Class8-DB-Dec17

Assign	
Status	Completed
Priority	
Date Created	@December 17, 2021 6:33 PM
Due Date	
Property	

9. Transaction

A transaction is an action, or a series of actions, carried out by a single user or an application program

all the transaction follow ACID principle

ACID represent:

- Atomicity
    - all the transaction are atomic , and we can not execute partially
    - **for keeping Atomicity (two ways)→ commit or rollback**
- Consistency
    - transactions take the database from one consistent state to another state
- Isolation(has different levels)
    - a transaction is not visible to other transactions until it competes
- Durability
    - Once a transaction has completed, its changes are made **permanent**

e.g.

Transaction: account A → account B, A send 100 to B

consist of 6 following steps (server by server, finish one and then jump to another)

step 1: read A for check the account balance

step 2: minus 100 from A

step 3: write A

step 4: read B

step 5: add 100 for B

step 6: write B

In the example A→ B previously, **atomicity** means the transaction should not be completed, when there is any interruption during step 1 to step 6 (or miss any one). If not, all the steps will be rollback(recover to before step 1)

A (**means**): shouldn't take money from A without giving it to B

C: money isn't lost or grained A+B

- the total number (A+B) of money can not increase or decrease during whole the transaction process

I: other queries shouldn't see A or B change until completion

D: when transaction done, the money doesn't not go back to A

10. Concurrency (in DB)

can implement several transactions in the same time

Three potential problems

Dirty Read

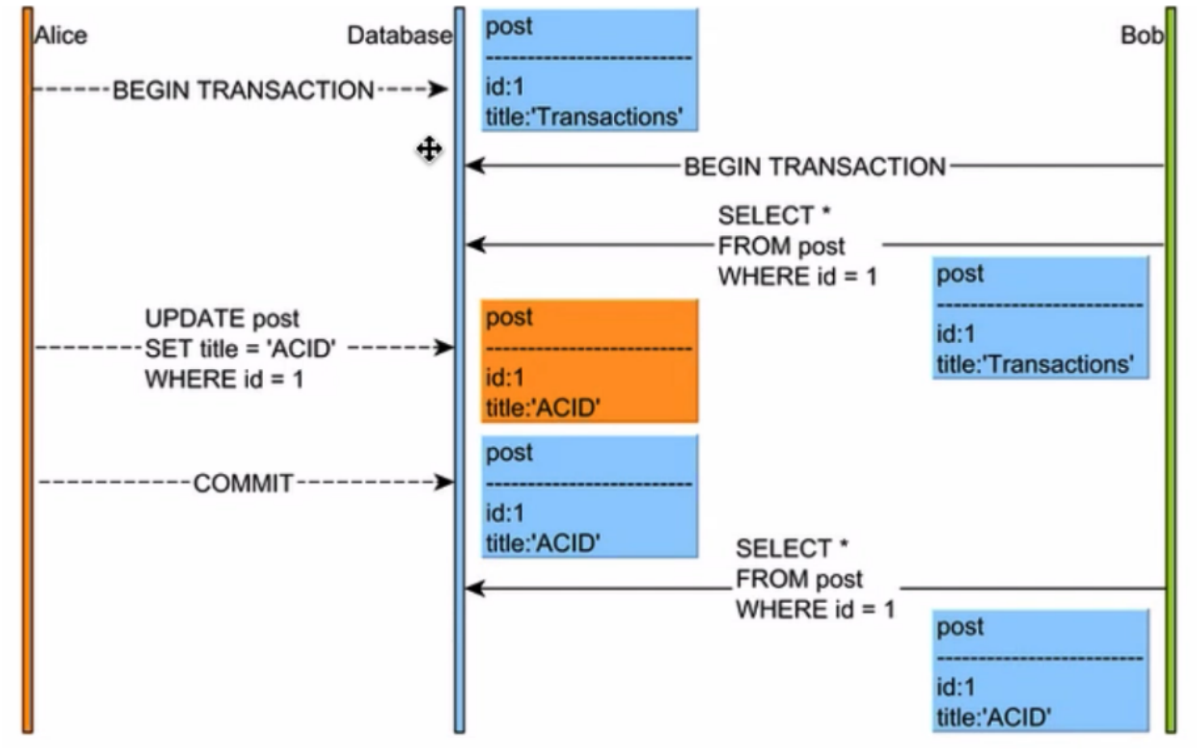
time	Transaction 1	Transaction 2
1	read x	
2	x = x - n	
3	write x	
4		read x
5		x = x + m
..		write
	...	commit
	...	
	Rollback	

Dirty Read above. There is any other transaction read x, when x is processing in a transaction. X will be rollback.

Unrepeatable read

*B want to read an item twice(in **ONE** Transaction), but after first read operation, A changes the name of the item(has committed), so when B read again, it can not confirm two reading operation read a same item or two items*

the Non-repeatable Read anomaly looks as follows:



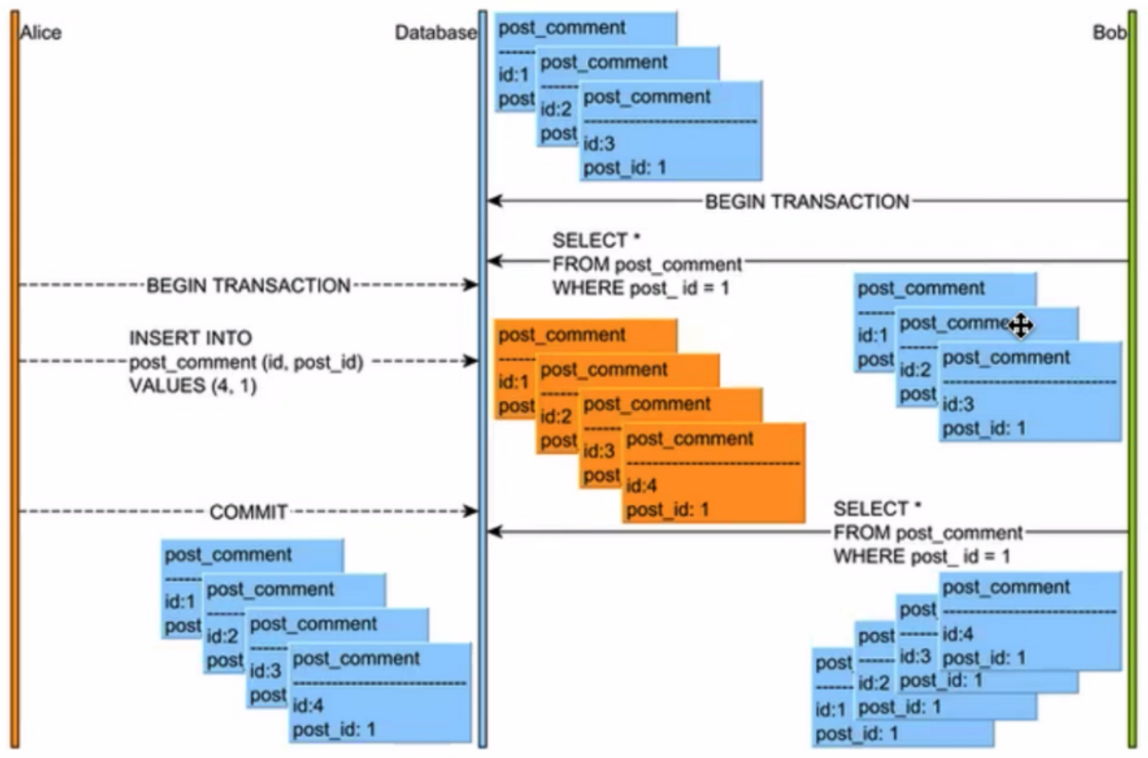
1. Alice and Bob start two database transactions
2. Bob's reads the post record and title column value is Transactions
3. Alice modifies the title of a given post record to the value of ACID
4. Alice commits her database transaction
5. If Bob's re-reads the post record, he will observe a different version of this table row

Only happen in the **Update scenario**

Phantom(幻影) Read

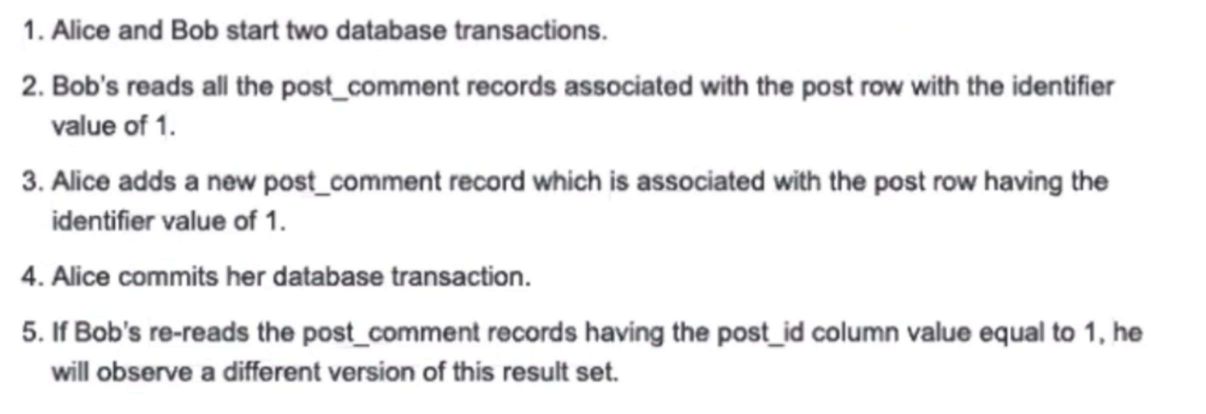
*B want to read a set of data twice(**ONE** Transaction, **Before** commit). First time B reads 3 items, and then A insert an item(commit). Second read, B reads 4 items.*

The **Phantom Read** anomaly can happen as follows:



problem: **can not get a constant number of result**

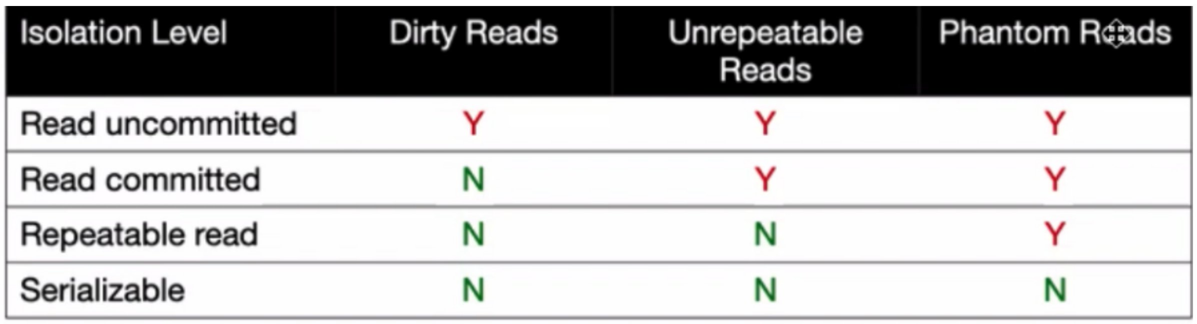
happen in **Insert** or **delete scenario**



1. Alice and Bob start two database transactions.
2. Bob's reads all the post_comment records associated with the post row with the identifier value of 1.
3. Alice adds a new post_comment record which is associated with the post row having the identifier value of 1.
4. Alice commits her database transaction.
5. If Bob's re-reads the post_comment records having the post_id column value equal to 1, he will observe a different version of this result set.

Cuz there are the problems above, we need tools to keep "Read" safe (security)

We don’t have to implement the lock, DB will help us do that, we just use it



Isolation Level	Dirty Reads	Unrepeatable Reads	Phantom Reads
Read uncommitted	Y	Y	Y
Read committed	N	Y	Y
Repeatable read	N	N	Y
Serializable	N	N	N

Isolation LeveL:

set Read uncommitted/Read committed/Repeatable Read/Serializable

RU: can read uncommitted data

RC: only read committed data

RR: to prevent unrepeatable data by locking the specific record

S: highest level can prevent all the problems

⚠️ Summary: (**Interview AND Evaluation**)

- dirty read: read uncommitted data from another transaction
- non-repeatable read: read committed data from an update query from another transaction
- phantom read: read committed data from an insert or delete query from another

Different Isolation level implement different locks to help us to manage transactions

Read uncommitted doesn’t implement any locks (have lots of problem)

Repeatable read add the lock on the specific record, because it add lock on the one record, another transaction can not update it. → prevent unrepeatable read

Think Deeply:

> When a record is locked, it is safe. However, when a transaction tries to access the record, it will be waiting until it being released. → system performance lower.

Serializable add locks on bunch of records (the whole range of record to make other transaction can not change any records in this range)

detail implementation will be related to read lock and write lock, and some other locks

```
// Spring
@Transactional(isolation_levl = " ")
public void method() {
}
```

11. Lock

btw, CAS (Deeply, JAVA lock)

binary lock:

- 1, 0
- a column named locked:
  - value is 0 means not be locked
  - value is 1 means be locked

shared and exclusive locks

- shared lock: read lock
  - allow different transactions read the data at same time, **BUT** write data
- exclusive lock: write lock

    - only one can **read** and **write** data

Homework 3 - 1:

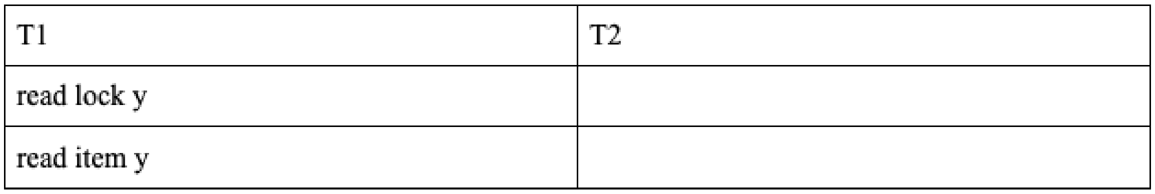
Optimistic lock and pessimistic lock

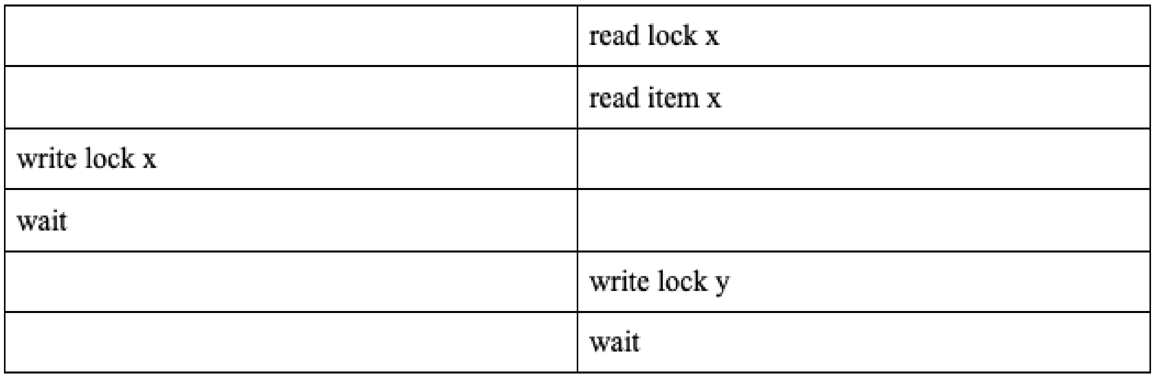
What are they and how do they implement, what difference between them?

and which scenarios use OL, and which use PL

DeadLock

- both locks waiting for each other, both transaction wait for other transactions to release locks





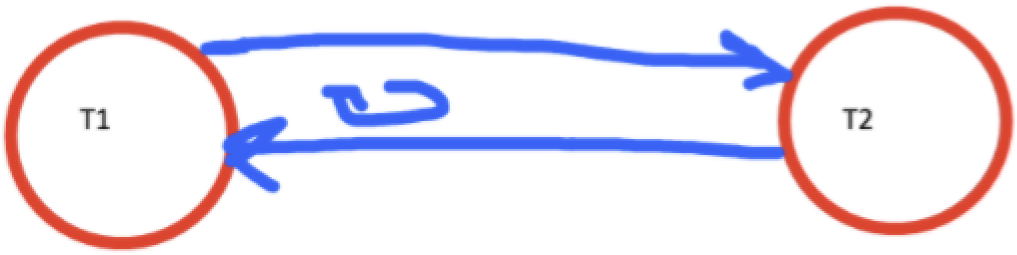
T1	T2
read lock y	
read item y	
	read lock x
	read item x
write lock x	
wait	
	write lock y
	wait

In the scenario above, Transaction 1 locked y and T2 locked x, then T1 want to write x, who is waiting for x to be released by T2, and simultaneously, T2 is waiting for T1 to release y to access

Solution:

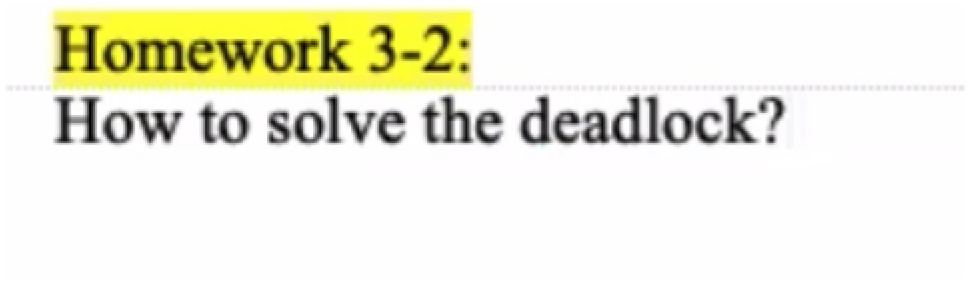
1. detect

   how to detect the deadlock → wait for graph



   There is one direction line between T1 to T2, and also have one direction line between T2 to T1. If there is a circle, it will be a deadlock.

2. how to handle(solve) the deadlock?



Homework 3-2:
How to solve the deadlock?

What is Optimistic lock?
Optimistic Lock is a strategy where you read a record, and take note of a version number or
timestamps or checksums/hashes and check that the version hasn't changed before you write

the record back.
If the info(version, sum/hash) is same, just finish writing. If the info is different, should discard
the record and can not update(write).

What is pessimistic lock?
Pessimistic lock is when a user starts to update a record, there will be a lock placed on the
record. Any others want to update the record simultaneously will be informed that there is an
update in process. The other ones should wait until the previous update to be finished(committed).

12. distributed transaction

In microservice architecture

e.g

Order piazza scenario

user → custom Microservice → order Microservice → refund Microservice

That is one transaction, but go through several Microservices

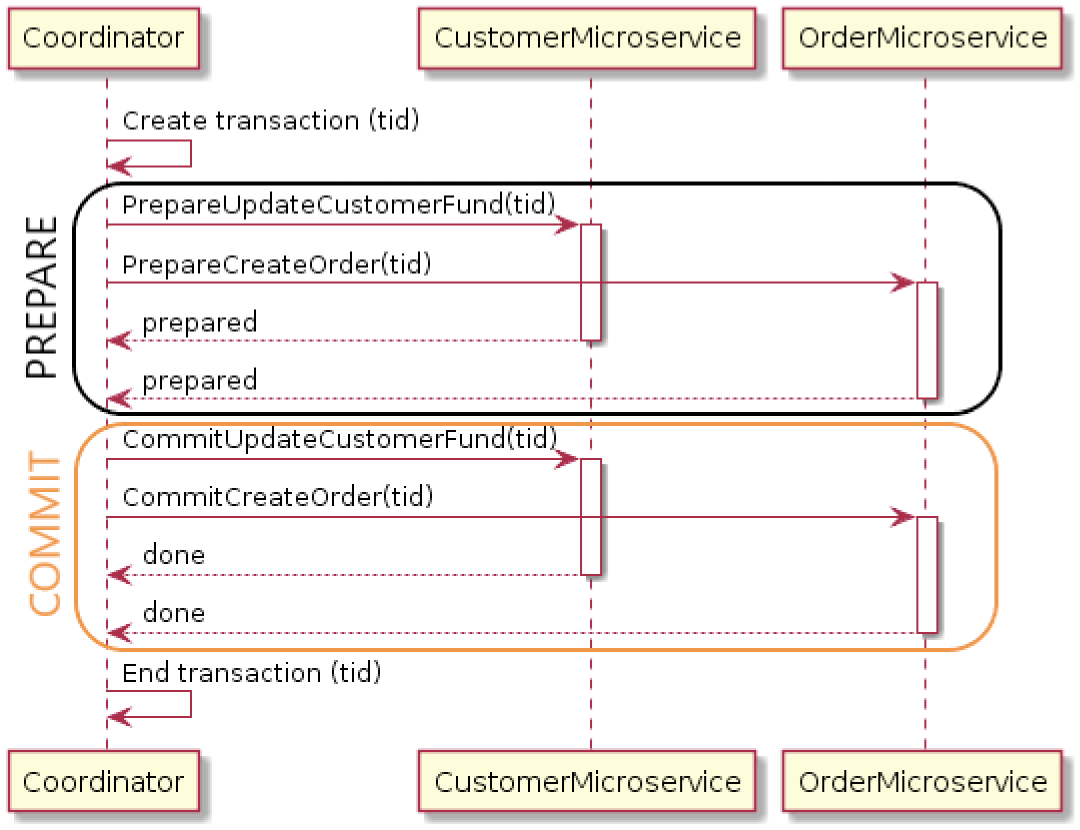
How to handle it

Think Deeply:

> When a user try to start a task, if not determine all services is ready, it could be failure of transaction

2PC (Two phase commit, design pattern)

- prepare phase (stage)
- commit phase (stage)



On prepare phase, coordinator create transaction and then check status of all the service to be accessed, whether is prepared (receive “prepared” response)

After all services prepared will enter commit phase, tasks can be assigned in the phase.

When tasks finish, it will send “done” to coordinator. All the microservice finish their tasks, the transaction can be ended.

Problem: if one service is not prepared or so slow to finish the task, all the process of transaction will be waiting for it, which makes low performance.

Homework 3-3

Homework 3-3

- Saga:

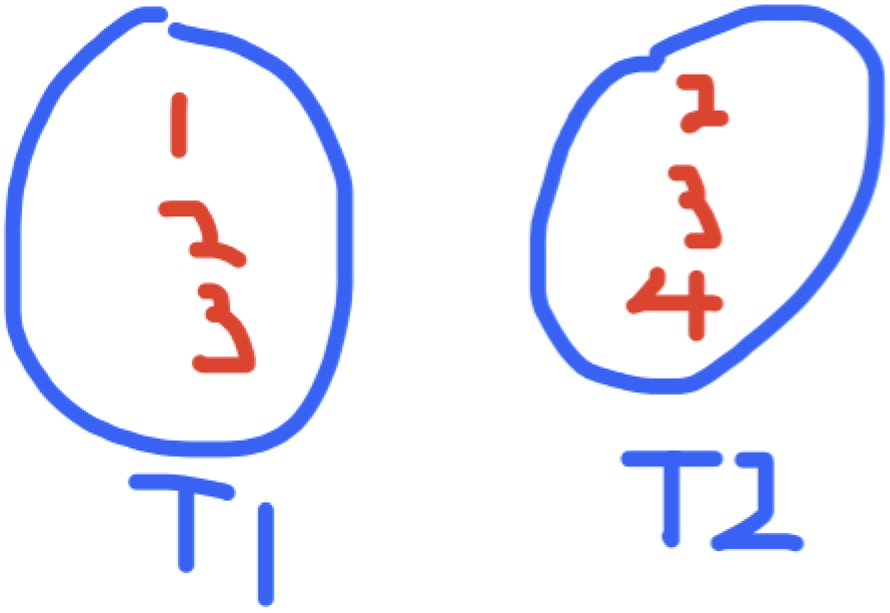
saga design pattern

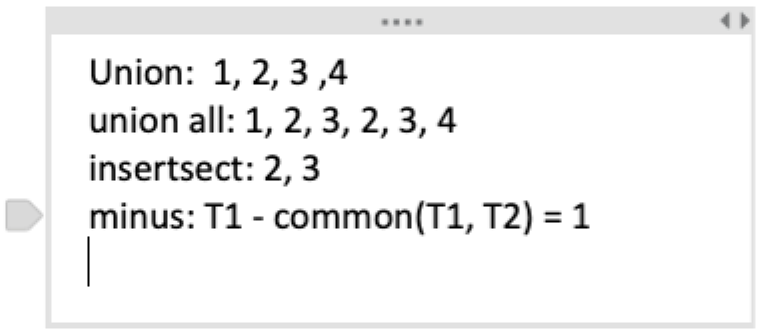
what is saga?

Saga is **a sequence of local transactions**. It works that replace one transaction in 2PC model, each local transaction **updates the data and sends message** or even to **trigger the next local transaction(other service)**. If a local transaction fail, the saga will **execute a series** of compensating **transactions** to **undo** the changes(what the preceding local transactions did, look like **rollback**)

13. SQL

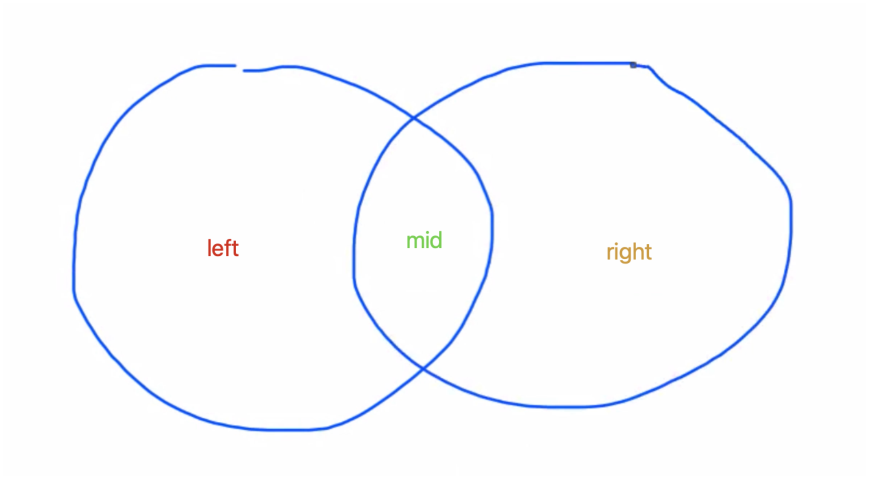
```
select * from ...
// aggregation function: max, min, count, avg, sum
// aggregation: n. 聚集, 集成；集结
// union, union all, intersect, minus
```





Union: 1, 2, 3 ,4
union all: 1, 2, 3, 2, 3, 4
insertsect: 2, 3
minus: T1 - common(T1, T2) = 1

intersect: the common elements of two table



inner join, left join, right join, outer join

inner join: the common part of two tables (mid)

left join: left table + the common part

right join: the common part + remained elements of right table

outer join: all elements of two table(left + mid + right)

group by (used by aggregation function)

e.g

want to find the max salary of department

we can group by department, and Max(salary)

- it will put out all Max salary of all departments (every department has one max salary)

group by also combine with “having”

having: filter conditions (we can only use having without group by)

what is difference between having and where? (where also is a fiter condition keyword) (**Interview/Evaluation Tag**)

cross join

A: 1, 2 ,3

B: 4, 5

result of cross join:

    1, 4
    1, 5
    2, 4
    2, 5
    3, 4
    3, 5