1. SQL

1.1 事务的 ACID 属性

把多条语句作为一个整体进行操作的功能，被称为数据库事务。数据库事务可以确保该事务范围内的所有操作都可以全部成功或者全部失败。如果事务失败，那么效果就和没有执行这些 SQL 一样，不会对数据库数据有任何改动。比如一个转账操作：A 账户转账 100 给 B 账户。

一般来说，事务是必须满足 4 个条件（ACID）：：原子性（**A**tomicity，或称不可分割性）、一致性（**C**onsistency）、隔离性（**I**solation，又称独立性）、持久性（**D**urability）。

原子性：将所有 SQL 作为原子工作单元执行，要么全部执行，要么全部不执行；

一致性：事务完成后，所有数据的状态都是一致的，即 A 账户只要减去了 100，B 账户则必定加上了 100；

隔离性：如果有多个事务并发执行，每个事务作出的修改必须与其他事务隔离；

持久性：事务完成后，对数据库数据的修改被持久化存储。

对于单条 SQL 语句，数据库系统自动将其作为一个事务执行，这种事务被称为隐式事务。要手动把多条 SQL 语句作为一个事务执行，使用 BEGIN 开启一个事务，使用 COMMIT 提交一个事务，这种事务被称为显式事务.

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

COMMIT 是指提交事务，即试图把事务内的所有 SQL 所做的修改永久保存。如果 COMMIT 语句执行失败了，整个事务也会失败。有些时候，我们希望主动让事务失败，这时，可以用 ROLLBACK 回滚事务，整个事务会失败：

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
ROLLBACK;
```

1.2 四种隔离级别

事务隔离级别	脏读	不可重复读	幻读
读未提交（read-uncommitted）	是	是	是
读提交（read-committed）	-	是	是
可重复读（repeatable-read）	-	-	是
串行化（serializable）	-	-	-

Read Uncommitted（读取未提交内容）

Read Uncommitted 是隔离级别最低的一种事务级别。在这种隔离级别下，一个事务会读到另一个事务更新后但未提交的数据，如果另一个事务回滚，那么当前事务读到的数据就是脏数据，这就是**脏读（Dirty Read）**。

Read Committed（读取提交内容）

在 Read Committed 隔离级别下，一个事务可能会遇到**不可重复读（Non Repeatable Read）**的问题。不可重复读是指，在一个事务内，多次读同一数据，在这个事务还没有结束时，如果另一个事务恰好修改了这个数据，那么，在第一个事务中，两次读取的数据就可能不一致。

Repeatable Read（可重读）

在 Repeatable Read 隔离级别下，一个事务可能会遇到**幻读（Phantom Read）**的问题。幻读是指，在一个事务中，第一次查询某条记录，发现没有，但是，当试图更新这条不存在的记录时，竟然能成功，并且，再次读取同一条记录，它就神奇地出现了。可见，幻读就是没有

读到的记录，以为不存在，但其实是可以更新成功的，并且，更新成功后，再次读取，就出现了。

Serializable（可串行化）

Serializable 是最严格的隔离级别。在 Serializable 隔离级别下，所有事务按照次序依次执行，因此，脏读、不可重复读、幻读都不会出现。虽然 Serializable 隔离级别下的事务具有最高的安全性，但是，由于事务是串行执行，所以效率会大大下降，应用程序的性能会急剧降低。如果没有特别重要的情景，一般都不会使用 Serializable 隔离级别。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

1.3 数据库的各种锁的总结：

一般可以分为两类，一个是悲观锁，一个是乐观锁，悲观锁一般就是我们通常说的数据库锁机制，乐观锁一般是指用户自己实现的一种锁机制。

1.3.1 悲观锁：

悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

悲观锁按照**使用性质**划分：

共享锁（Share locks 简记为 S 锁）：也称**读锁**，事务 A 对对象 T 加 S 锁，其他事务也只能对 T 加 S，多个事务可以同时读，但不能有写操作，直到 A 释放 S 锁。

排它锁（Exclusive locks 简记为 X 锁）：也称**写锁**，事务 A 对对象 T 加 X 锁以后，其他事务不能对 T 加任何锁，只有事务 A 可以读写对象 T 直到 A 释放 X 锁。

更新锁（Update locks 简记为 U 锁）：用来预定要对此对象施加 X 锁，它允许其他事务读，但不允许再施加 U 锁或 X 锁；当被读取的对象将要被更新时，则升级为 X 锁，主要是用来防止死锁的。因为使用共享锁时，修改数据的操作分为两步，首先获得一个共享锁，读取数据，然后将共享锁升级为排它锁，然后再执行修改操作。这样如果同时有两个或多个事务同时对一个对象申请了共享锁，在修改数据的时候，这些事务都要将共享锁升级为排它锁。这些事务都不会释放共享锁而是一直等待对方释放，这样就造成了死锁。如果一个数据在修改前直接申请更新锁，在数据修改的时候再升级为排它锁，就可以避免死锁。

悲观锁按照**作用范围**划分：

行锁：锁的作用范围是行级别，数据库能够确定哪些行需要锁的情况下使用行锁，如果不知道会影响哪些行的时候就会使用表锁。

表锁：锁的作用范围是整张表。

1.3.2 乐观锁：

乐观锁（ Optimistic Locking ） 相对悲观锁而言，乐观锁假设认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做（**一般是回滚事务**）。那么我们如何实现乐观锁呢，一般来说有以下方式：

版本号（记为 version）：就是给数据增加一个版本标识，在数据库上就是表中增加一个 version 字段，每次更新把这个字段加 1，读取数据的时候把 version 读出来，更新的时候比较 version，如果还是开始读取的 version 就可以更新了，如果现在的 version 比老的 version 大，说明有其他事务更新了该数据，并增加了版本号，这时候得到一个无法更新的通知，用户自行根据这个通知来决定怎么处理，比如重新开始一遍。这里的关键是判断 version 和更新两个动作需要作为一个原子单元执行，否则在你判断可以更新以后正式更新之前有别的事务修改了 version，这个时候你再去更新就可能会覆盖前一个事务做的更新，造成第二类丢失更新。

时间戳（timestamp）：和版本号基本一样，只是通过时间戳来判断而已，注意时间戳要使用数据库服务器的时间戳不能是业务系统的时间。

待更新字段：和版本号方式相似，只是不增加额外字段，直接使用有效数据字段做版本控制信息，因为有时候我们可能无法改变旧系统的数据库表结构。假设有个待更新字段叫 count，先去读取这个 count，更新的时候去比较数据库中 count 的值是不是我期望的值（即开始读的值），

如果是就把我修改的 count 的值更新到该字段，否则更新失败。

所有字段：和待更新字段类似，只是使用所有字段做版本控制信息，只有所有字段都没变化才会执行更新

1.4 两阶段加锁协议

是指所有的事务必须分两个阶段对数据项加锁和解锁。即事务分两个阶段，第一个阶段是获得封锁。事务可以获得任何数据项上的任何类型的锁，但是不能释放；第二阶段是释放封锁，事务可以释放任何数据项上的任何类型的锁，但不能申请。

扩展阶段：封锁的阶段

在对任何数据项的读、写之前，要申请并获得该数据项的封锁。其实也就是该阶段可以进入加锁操作，在对任何数据进行读操作之前要申请获得 S 锁，在进行写操作之前要申请并获得 X 锁，加锁不成功，则事务进入等待状态，直到加锁成功才继续执行。就是加锁后就不能解锁了。

收缩阶段：释放封锁的阶段

每个事务中，所有的封锁请求必须先于解锁请求。当事务释放一个封锁后，事务进入封锁阶段，在该阶段只能进行解锁而不能再进行加锁操作。

1.5 关系模型

表的每一行称为**记录（Record）**，记录是一个逻辑意义上的数据。

表的每一列称为**字段（Column）**，同一个表的每一行记录都拥有相同的若干字段。字段定义了数据类型（整型、浮点型、字符串、日期等），以及是否允许为 NULL。注意 NULL 表示字段数据不存在。一个整型字段如果为 NULL 不表示它的值为 0，同样的，一个字符串型字段为 NULL 也不表示它的值为空串’’。

在关系数据库中，关系是通过**主键和外键**来维护的

主键：

对于关系表，有个很重要的约束，就是任意两条记录不能重复。不能重复不是指两条记录不完全相同，而是指能够通过某个字段唯一区分出不同的记录，这个字段被称为**主键**。**主键**是关系表中记录的唯一标识。主键的选取非常重要：主键不要带有业务含义，而应该使用 BIGINT 自增或者 GUID 类型。主键也不应该允许 NULL。可以使用多个列作为联合主键，但联合主键并不常用。

外键：

把数据与另一张表关联起来，这种列称为**外键**。关系数据库通过外键可以实现一对多、多对多和一对一的关系。外键既可以通过数据库来约束，也可以不设置约束，仅依靠应用程序的逻辑来保证。

索引：

索引是关系数据库中对某一列或多个列的值进行预排序的数据结构。通过使用索引，可以让数据库系统不必扫描整个表，而是直接定位到符合条件的记录，这样就大大加快了查询速度。

1.6 索引结构(B-树/B+树，哈希，空间，全文)

1.6.1 B-Tree 索引

B-Tree 索引是大多数 MySQL 存储引擎的默认索引类型。

B-树

是一种多路搜索树（并不是二叉的）：

1.定义任意非叶子结点最多只有 M 个儿子；且 M>2；

2.根结点的儿子数为[2, M]；

3.除根结点以外的非叶子结点的儿子数为[M/2, M]；

4.每个结点存放至少 M/2-1（取上整）和至多 M-1 个关键字；（至少 2 个关键字）

5.非叶子结点的关键字个数=指向儿子的指针个数-1；

6.非叶子结点的关键字：K[1], K[2], …, K[M-1]；且 K[i] < K[i+1]；

7.非叶子结点的指针：P[1], P[2], …, P[M]；其中 P[1]指向关键字小于 K[1]的
子树，P[M]指向关键字大于 K[M-1]的子树，其它 P[i]指向关键字属于(K[i-1], K[i])的子树；

8.所有叶子结点位于同一层；

B+树

B+树是 B-树的变体，也是一种多路搜索树：

1.其定义基本与 B-树同，除了：

2.非叶子结点的子树指针与关键字个数相同；

3.非叶子结点的子树指针 P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；

5.为所有叶子结点增加一个链指针；

6.所有关键字都在叶子结点出现；

1.6.2 哈希索引

基于哈希表实现，优点是查找非常快。

在 MySQL 中只有 Memory 引擎显式支持哈希索引。

哈希索引只包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行。

1.6.3. 空间索引（R-Tree）

MyISAM 存储引擎支持空间索引，可以用于地理数据存储。

空间索引会从所有维度来索引数据，可以有效地使用任意维度来进行组合查询。

1.6.4 全文索引

MyISAM 存储引擎支持全文索引，用于查找文本中的关键词，而不是直接比较索引中的值。

1.7 索引分类(聚簇索引和 非聚簇索引)

单值索引

即一个索引只包含单个列，一个表可以有多个单列索引。

唯一索引

索引列的值必须唯一，但允许有空值。

复合索引

即一个索引包含多个列。

1.8 SQL 索引优化

（1）独立的列

在进行查询时，索引列不能是表达式的一部分，也不能是函数的参数，否则无法使用索引例如下面的查询不能使用 actor_id 列的索引：

```
SELECT actor_id FROM sakila.actor WHERE actor_id + 1 = 5;
```

（2） 前缀索引

对于 BLOB、TEXT 和 VARCHAR 类型的列，必须使用前缀索引，只索引开始的部分字符。对于前缀长度的选取需要根据 索引选择性 来确定：不重复的索引值和记录总数的比值。选择性越高，查询效率也越高。最大值为 1 ，此时每个记录都有唯一的索引与其对应。

（3） 多列索引

在需要使用多个列作为条件进行查询时，使用多列索引比使用多个单列索引性能更好。例如下面的语句中，最好把 actor_id 和 file_id 设置为多列索引。

```
SELECT file_id, actor_ id FROM sakila.film_actor WhERE actor_id = 1 OR film_id = 1;
```

（4）索引列的顺序

让选择性最强的索引列放在前面。

（5） 聚簇索引

聚簇索引并不是一种索引类型，而是一种数据存储方式。

术语“聚簇”表示数据行和相邻的键值紧密地存储在一起，InnoDB 的聚簇索引的数据行存放在 B-Tree 的叶子页中。因为无法把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引。

优点：可以把相关数据保存在一起，减少 I/O 操作；因为数据保存在 B-Tree 中，因此数据访问更快。

缺点：聚簇索引最大限度提高了 I/O 密集型应用的性能，但是如果数据全部放在内存，就没必要用聚簇索引。插入速度严重依赖于插入顺序，按主键的顺序插入是最快的。更新操作代价很高，因为每个被更新的行都会移动到新的位置。当插入到某个已满的页中，存储引擎会将该页分裂成两个页面来容纳该行，页分裂会导致表占用更多的磁盘空间。如果行比较稀疏，或者由于页分裂导致数据存储不连续时，聚簇索引可能导致全表扫描速度变慢。

（6）覆盖索引

索引包含所有需要查询的字段的值。

1.9 哪些情况需要创建索引

①主键自动建立唯一索引
②频繁作为查询条件的字段应该创建索引
③查询中与其他表关联的字段，外键关系建立索引
④频繁更新的字段不适合建立索引，因为每次更新不单单是更新了记录还会更新索引
⑤WHERE 条件里用不到的字段不创建索引
⑥单键/组合索引的选择问题，who?(在高并发下倾向创建组合索引)
⑦查询中排序的字段，排序的字段若通过索引去访问将大大提高排序速度
⑧查询中统计或者分组字段

1.10 哪些情况不需要创建索引

①表记录太少
②经常增删改的表
③注意，如果某个数据列包含许多重复的内容，为它建立索引就没有太大的实际效果。

1.11 查询性能优化(Explain 及其它)

1. Explain

用来分析 SQL 语句，分析结果中比较重要的字段有：

- select_type : 查询类型，有简单查询、联合查询和子查询
- key : 使用的索引
- rows : 扫描的行数

2. 减少返回的列

慢查询主要是因为访问了过多数据，除了访问过多行之外，也包括访问过多列。

最好不要使用 SELECT * 语句，要根据需要选择查询的列。

3. 减少返回的行

最好使用 LIMIT 语句来取出想要的那些行。

还可以建立索引来减少条件语句的全表扫描。例如对于下面的语句，不适用索引的情况下需要进行全表扫描，而使用索引只需要扫描几行记录即可，使用 Explain 语句可以通过观察 rows 字段来看出这种差异。

```
SELECT * FROM sakila.film_actor WHERE film_id = 1;
```

4. 拆分大的 DELETE 或 INSERT 语句

如果一次性执行的话，可能一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。

```
DELEFT FROM messages WHERE create < DATE_SUB(NOW(), INTERVAL 3 MONTH);
rows_affected = 0 do { rows_affected = do_query( "DELETE FROM messages WHERE create < DATE_SUB(NOW(), INTERVAL 3 MONTH) LIMIT 10000") } while rows_affected > 0
```

1.12 分库与分表

1. 分表与分区的不同

分表，就是讲一张表分成多个小表，这些小表拥有不同的表名；而分区是将一张表的数据分为多个区块，这些区块可以存储在同一个磁盘上，也可以存储在不同的磁盘上，这种方式下表仍然只有一个。

2. 使用分库与分表的原因

随着时间和业务的发展，数据库中的表会越来越多，并且表中的数据量也会越来越大，那么读写操作的开销也会随着增大。

3. 垂直切分

将表按功能模块、关系密切程度划分出来，部署到不同的库上。例如，我们会建立商品数据库 payDB、用户数据库 userDB 等，分别用来存储项目与商品有关的表和与用户有关的表。

4. 水平切分

把表中的数据按照某种规则存储到多个结构相同的表中，例如按 id 的散列值、性别等进行划分，

5. 垂直切分与水平切分的选择

如果数据库中的表太多，并且项目各项业务逻辑清晰，那么垂直切分是首选。

如果数据库的表不多，但是单表的数据量很大，应该选择水平切分。

6. 水平切分的实现方式

最简单的是使用 merge 存储引擎。

7. 分库与分表存在的问题

(1) 事务问题

在执行分库分表之后，由于数据存储到了不同的库上，数据库事务管理出现了困难。如果依赖数据库本身的分布式事务管理功能去执行事务，将付出高昂的性能代价；如果由应用程序去协助控制，形成程序逻辑上的事务，又会造成编程方面的负担。

(2) 跨库跨表连接问题

在执行了分库分表之后，难以避免会将原本逻辑关联性很强的数据划分到不同的表、不同的库上。这时，表的连接操作将受到限制，我们无法连接位于不同分库的表，也无法连接分表粒度不同的表，导致原本只需要一次查询就能够完成的业务需要进行多次才能完成。

1.13 SQL 与 NoSQL 的区别

存储方式： SQL 数据存在特定结构的表中；而 NoSQL 则更加灵活和可扩展，存储方式可以省是 JSON 文档、哈希表或者其他方式。SQL 通常以数据库表形式存储数据。

表/数据集合的数据的关系： 在 SQL 中，必须定义好表和字段结构后才能添加数据，例如定义表的主键(primary key)，索引(index),触发器(trigger),存储过程(stored procedure)等。表结构可以在被定义之后更新，但是如果有比较大的结构变更的话就会变得比较复杂。在 NoSQL 中，数据可以在任何时候任何地方添加，不需要先定义表。 NoSQL 可能更加适合初始化数据还不明确或者未定的项目中。

SQL 提供了 Join 查询，可以将多个关系数据表中的数据用一条查询语句查询出来。NoSQL 没有提供。

SQL 删数据为了数据完整性，比如外键，不能随便删，而 NoSQL 是可以随意删除的。

在处理非结构化/半结构化的大数据时；在水平方向上进行扩展时；随时应对动态增加的数据项时可以优先考虑使用 NoSQL 数据库。

NoSQL 缺点：事务支持较弱，join 等复杂操作能力较弱，通用性较差

1.14 SQL 的增删改查

关系数据库的基本操作就是增删改查，即 CRUD：Create、Retrieve、Update、Delete。

对于增、删、改、查，对应的 SQL 语句分别是：

INSERT：插入新记录；

DELETE：删除已有记录；

UPDATE：更新已有记录；

SELECT：查询已有记录。

1.14.1 INSERT

使用 INSERT，我们就可以一次向一个表中插入一条或多条记录。

```
INSERT INTO <表名> (字段1, 字段2, ...) VALUES (值1, 值2, ...);
```

```
INSERT INTO students (class_id, name, gender, score) VALUES (2, '大牛', 'M', 80);
```

```
INSERT INTO students (class_id, name, gender, score) VALUES
  (1, '大宝', 'M', 87),
  (2, '二宝', 'M', 81);
```

1.14.2 DELETE

使用 DELETE，我们就可以一次删除表中的一条或多条记录。

```
DELETE FROM <表名> WHERE ...;
```

```
DELETE FROM students WHERE id=1;
```

```
DELETE FROM students WHERE id>=5 AND id<=7;
```

不带 WHERE 条件的 DELETE 语句会删除整个表的数据

1.14.3 UPDATE

使用 UPDATE，我们就可以一次更新表中的一条或多条记录。

```
UPDATE <表名> SET 字段1=值1, 字段2=值2, ... WHERE ...;
```

```
UPDATE students SET name='大牛', score=66 WHERE id=1;
```

```
UPDATE students SET name='小牛', score=77 WHERE id>=5 AND id<=7;
```

UPDATE 语句可以没有 WHERE 条件，这时，整个表的所有记录都会被更新

1.14.4 SELECT

(1)基本查询

使用 SELECT 查询的基本语句可以查询一个表的所有行和所有列的数据。

```
SELECT * FROM <表名>
```

SELECT 查询的结果是一个二维表。

(2)条件查询

很多时候，我们并不希望获得所有记录，而是根据条件选择性地获取指定条件的记录，通过 WHERE 条件查询，可以筛选出符合指定条件的记录，而不是整个表的所有记录。

```
SELECT * FROM <表名> WHERE <条件表达式>
```

条件	表达式举例1	表达式举例2	说明
使用=判断相等	score = 80	name = 'abc'	字符串需要用单引号括起来
使用>判断大于	score > 80	name > 'abc'	字符串比较根据ASCII码，中文字符比较根据数据库设置
使用>=判断大于或相等	score >= 80	name >= 'abc'	
使用<判断小于	score < 80	name <= 'abc'	
使用<=判断小于或相等	score <= 80	name <= 'abc'	
使用<>判断不相等	score <> 80	name <> 'abc'	
使用LIKE判断相似	name LIKE 'ab%'	name LIKE '%bc%'	%表示任意字符，例如'ab%'将匹配'ab'，'abc'，'abcd'

(3)投影查询

如果我们只希望返回某些列的数据，而不是所有列的数据，我们可以使用 SELECT *表示查询表的所有列，使用 SELECT 列 1, 列 2, 列 3 则可以仅返回指定列，这种操作称为投影。

SELECT 语句可以对结果集的列进行重命名。

```
SELECT 列 1, 列 2, 列 3 FROM ...
```

(4)排序

使用 SELECT 查询时，细心的读者可能注意到，查询结果集通常是按照 id 排序的，也就是根据主键排序。如果要根据其他条件排序可以加上 ORDER BY 子句。例如按照成绩从低到高进行排序：

```
SELECT id, name, gender, score FROM students ORDER BY score;
```

如果要反过来，按照成绩从高到底排序，我们可以加上 DESC 表示“倒序”：

```
SELECT id, name, gender, score FROM students ORDER BY score DESC;
```

如果 score 列有相同的数据，要进一步排序，可以继续添加列名。例如，使用 ORDER BY score DESC, gender 表示先按 score 列倒序，如果有相同分数的，再按 gender 列排序：

```
SELECT id, name, gender, score FROM students ORDER BY score DESC, gender;
```

(5)分页查询

分页实际上就是从结果集中“截取”出第 M~N 条记录。这个查询可以通过 LIMIT <M> OFFSET <N>子句实现，分页查询需要先确定每页的数量和当前页数，然后确定 LIMIT 和 OFFSET 的值。

```
SELECT id, name, gender, score
FROM students
ORDER BY score DESC
LIMIT 3 OFFSET 0;
```

(5)聚合查询

使用 SQL 提供的聚合查询，可以方便地计算总数、合计值、平均值、最大值和最小值；除了 COUNT()函数外，SQL 还提供了如下聚合函数：

函数	说明
SUM	计算某一列的合计值，该列必须为数值类型
AVG	计算某一列的平均值，该列必须为数值类型
MAX	计算某一列的最大值
MIN	计算某一列的最小值

(5)多表查询

SELECT 查询不但可以从一张表查询数据，还可以从多张表同时查询数据。查询多张表的语法是：SELECT * FROM <表 1> <表 2>

(5)连接查询

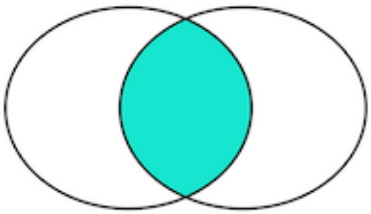
连接查询是另一种类型的多表查询。连接查询对多个表进行 JOIN 运算，简单地说，就是先确定一个主表作为结果集，然后，把其他表的行有选择性地“连接”在主表结果集上。

INNER JOIN 是最常用的一种 JOIN 查询，它的语法是 SELECT ... FROM <表 1> INNER JOIN

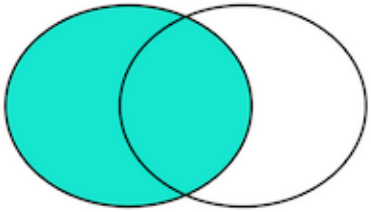
〈表 2〉 ON 〈条件...〉;

```
SELECT ... FROM tableA ??? JOIN tableB ON tableA.column1 = tableB.column2;
```

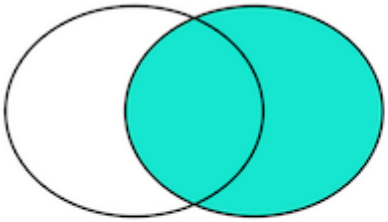
我们把tableA看作左表，把tableB看成右表，那么INNER JOIN是选出两张表都存在的记录：



LEFT OUTER JOIN是选出左表存在的记录：



RIGHT OUTER JOIN是选出右表存在的记录：



FULL OUTER JOIN则是选出左右表都存在的记录：