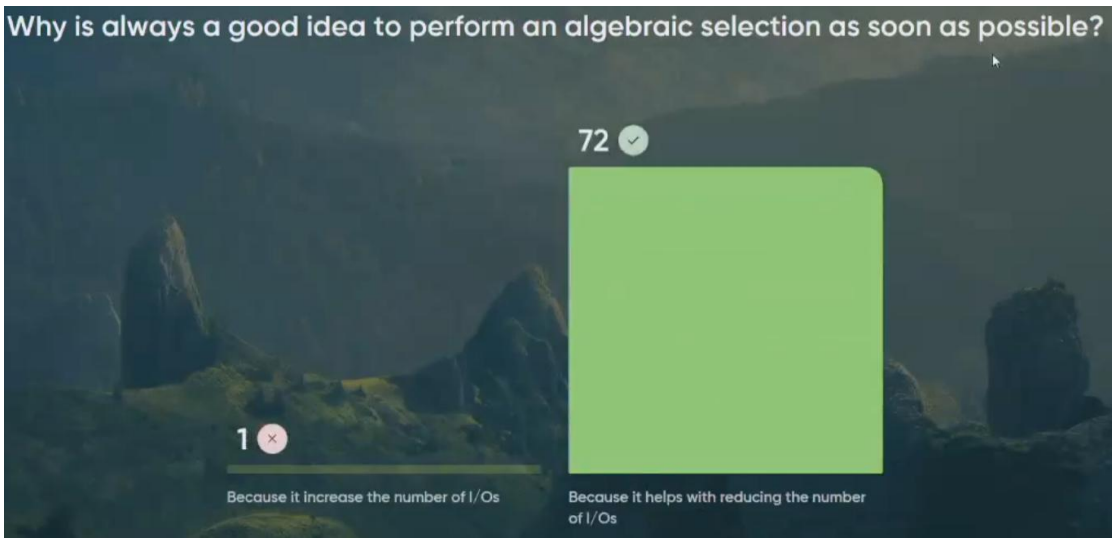

Binary search can be used for any type of data
66
11
True
False

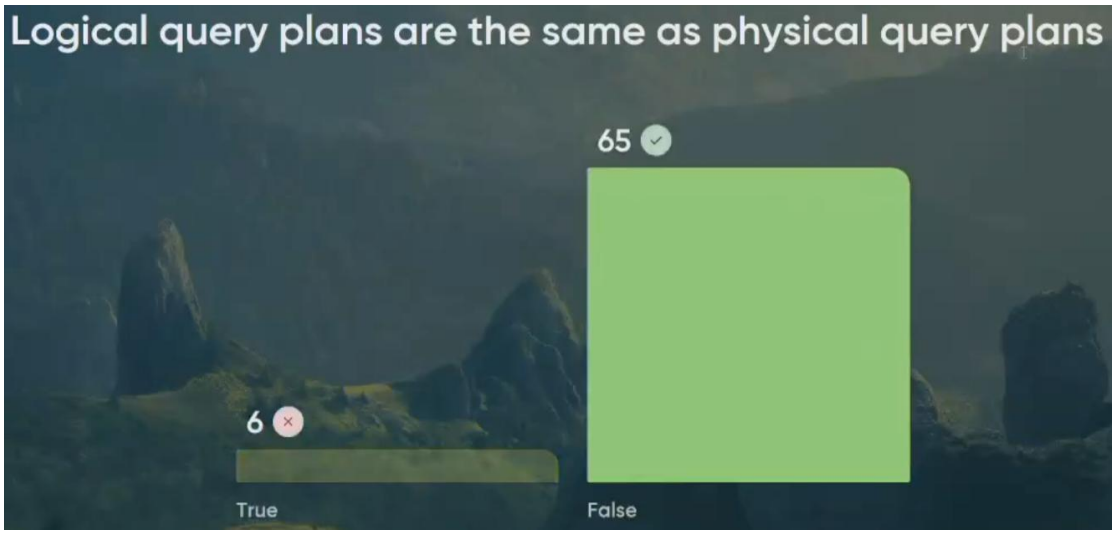

When using B+trees, the number of data page accesses to the first copy of the value, is the same regardless of the search value
36
40
True
False


Access paths are important because
50
25
they describe how a query should be executed
they determine how data is to be accessed

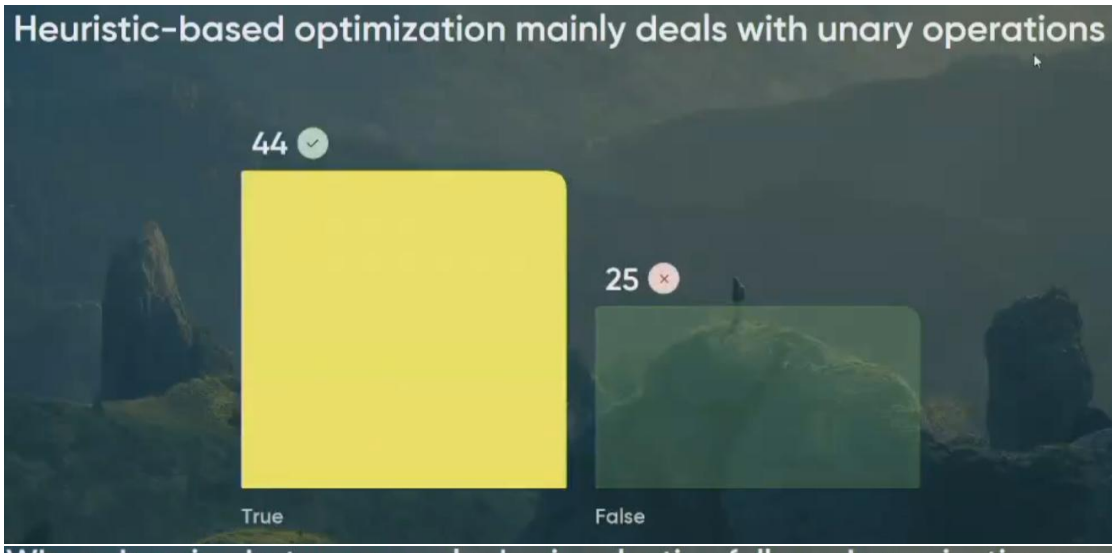

The use of any index is always useful no matter what the search key is
62
11
True
False

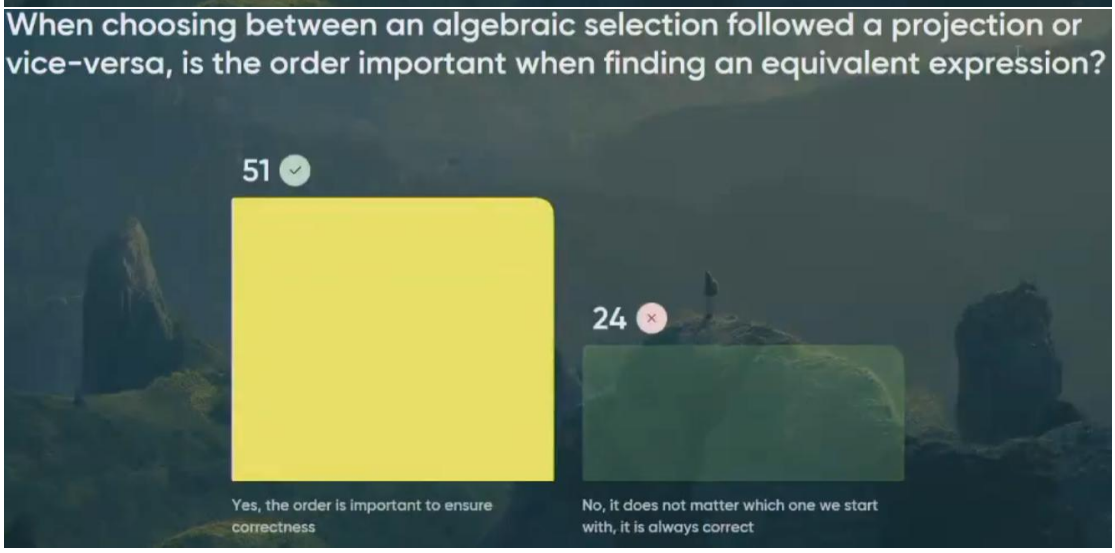

Finding equivalent algebraic expressions helps with
70
3
Ensuring better database design
Finding more efficient logical execution plans


Why is always a good idea to perform an algebraic selection as soon as possible?
72
1
Because it increase the number of I/Os
Because it helps with reducing the number of I/Os


Logical query plans are the same as physical query plans
65
6
True
False


Heuristic-based optimization is about
61
10
Enforcing integrity constraints
Minimizing I/Os


Heuristic-based optimization mainly deals with unary operations
44
25
True
False


When choosing between an algebraic selection followed a projection or vice-versa, is the order important when finding an equivalent expression?
51
24
Yes, the order is important to ensure correctness
No, it does not matter which one we start with, it is always correct

Example

```sql
DELETE FROM Language WHERE name = 'German';
```

```sql
INSERT INTO Actor VALUES (4711, 'Arnold', 'Schwarzenegger', 'AT');
```

```sql
UPDATE Film
SET rental_rate = rental_rate *1.2
WHERE title = 'ANGELS LIFE';
```

```sql
SELECT COUNT(*)
FROM Film
WHERE length > 180;
-- 但是WHERE length / 3.0 > 60; 就可以，因为这样的除法是小数，不会进行下面的整除

-- length / 3 用整数除法，直接把结果向下取整！181 / 3 = 60.333...

-- 整数除法结果是 60 （小数部分被直接截断）

-- 60 > 60 是 ❌不成立！
```

也就是说在这里，如果用 length/3，那么当 length = 181 的时候就不会算入，因为 181/3=60

```sql
-- 记得在select的时候表明是从哪个表里获得的字段
-- FROM 表在outer join的时候需要注意顺序

SELECT f.title, COALESCE(a.first_name, 'N/A') actor_firstname, COALESCE(a.last_name, 'N/A') actor_lastname
FROM Film f
LEFT JOIN Film_Actor fa
ON f.film_id = fa.film_id
LEFT JOIN Actor a
ON fa.actor_id = a.actor_id
WHERE f.release_year = 2004;
```

```sql
-- 注意union的也可以被ORDER BY
-- 只用在末尾添加;
SELECT name
FROM Country
UNION
SELECT name
FROM Language

ORDER BY name;
```

Write an SQL query that finds all films (by film_id and title) categorised as both “Drama” and “Family” film.

```
-- 这个很重要
SELECT f.film_id, f.title
FROM Film f
JOIN Film_Category fc ON f.film_id = fc.film_id
JOIN Category c ON fc.category_id = c.category_id
WHERE c.name = 'Drama'

INTERSECT

SELECT f.film_id, f.title
FROM Film f
JOIN Film_Category fc ON f.film_id = fc.film_id
JOIN Category c ON fc.category_id = c.category_id
WHERE c.name = 'Family';
```

在 Cartesian 里面，自己是会和自己生成 row 的

```
-- 需要考虑到自己和自己是会生成一个row的，因此要排除掉
-- 比如下面就是通过 a1.actor_id <> a2.actor_id
SELECT a1.actor_id, a1.first_name, a1.last_name
FROM Actor a1, Actor a2
WHERE a1.first_name = a2.first_name
AND a1.last_name = a2.last_name
AND a1.actor_id <> a2.actor_id
ORDER BY first_name, last_name, actor_id;
```

```
-- UTC是基准，比如中国的UTC + 8
-- SELECT EXTRACT(TIMEZONE FROM TIMESTAMP '2025-05-04 12:00:00+08:00'); 像这里是28800 秒 对应的是 8 小时
-- SELECT EXTRACT(TIMEZONE_HOUR FROM TIMESTAMP '2025-05-04 12:00:00-08:00'); -8 因为小时部分为-8
-- SELECT EXTRACT(TIMEZONE_MINUTE FROM TIMESTAMP '2025-05-04 12:00:00+08:00'); 0 因为分钟部分为0

-- 除了上面的语法外，还需要记住如何用format string来表示它们
/*
'YYYY': 四位数年份（如: 2025）

'MM': 两位数月份（如: 05）

'DD': 两位数日期（如: 04）

0 在格式化字符串中表示填充字符，用于确保数字的位数达到指定长度。
s 表示数字的正负符号
FM: 去除填充字符。确保在格式化过程中不会添加前导零或空格。


TO_CHAR(5, 'FM00') 会返回 05，即使数字是单一数字，FM 也会强制将它变成两位数字。
TO_CHAR(5, 'FM00x') 会返回 05x，因为x并不会被识别成填充符号。
TO_CHAR(-5, 'S00') 会返回 -05

和select用
SELECT TO_CHAR(CURRENT_DATE, 'YYYY');
*/

SELECT ('UTC' ||
        TO_CHAR(EXTRACT(TIMEZONE_HOUR FROM CURRENT_TIMESTAMP), 'S00') ||
        ':' ||
        TO_CHAR(EXTRACT(TIMEZONE_MINUTE FROM CURRENT_TIMESTAMP),'FM00')
) tz_offset;
```

```
SELECT film_id, title
FROM Film
-- 聚合函数不能直接用于 WHERE 子句中，需要用subquery替代
-- 聚合函数可以用于HAVING
WHERE rental_rate = (SELECT MIN(rental_rate) FROM Film) -- subquery不用;
AND release_year = 2006
ORDER BY title;
```

```
-- USING 自动合并USING()里的列名，也就是说不会产生歧义
SELECT film_id, title, actor_id, first_name, last_name
  FROM Film JOIN Film_Actor USING (film_id)
            JOIN Actor      USING (actor_id)
 WHERE film_id IN (
   SELECT film_id
     FROM Film_actor JOIN Actor USING (actor_id)
    WHERE first_name = 'PENELOPE' AND
          last_name  = 'GUINESS'
 );
```

`mod(a, b)`

Computes the remainder of `a / b`.

`round(n, d)`

Rounds `n` to `d` decimal places.

`trunc(n, d)`

Truncates `n` to `d` decimal places.

`ceil(n)`

Computes the smallest integer value not less than `n`.

`floor(n)`

Computes the largest integer value not greater than `n`.

`abs(n)`

Computes the absolute value of `n`.

Here's an example of the above functions:

▶ Run

```
1 SELECT mod(15, 4), round(15.67, 1),
2        trunc(15.67, 1), ceil(15.67),
3        floor(15.67), abs(-121.4)
```

可以在 select 和 where 里面使用

ROUND()是四舍五入，对于 150/60 来说，结果为 2；但是对于 ROUND(150/60,0)来说就是 3，对于 ROUND(150/60)来说就是 2

```
select 150/2, ROUND(150/60.0), ROUND(150/60)
```

?column?	round	round
75	3	2

如果不想四舍五入就用 TRUNC()

```sql
-- create table的;在括号外面
CREATE TABLE Film_Screening(
    screening_id INT PRIMARY KEY,
    film_id INT NOT NULL,
    theatre VARCHAR(50),
    start_time TIMESTAMP NOT NULL,
    film_start TIME,
    last_tickets DATE,

    -- 最后指定的column需要带上括号
    -- 主表和附表都要指定column
    -- 当前表不用表明                    这里要带
    FOREIGN KEY (film_id) REFERENCES Film (film_id)

);
```

```sql
SELECT COUNT(*) as count
FROM Film
-- EXTRACT里面是from不是逗号
-- 年份不确定的话，最后一个个数，25 24 23 22 21 20 19 18 17 16 而25-10 = 15
-- 还要保证年份不能超过当前
WHERE release_year >= EXTRACT(YEAR FROM CURRENT_DATE) - 9 AND release_year <= EXTRACT(YEAR FROM CURRENT_DATE);
```

```sql
SELECT COUNT(*)
FROM Film
-- 用DATE '' 可以强转数据类型为日期类
-- 需要用()来隔开string concatenate
WHERE EXTRACT(DOY FROM DATE (release_year || '-12-31')) = 366;
```

```sql
/*
    外层的 WHERE release_year = 2000 是第一道筛选;

    然后 EXISTS(...) 是逐行对外层每条记录进行条件验证;
*/
SELECT film_id, title
FROM Film f1
WHERE release_year = 2000
AND EXISTS(
    SELECT *
    FROM Film f2
    JOIN Film_Actor fa USING (film_id)
    WHERE f1.film_id = f2.film_id
)
ORDER BY title;
```

```sql
SELECT release_year, COUNT(*) number_of_films
FROM Film
GROUP BY release_year
-- chronologically 按时间顺序
-- DESC要放在后面
ORDER BY number_of_films DESC, release_year;
```

```sql
-- COUNT(*) 会计算NULL，因此这里要用COUNT(film_id)，否则那些没有对应电影的category也会有count为1

SELECT category_id, name, COUNT(film_id) count
FROM Category
LEFT JOIN Film_Category USING (category_id)
GROUP BY name, category_id
ORDER BY count DESC, name ASC;
```

```sql
SELECT film_id, title, COUNT(actor_id) count
FROM Film
JOIN Film_Actor USING (film_id)-- USING 要带上括号
GROUP BY film_id, title
-- HAVING 子句必须使用完整的聚合表达式，不能直接引用 SELECT中的alias;
-- HAVING可以使用非聚合函数，比如actor_id > 5
HAVING COUNT(actor_id) >= 5
ORDER BY title;
```

```sql
SELECT c.name country, COUNT(actor_id) num_actors
FROM Actor a
JOIN Country c
ON a.nationality = c.short_code
GROUP BY country -- GROUP BY能用SELECT中的alias
ORDER BY num_actors DESC, country
LIMIT 5;

/*
GROUP BY能用SELECT中的alias, 下面这种cancate的也可以

SELECT (c.name || c.name) country, COUNT(actor_id) num_actors
FROM Actor a
JOIN Country c
ON a.nationality = c.short_code
GROUP BY country -- GROUP BY能用SELECT中的alias
ORDER BY num_actors DESC, country
LIMIT 5;
*/

-- ordinal ranking: 每个条目都有独立编号，即使有相同值，也不会并列; ROW_NUMBER()
-- dense ranking: 如果有相同值，会并列排名，下一个排名不会跳过; DENSE_RANK()
-- Standard ranking (或叫 sparse ranking): 有相同值时会并列，下一个排名会跳过; RANK()

SELECT first_name, last_name, COUNT(film_id) films
FROM Actor
JOIN Film_Actor USING (actor_id)
WHERE nationality = 'US'
GROUP BY actor_id, first_name, last_name -- 注意！！名字组合可能不唯一，因此要用actor_id来区分
ORDER BY films DESC, first_name, last_name
LIMIT 10;


-- rank() OVER (ORDER BY mark DESC) 是窗口函数，只是生成了一列排名，不影响结果行的顺序。
-- RANK()给的是Standard ranking
SELECT RANK() OVER (ORDER BY COUNT(film_id) DESC) rank, name category, COUNT(film_id) films
FROM Category
JOIN Film_Category USING (category_id)
GROUP BY name
ORDER BY rank, name
LIMIT 5;
```

```sql
-- 只有date可以直接运算，而interval只是一段时间

SELECT film_id, title, rental_rate, replacement_cost,
    GREATEST(rental_rate*(DATE '2016-04-12' - DATE '2016-03-24'),
    replacement_cost) late_fee
FROM Film f
JOIN Film_Actor fa USING (film_id)
JOIN Actor a USING (actor_id)
WHERE (a.first_name || ' ' || a.last_name) = 'JOE SWANK'


-- STRING_AGG(expression, separator ORDER BY attributes)
-- 他就是一个聚合函数，和MAX()之类的并无差别

SELECT film_id, title, STRING_AGG(name, ',' ORDER BY name) cat_agg
FROM Film
JOIN Film_Category USING (film_id)
JOIN Category USING (category_id)
GROUP BY film_id, title;


/*
CREATE DOMAIN GradingScheme
       CHAR(2) CHECK (value in ('FA','PA','CR','DI','HD'));
创建自定义数据类型
*/
CREATE DOMAIN Distance INTEGER CHECK (value BETWEEN 0 AND 25000);

CREATE TABLE Airplane(
       model VARCHAR(10) PRIMARY KEY,
       manufacturer VARCHAR(50) NOT NULL,
       flight_range Distance,
       cruise_speed INT,
       enginetype VARCHAR(50) DEFAULT 'turbofan',
       first_flight DATE,

       -- check condition需要括号包裹
       CHECK (enginetype in ('turbofan', 'turbojet', 'turboprop')),
       CHECK (cruise_speed > 200)
       -- BETWEEN不能写成25000 AND 0
       -- CHECK (flight_range BETWEEN 0 AND 25000)
);
```

```
/*
记住 ON DELETE/UPDATE
        CASCADE
        SET NULL
        SET DEFAULT
两个可以同时存在，比如 ON DELETE .... ON UPDATE ....
*/
CREATE TABLE Inventory(
    inventory_id INT PRIMARY KEY,
    film_id SMALLINT NOT NULL,
    store_id INT,
    quantity INT,
    last_update TIMESTAMP NOT NULL,

    --  FK前面这里也要带上括号
    FOREIGN KEY (film_id) REFERENCES film (film_id) ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (store_id) REFERENCES Store (store_id) ON DELETE SET NULL ON UPDATE CASCADE,
    CHECK (quantity >= 0)
);
```

```
CREATE OR REPLACE FUNCTION trg_check_vehicle_classification_fn()
RETURNS TRIGGER AS $$
BEGIN
    IF NOT EXISTS (
        SELECT 1 FROM NewCar WHERE VIN = NEW.VIN
    ) AND NOT EXISTS (
        SELECT 1 FROM PreOwnedCar WHERE VIN = NEW.VIN
    ) THEN
        RAISE EXCEPTION 'Vehicle (VIN=%) must appear in either NewCar or PreOwnedCar', NEW.VIN;
    END IF;

    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

▷Run | Select
CREATE CONSTRAINT TRIGGER trg_check_vehicle_classification
AFTER INSERT OR UPDATE ON Vehicle
FOR EACH ROW
EXECUTE FUNCTION trg_check_vehicle_classification_fn();
```

```
CREATE OR REPLACE FUNCTION on_sale_must_have_listing()
RETURNS TRIGGER AS $$
BEGIN
    IF(
        NEW.Status = 'for sale'
    )
    THEN
        IF NOT EXISTS(
            SELECT 1 FROM VehicleListing WHERE VIN = NEW.VIN
        )
        THEN
            RAISE EXCEPTION 'Car % does no have vehicle listing while it is curretly for sale!', NEW.VIN;
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

▷Run | Select
CREATE CONSTRAINT TRIGGER trg_on_sale_must_have_listing
AFTER INSERT ON Vehicle
FOR EACH ROW
EXECUTE FUNCTION on_sale_must_have_listing();
```

```sql
CREATE OR REPLACE FUNCTION check_car_on_sale()
RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (
        SELECT 1 FROM Vehicle
        WHERE VIN = NEW.VIN AND Status = 'for sale'
    ) THEN
        -- 9
        UPDATE Vehicle
        SET Status = 'has been sold'
        WHERE VIN = NEW.VIN;
        RETURN NEW;
    ELSE
        RAISE EXCEPTION 'Vehicle with VIN % is not for sale or not inside table', NEW.VIN;
    END IF;
END;

$$ LANGUAGE plpgsql;
```

Run | Select

```sql
CREATE TRIGGER trg_check_car_on_sale
BEFORE INSERT ON Sale
FOR EACH ROW
EXECUTE FUNCTION check_car_on_sale();
```