- **数据库**
  数据库：物理操作文件系统或其他形式文件类型的集合

- **过程存储**: 在大型数据库系统中，一组为了完成特定功能的 **SQL 语句集**，存储在数据库中，**经过第一次编译后再次调用不需要再次编译**，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。存储过程是数据库中的一个重要对象。
  优点
  1. 存储过程是一个编译过的代码块，执行效率比执行一堆T-SQL语句较高
  2. 可以重复使用，减少开发人员的工作量
  3. 存储过程位于服务器上，调用时只需要传递存储过程的名称和参数，降低网络传输的数据量，提高通信速率
  4. 使不够权限的用户能够在控制下间接的存取数据库，确保数据安全。防止SQL注入攻击

- **SQL注入**: 把sql语句插入到web表单或者请求域名、页面等字符串中，从而达到欺骗服务器执行恶意SQL语句。
  预防
  1. 不要完全信任用户的输入，对用户输入进行校验，通过正则化之类的方法
  2. 不要用sql拼接
  3. 对于重要的信息不要直接存取，加入hash加密之类的
  4. 不要用管理员权限直接操作数据库，对于每个应用有它自己的权限
  5. 用一些辅助软件或者网络平台来检测SQL注入

- **索引**:
  索引就是根据数据库表中的一列或者几列的值进行排序的结构，提高mysql检索效率的数据结构。类比：书籍或者图书馆的检索索引
  为什么需要索引：
  索引文件本身也很大，不可能全部存储在内存中，因此索引一般是以索引文件的形式存储在磁盘上。因此索引在查找的时候会有磁盘IO消耗，相对于内存存取，IO存取的消耗要高几个数量级
  评价索引的优劣：
  在查找过程中磁盘IO操作次数的渐进复杂度。尽量减少磁盘IO的存取次数。
  索引记录=键值（定义索引是指定的字段的值）+逻辑指针（指向数据页或者下一个索引页）
  B-TREE增加、删除节点实例分析
  https://www.2cto.com/database/201411/351106.html
  优点：
  1. 提高数据检索的效率，降低数据库的IO成本
  2. 利用索引来排序，降低CPU消耗
  缺点：
  1. 数据库表插入、删除更新的时候，速度降低，因为需要更新保存索引表
  2. 索引文件占用一定的磁盘空间
  3. 索引不一定能够提高查询的性能：索引需要空间存储，并且需要定期维护，当记录在增减时，索引本身也会被修改，每次插入、删除、更新会多付出4、5次的磁盘IO。如果使用不必要的索引反而会使查询时间变慢。
  B-Tree特性
  B-Tree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；

类型	关键字分布	搜索结束	搜索性能	其他
B树	整棵树	可能在非叶子节点结束	二分查找
B+树	叶子节点	一定在叶子节点结束	二分查找	更适合文件索引系统；每个结点的指针上限为2d而不是2d+1。内结点不存储data，

只存储key；叶子结点不存储指针。

为什么不用红黑树而用了B-TREE
在B-TREE中，磁盘按需读取，每次都会预读的长度一般为页的整数倍，并且数据库把一个节点的大小设置为一个页，每个节点只需要一次IO就可以完全载入。B-tree的m值设置为很大，那么让树的高度降低，有利于一次完全载入。在红黑树结构中，逻辑相邻的节点物理上不一定相邻，也就是说读取同等的数据红黑树需要多次I/O。所以选择B-树效率更好。
红黑树结构，h明显要深得多。由于逻辑上很近的结点（父子结点）物理上可能离得很远，无法利用局部性原理。所以即使红黑树的I/O渐进复杂度也为O(h)

那为何最终选了B+树呢？
因为B+树内节点去掉了data域，因此可以拥有更大的出度，就是说一个结点可以存储更多的内结点，那么I/O效率更高。

MYSQL如何实现索引：
索引是存储引擎级别的概念，不同的存储引擎实现索引的方式不同。

存储引擎	索引结构	索引类型	叶节点的data域
MyISAM	B+ Tree	非聚集索引，指B+Tree的叶子节点上的data，并不是数据本身，而是数据存放的地址	存放数据记录的地址
InnoDB	B+ Tree	聚集索引，就是指主索引文件和数据文件为同一份文件	存储相应主键的值而不是地址

区别：
非聚簇索引比聚簇索引多了一次读取数据的IO操作，所以查找性能上会差。
第一个重大区别是：InnoDB的数据文件本身就是索引文件。
第二个与MyISAM索引的不同是：InnoDB的辅助索引data域存储相应记录主键的值而不是地址。聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

- **mysql的索引引擎**
  myisam是属于非聚集索引，data域存放的是数据记录的地址，因此还要多一次读取数据的io操作，因此在查找性能上会差。
  索引与数据分开
  innodb是属于聚集索引，支持事务、行级锁和外键约束等比较高级的数据库功能。一般用于大数据级的数据库。是默认的引擎。对于大量的读操作的话效率比较低。
  索引文件与数据文件是同一个文件。索引与数据合并
  共同点：
  底层都是b+树
  区别：

  i. 是否支持事务，行级锁，外键的约束
  ii. innodb的索引文件就是数据文件，而myisam的叶子data域存储的是数据的地址，还需要多一次io操作去读取数据。
  使用场景：
  要支持事务、行级锁和外键约束等时，insert\update操作比较多的时候，不适合select，会用到innodb
  如果是大量需要select查询操作的需求，支持全文搜索，可以选择myisam，更强调性能

- **聚集索引和辅助索引**
  数据库中的 B+ 树索引可以分为聚集索引（clustered index）和辅助索引（secondary index），它们之间的最大区别就是，聚集索引中存放着一条行记录的全部信息，而辅助索引中只包含索引列和一个用于查找对应行记录的『书签』。
  聚集索引叶节点中保存的是整条行记录，而不是其中的一部分。当我们使用聚集索引对表中的数据进行检索时，可以直接获得聚集索引所对应的整条行记录数据所在的页，不需要进行第二次操作。
  辅助索引，也叫非聚集索引，辅助索引也是通过 B+ 树实现的，但是它的叶节点并不包含行记录的全部数据，仅包含索引中的所有键和一个用于查找对应行记录的『书签』，在 InnoDB 中这个书签就是当前记录的主键。辅助索引只用于加速数据的查找，所以一张表上往往有多个辅助索引以此来提升数据库的性能。就是说你得到一个[书签]之后，在通过聚集索引获得整条行记录。

- **事务：** 事务是一个序列操作，事务里的操作要么执行，要么不执行。是维护数据库一致性的单位。

四个属性：ACID，原子性，一致性，隔离性，持久性

- 原子性：一个事务中的所有操作要么执行，要么不执行，如果中途某个操作执行失败了，就会回滚到事务开始的状态
- 一致性：在事务的开始和结束后，数据库的完整性没有被破坏。
- 隔离性：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
- 持久性：处理成功事务后，对数据库的修改是持久的。

锁：在所有的 DBMS 中，锁是实现事务的关键，锁可以保证事务的完整性和并发性。与现实生活中锁一样，它可以使某些数据的拥有者，在某段时间内不能使用某些数据或数据结构。当然锁还分级别的。

如何实现：
原子性：要保证原子性，就需要在异常发生的时候能够对已经执行的操作进行回滚，在mysql中通过 **undo log（回滚日志）** 实现,所有的事务进行的修改都会放在这个回滚日志中，然后对数据库中对应行进行写入。它还能够在整个系统发生崩溃、数据库进程直接被杀死后，当用户再次启动数据库进程时，还能够立刻通过查询回滚日志将之前未完成的事务进行回滚，这也就需要回滚日志必须先于数据持久化到磁盘上，是我们需要先写日志后写数据库的主要原因。回滚日志并不能将数据库物理地恢复到执行语句或者事务之前的样子。它只会按照日志逻辑地将数据库中的修改撤销掉看，可以理解为，我们在事务中使用的每一条 INSERT 都对应了一条 DELETE，每一条 UPDATE 也都对应一条相反的 UPDATE 语句。
注意：外部的输出是无法回滚的。

事务的状态由三种，active正在执行，commited已经提交，failed失败。可能会有中间状态，比如partially commited最后一条语句执行之后， aborted事务被回滚后并且数据库恢复到事务进行之前的状态之后
并行性的原子性实现：如果tranction2依赖于transaction1，而transaction3也依赖于transction1，而transaction1由于执行出现了问题需要回滚，那么transaction2和transaction3也要进行回滚，叫做级联回滚。

持久性：事务的持久性就体现在，一旦事务被提交，那么数据一定会被写入到数据库中并持久存储起来。事务的持久性是通过日志来实现，mysql用的是**redo log**(重做日志)来实现持久性。由两个部分组成，一个是内存中的重做日志缓存，因为重做日志缓存区在内存里面，是易丢失的；另一个是在磁盘上的重做日志文件，它是持久的。
当我们在一个事务中尝试对数据进行修改时，它会先将数据从磁盘读入内存，并更新内存中缓存的数据，然后生成一条重做日志并写入重做日志缓存，当事务真正提交时，MySQL 会将重做日志缓存中的内容刷新到重做日志文件，再将内存中的数据更新到磁盘上

隔离性
事务的隔离级别：
read uncommited：查询的时候不加锁，可能会读到未提交的行(dirty read)
read commited：只对记录加记录锁，而不会在记录之间加间隙锁，所以允许新的记录插入到被锁定记录的附近，所以再多次使用查询语句时，可能得到不同的结果（Non-Repeatable Read）；
repeateable read：多次读取同一范围的数据会返回第一次查询的快照，不会返回不同的数据行，但是可能发生幻读（Phantom Read）【mysql作为默认设置】
serializable：InnoDB 隐式地将全部的查询语句加上共享锁，解决了幻读的问题；
随着隔离的级别越来越严格，数据库对于并发执行事务的性能也逐渐下降。

隔离的实现是通过锁、时间戳来实现的。不会锁住整个数据库，而是锁住那些要访问的数据项。锁的话分为两种：共享锁(读锁)和互斥锁(写锁）

一致性
如果一个事务原子地在一个一致地数据库中独立运行，那么在它执行之后，数据库的状态一定是一致的。对事务的要求不止包含对数据完整性以及合法性的检查，还包含应用层面逻辑的正确。
CAP 定理中的数据一致性，其实是说分布式系统中的各个节点中对于同一数据的拷贝有着相同的值。

- **innoDB对数据的存储**
  在 InnoDB 存储引擎中，所有的数据都被逻辑地存放在表空间中，表空间（tablespace）是存储引擎中最高的存储逻辑单位，在表空间的下面又包括段（segment）、区（extent）、页（page）

- **存储大对象**
  用varchar或者blob，我们并不会直接将所有的内容都存放在数据页节点中，而是将行数据中的前 768 个字节存储在数据页

中，后面会通过偏移量指向溢出页。

- **脏读**
  一个事务读取了其他事务未提交的数据

- **不可重复读**
  在一个事务中，同一行记录被访问了两次却得到了不同的结果。
  不可重复读的原因就是，在 READ COMMITED 的隔离级别下，存储引擎不会在查询记录时添加行锁，锁定 id = 3 这条记录。

- **幻读**
  在一个事务中，同一个范围内的记录被读取时，其他事务向这个范围添加了新的记录
  在标准的事务隔离级别中，幻读是由更高的隔离级别 SERIALIZABLE 解决的，但是它也可以通过 MySQL 提供的 Next-Key 锁解决：

REPEATABLE READ 和 READ UNCOMMITED 其实是矛盾的，如果保证了前者就看不到已经提交的事务，如果保证了后者，就会导致两次查询的结果不同，MySQL 为我们提供了一种折中的方式，能够在 REPEATABLE READ 模式下加锁访问已经提交的数据，其本身并不能解决幻读的问题，而是通过文章前面提到的 Next-Key 锁来解决。

- **事务日志**
  分为两种：回滚日志(undo log,用于原子性)与重做日志(redo log，用于持久性)
  发生错误或者需要回滚的事务能够成功回滚（原子性）；
  在事务提交后，数据没来得及写会磁盘就宕机时，在下次重新启动后能够成功恢复数据（持久性）；

- **drop,delete, truncate的区别**
  truncate、delete用于删除表中的数据，但是不删除表，但delete可以配合where来使用，删除部分记录
  truncate、drop是一次性从表中删除所有的数据并且不会把单独的操作记录记入到日志中，删除是不可以恢复的，并且不会激活与表相关的删除触发器。truncate会将表和索引占用的空间恢复到初始大小。truncate只能对表进行操作。
  delete语句每次从表中删除一行，然后同时将该行的操作记录作为事务记录在日志中保存以便进行回滚操作。delete不会减少表和索引占用的空间。delete可以对表和视图。
  drop直接删除表
  速度：drop>truncate>delete(delete每次删除一行，所以比较慢)

DML(DATA MAINTAIN LANGUAGE):操作先放到rollback segment，事务提交后才生效。如果有相应的trigger，会激活。可以回滚。
DDL(DATA DEFINE LANGUAGE)：操作立即执行，原数据不会放到rollback segment中，不能回滚。

1. 在速度上，一般来说，drop> truncate > delete。
2. 在使用drop和truncate时一定要注意，虽然可以恢复，但为了减少麻烦，还是要慎重。
3. 如果想删除部分数据用delete，注意带上where子句，回滚段要足够大；
   如果想删除表，当然用drop；
   如果想保留表而将所有数据删除，如果和事务无关，用truncate即可；
   如果和事务有关，或者想触发trigger，还是用delete；
   如果是整理表内部的碎片，可以用truncate跟上reuse stroage，再重新导入/插入数据

- **超键、主键、候选键、外键**

  - 超键：在关系中**能唯一标识元组的属性集**称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。
  - 候选键：是 **最小超键**，即没有冗余元素的超键。
  - 主键：**数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。** 一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。
  - 外键：在 **一个表中存在的另一个表的主键** 称此表的外键。
    举个例子：
    身份证、学号、姓名、年龄、系别
    <身份证,姓名>可以作为超键，但是不能作为候选键，因为去掉姓名仍然能够唯一标识该元祖，也就是不是最小超键，所以不是候选键。

- **视图**

视图是一种虚拟的表，能够对视图进行增删改查的操作并且不影响基本表。使得我们获取数据更容易。

- 只暴露部分字段给访问者
- 查询的数据来源于不同的表，查询者想要用统一的方式查询，这时可以建立一个视图。查询者直接从视图中获取数据，不需要考虑数据来源于不同表所带来的差异。

- **三个范式**
  范式越高，数据的冗余度越小。其实没有冗余的数据库设计是可以做到的。但是，没有冗余的数据库未必是最好的数据库，有时为了提高运行效率，就必须降低范式标准，适当保留冗余数据。降低范式就是增加字段，允许冗余。

  - 1NF：属性不可分
  - 2NF：符合1NF，表必须有一个主键；没有包含在主键中的列必须完全依赖于主键，而不能只依赖于主键的一部分。
    比如说主键为,那么非主键C一定是->C;如果A->C,则不满足二范式。
  - 3NF：符合2NF，并且，消除传递依赖。
    考虑一个订单表[Order]（OrderID，OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity）主键是（OrderID）。
    其中 OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity 等非主键列都完全依赖于主键（OrderID），所以符合 2NF。不过问题是 CustomerName，CustomerAddr，CustomerCity 直接依赖的是 CustomerID（非主键列），而不是直接依赖于主键，它是通过传递才依赖于主键，所以不符合 3NF。通过拆分[Order]为[Order]（OrderID，OrderDate，CustomerID）和[Customer]（CustomerID，CustomerName，CustomerAddr，CustomerCity）从而达到 3NF。
  - BCNF：符合3NF，并且，主属性不依赖于主属性。
    举个例子：
    码：(管理员、物品名),(仓库名、物品名)
    主属性：管理员、物品名、仓库名
    非主属性：数量
    这里属于3NF，但仍然存在插入异常（如果增加一个仓库，不能指派管理员，因为还没有物品），删除异常（物品被请空之后，仓库和管理员信息也被删除了）和更新异常（更新一个仓库的管理员名字，需要修改多条数据）的问题
    在 3NF 的基础上消除主属性对于码的部分与传递函数依赖
    表1：（仓库名）->(管理员)
    表2：（仓库名）->(物品名，数量)

- **异常**
  插入异常，删除异常，修改异常
  表中的一行就是一个元组。

- **数据库五大约束**

1. primary KEY:设置主键约束；
2. UNIQUE：设置唯一性约束，不能有重复值；
3. DEFAULT 默认值约束，height DOUBLE(3,2)DEFAULT 1.2 height不输入是默认为1,2
4. NOT NULL：设置非空约束，该字段不能为空；
5. FOREIGN KEY :设置外键约束。

- **触发器的作用**
  触发器是一种特殊的存储过程，主要是通过事件来触发而被执行的。它可以强化约束，来维护数据的完整性和一致性，可以跟踪数据库内的操作从而不允许未经许可的更新和变化。可以联级运算。如，某表上的触发器上包含对另一个表的数据操作，而该操作又会导致该表触发器被触发。

- **锁**
  锁分为两种：乐观锁和悲观锁，innoDB用的是悲观锁，按照锁的粒度又分为行锁和表锁。

  - 乐观锁与悲观锁 - 实现并发控制机制
    乐观锁是一种思想，它其实并不是一种真正的『锁』，它会先尝试对资源进行修改，在写回时判断资源是否进行了改变，如果没有发生改变就会写回，否则就会进行重试，在整个的执行过程中其实都没有对数据库进行加锁；
    悲观锁就是一种真正的锁了，它会在获取资源前对资源进行加锁，确保同一时刻只有有限的线程能够访问该资源，其他想要尝试获取资源的操作都会进入等待状态，直到该线程完成了对资源的操作并且释放了锁后，其他线程才能重新操作

资源；
使用范围：乐观锁：需要很高的响应频率和并发量超级大的时候；悲观锁：冲突频率和重试成本比较高的时候

- 行级锁：共享锁和互斥锁
  共享锁：允许事务对一条行数据进行读取。共享锁是兼容的，可以并行读
  互斥锁：允许事务对一条行数据进行删除和更新，互斥锁是不兼容的。只能串行写
- 不同粒度的锁：行锁和表锁
- 锁的算法
  - record lock 记录锁：加到索引记录上的锁
  - gap lock 间隙锁：对索引记录中的一段连续区域的锁
  - next-key lock：记录锁和记录前的间隙锁的结合。Next-Key 锁锁定的是当前值和前面的范围。

一个表一千个列值为true和false，写sql 查询 有300个列值为true的行。
数据库索引、设计、范式、引擎、索引为啥用的是B+树不用别的数据结构、时间复杂度、利用索引查询一条记录过程（分别从查询B+树、数据页、链表、聚簇索引等等角度来阐述）
左连接
数据库隔离级别，每个级别会引发什么问题，mysql默认是哪个级别
MYSQL的两种存储引擎区别（事务、锁级别等等），各自的适用场景
数据库的优化（从sql语句优化和索引两个部分回答）
索引有B+索引和hash索引，各自的区别
B+索引数据结构，和B树的区别
索引的分类（主键索引、唯一索引），最左前缀原则，哪些情况索引会失效
聚集索引和非聚集索引区别。
有哪些锁（乐观锁悲观锁），select时怎么加排它锁
关系型数据库和非关系型数据库区别
了解nosql
数据库三范式，根据某个场景设计数据表（可以通过手绘ER图）
数据库的主从复制
使用explain优化sql和索引
long_query怎么解决
内连接、外连接、交叉连接、笛卡儿积等
深入
MVCC机制
根据具体场景，说明版本控制机制
死锁怎么解决
varchar和char的使用场景。
mysql并发情况下怎么解决（通过事务、隔离级别、锁）

- **MySQL的union all和union有什么区别**
  UNION和UNION ALL关键字都是将两个结果集合并为一个，但这两者从使用和效率上来说都有所不同。
  1、对重复结果的处理：UNION在进行表链接后会筛选掉重复的记录，Union All不会去除重复记录。
  2、对排序的处理：Union将会按照字段的顺序进行排序；UNION ALL只是简单的将两个结果合并后就返回。
  从效率上说，UNION ALL 要比UNION快很多，所以，如果可以确认合并的两个结果集中不包含重复数据且不需要排序时的话，那么就使用UNION ALL。

- **MySQL有哪几种join方式，底层原理是什么**
  Left join
  Right join
  Inner join
  Netsted loop join(两层循环嵌套，性能取决于内层循环，外循环是驱动表，内循环是被驱动表)
  Block nested loop join（用一个缓冲区，从驱动表中读取多条数据，被驱动表中的每条数据尝试与之匹配，如果缓冲区越大，就能一次取出更多的记录，减少内循环的次数）
  Nested Loop Join（NLJ）算法：
  首先介绍一种基础算法:NLJ，嵌套循环算法。循环外层是驱动表，循坏内层是被驱动表。驱动表会驱动被驱动表进行连接操作。首先驱动表找到第一条记录，然后从头扫描被驱动表，逐一查找与驱动表第一条记录匹配的记录然后连接起来形成结果表中的一条记。被驱动表查找完后，再从驱动表中取出第二个记录，然后从头扫描被驱动表，逐一查找与驱动表第二条记录匹配的记录，连接起来形成结果表中的一条记录。重复上述操作，直到驱动表的全部记录都处理完毕为止。这就是嵌套循环连接算法的基本思想，伪代码如下。

```
foreach row1 from t1
    foreach row2 from t2
        if row2 match row1 //row2与row1匹配，满足连接条件
            join row1 and row2 into result //连接row1和row2加入结果集
```

首先加载t1，然后从t1中取出第一条记录，之后加载t2表，与t2表中的记录逐个匹配，连接匹配的记录。
Block Nested Loop Join(BNLJ)算法：
再介绍一种高级算法：BNLJ，块嵌套循环算法，可以看作对NLJ的优化。大致思想就是建立一个缓存区，一次从驱动表中取多条记录，然后扫描被驱动表，被驱动表的每一条记录都尝试与缓冲区中的多条记录匹配，如果匹配则连接并加入结果集。缓冲区越大，驱动表一次取出的记录就越多。这个算法的优化思路就是减少内循环的次数从而提高表连接效率。
影响性能的因素
1. 内循环的次数：现在考虑这么一个场景，当t1有100条记录，t2有10000条记录。那么，t1驱动t2与t2驱动t1，他们之间在效率上孰优孰劣？如果是单纯的分析指令执行次数，他们都是100*10000,但是考虑到加载表的次数呢。首先分析t1驱动t2，t1表加载1次，t2表需要加载100次。然后分析t2驱动t1，t2表首先加载1次，但是t1表要加载10000次。所以，t1驱动t2的效率要优于t2驱动t1的效率。由此得出，小表驱动大表能够减少内循环的次数从而提高连接效率。
另外，如果使用Block Nested Loop Join算法的话，通过扩大一次缓存区的大小也能减小内循环的次数。由此又可得，设置合理的缓冲区大小能够提高连接效率
2. 快速匹配：扫描被驱动表寻找合适的记录可以看做一个查询操作，如何提高查询的效率呢？建索引啊！由此还可得出，在被驱动表建立索引能够提高连接效率
3. 排序：假设t1表驱动t2表进行连接操作，连接条件是t1.id=t2.id，而且要求查询结果对id排序。现在有两种选择，方式一[…ORDER BY t1.id]，方式二[…ORDER BY t2.id]。如果我们使用方式一的话，可以先对t1进行排序然后执行表连接算法，如果我们使用方式二的话，只能在执行表连接算法后，对结果集进行排序（Using temporary），效率自然低下。由此最后可得出，优先选择驱动表的属性进行排序能够提高连接效率。