在一个千万级的数据库查寻中，如何提高查询效率？分别说出在数据库设计、SQL语句、java等层面的解决方案。

1）数据库设计方面：

a. 对查询进行优化，应尽量避免全表扫描，首先应考虑在where及order by涉及的列上建立索引。

b. 应尽量避免在where子句中对字段进行null值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：select id from t where num is null 可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：select id from t where num=0

c. 并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时,查询可能不会去利用索引，如一表中有字段sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。

d. 索引并不是越多越好，索引固然可以提高相应的select的效率，但同时也降低了insert及update的效率，因为insert或update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

e. 应尽可能的避免更新索引数据列，因为索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新索引数据列，那么需要考虑是否应将该索引建为索引。

f. 尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

g. 尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

h. 尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

i. 避免频繁创建和删除临时表，以减少系统表资源的消耗。

j. 临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。

k. 在新建临时表时，如果一次性插入数据量很大，那么可以使用select into代替create table，避免造成大量log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

l. 如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先truncate table ，然后drop table ，这样可以避免系统表的较长时间锁定。

2)SQL语句方面：

a. 应尽量避免在where子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

b. 应尽量避免在where子句中使用or来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：select id from t where num=10 or num=20可以这样查询：select id from t where num=10 union all select id from t where

num=20

c. in 和 not in 也要慎用，否则会导致全表扫描，如： select id from t where num in(1,2,3)对于连续的数值，能用between就不要用in了：select id from t where num between 1 and 3

d. 下面的查询也将导致全表扫描： select id from t where name like ‘%abc%’

e. 如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：select id from t where num=@num 可以改为强制查询使用索引：select id from t with(index(索引名)) where num=@num

f. 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如： select id from t where num/2=100 应改为: select id from t where num=100\*2

g. 应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如： select id from t where substring(name,1,3)=’abc’–name以abc开头的id select id from t where datediff(day,createdate,’2005-11-30′)=0–‘2005-11-30’生成的id 应改为: select id from t where name like ‘abc%’ select id from t where createdate>=’2005-11-30′ and createdate<’2005-12-1′

h. 不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

i. 不要写一些没有意义的查询，如需要生成一个空表结构：select col1,col2 into #t from t where 1=0 这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样： create table #t(„)

j. 很多时候用 exists 代替 in 是一个好的选择：select num from a where num in(select num from b)用下面的语句替换：select num from a where exists(select 1 from b where num=a.num)

k. 任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。

l. 尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

m. 尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

n. 尽量避免大事务操作，提高系统并发能力。

3)java方面:

a.尽可能的少造对象。

b.合理摆正系统设计的位置。大量数据操作，和少量数据操作一定是分开的。大量的数据操作，肯定不是ORM框架搞定的。，

c.使用jDBC链接数据库操作数据

d.控制好内存，让数据流起来，而不是全部读到内存再处理，而是边读取边处理；

e.合理利用内存，有的数据要缓存

脏读：一个事务读到了另一个事务已经修改而没提交的事务

幻影读：同一个事务，两次读取同一数据 得到不同的结果

**存储过程**

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

优点

①重复使用。存储过程可以重复使用，从而可以减少数据库开发人员的工作量。

②提高性能。存储过程在创建的时候在进行了编译，将来使用的时候不再重新翻译。一般的SQL语句每执行一次就需要编译一次，所以使用存储过程提高了效率。

③减少网络流量。存储过程位于服务器上，调用的时候只需要传递存储过程的名称以及参数就可以了，因此降低了网络传输的数据量。

④安全性。参数化的存储过程可以防止SQL注入式攻击，而且可以将Grant、Deny以及Revoke权限应用于存储过程。

desc降序 asc升序

**产生死锁的原因**

因为系统资源不足。（2） 进程运行推进的顺序不合适。（3） 资源分配不当等。  
如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。

**产生死锁的四个必要条件**：

（1） 互斥条件：一个资源每次只能被一个进程使用。  
（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。  
（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。  
（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系

**如何将死锁减至最少**

保持事务简短并在一个批处理中

在同一数据库中并发执行多个需要长时间运行的事务时通常发生死锁。事务运行时间越长，其持有排它锁或更新锁的时间也就越长，从而堵塞了其它活动并可能导致死锁

使用低隔离级别

确定事务是否能在更低的隔离级别上运行。执行提交读允许事务读取另一个事务已读取（未修改）的数据，而不必等待第一个事务完成。使用较低的隔离级别（例如提交读）而不使用较高的隔离级别（例如可串行读）可以缩短持有共享锁的时间，从而降低了锁定争夺。

避免事务中的用户交互

避免编写包含用户交互的事务，因为运行没有用户交互的批处理的速度要远远快于用户手动响应查询的速度，例如答复应用程序请求参数的提示。例如，如果事务正在等待用户输入，而用户去吃午餐了或者甚至回家过周末了，则用户将此事务挂起使之不能完成。这样将降低系统的吞吐量，因为事务持有的任何锁只有在事务提交或回滚时才会释放。即使不出现死锁的情况，访问同一资源的其它事务也会被阻塞，等待该事务完成

**Sql四种连接方式 左外连接、右外连接、内连接、全连接**

内连接与乘法结果相同。select \* from A,B where A.Aid=B.Bnameid与Select \* from A JOIN B ON A.Aid=B.Bnameid的运行结果相同

外连接：外连接分为两种，一种是左连接（Left JOIN）和右连接（Right JOIN）

左连接（Left JOIN）：即两张表公共部分记录集C＋表A记录集A1。

语句如下：select \* from A Left JOIN B ON A.Aid=B.Bnameid

在语句中，A在B的左边，并且是Left Join，所以其运算方式为：A左连接B的记录=图3公共部分记录集C＋表A记录集A1。

完全连接 FULL  JOIN 或 FULL OUTER JOIN  
完整外部联接返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。如果表之间有匹配行，则整个结果集行包含基表的数据值。

交叉连接（笛卡尔积）

生成笛卡尔积——它不使用任何匹配或者选取条件，而是直接将一个数据源中的每个行与另一个数据源的每个行一一匹配

**数据库索引**

索引是对数据库表中一个或多个列（例如，employee 表的姓名 (name) 列）的值进行排序的结构。如果想按特定职员的姓来查找他或她，则与在表中搜索所有的行相比，索引有助于更快地获取信息。

例如这样一个查询：select \* from table1 where id=10000。如果没有索引，必须遍历整个表，直到ID等于10000的这一行被找到为止；有了索引之后(必须是在ID这一列上建立的索引)，即可在索引中查找。由于索引是经过某种算法优化过的，因而查找次数要少的多。可见，索引是用来定位的。

数据库索引好比是一本书前面的目录，能加快数据库的查询速度。索引分为**聚簇索引和非聚簇索引两种**，聚簇索引是按照数据存放的物理位置为顺序的，而非聚簇索引就不一样了；**聚簇索引能提高多行检索的速度，而非聚簇索引对于单行的检索很快**

根据数据库的功能，可以在数据库设计器中创建三种索引：**唯一索引、主键索引和聚集索引**。

唯一索引 唯一索引是不允许其中任何两行具有相同索引值的索引。

当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在employee表中职员的姓(lname)上创建了唯一索引，则任何两个员工都不能同姓。

主键索引

数据库表经常有一列或多列组合，其值唯一标识表中的每一行。该列称为表的主键。

在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

聚集索引

在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度

语法 CREATE INDEX mytable\_categoryid ON mytable (category\_id);

索引可以避免全表扫描。多数查询可以仅扫描少量索引页及数据页，而不是遍历所有数据页。

对于非聚集索引，有些查询甚至可以不访问数据页。

聚集索引可以避免数据插入操作集中于表的最后一个数据页。

一些情况下，索引还可用于避免排序操作。

建立太多的索引将会影响更新和插入的速度，因为它需要同样更新每个索引文件。对于一个经常需要更新和插入的表格，就没有必要为一个很少使用的where字句单独建立索引了，对于比较小的表，排序的开销不会很大，也没有必要建立另外的索引。

**日志文件**（Log File）记录所有对数据库数据的修改，主要是保护数据库以防止故障,以及恢复数据时使用。

**数据库三范式**

第一范式（1NF）

所谓第一范式（1NF）是指在关系模型中，对域添加的一个规范要求，所有的域都应该是原子性的，即数据库表的每一列都是不可分割的原子数据项，而不能是集合，数组，记录等非原子数据项。即实体中的某个属性有多个值时，必须拆分为不同的属性。在符合第一范式（1NF）表中的每个域值只能是实体的一个属性或一个属性的一部分。简而言之，第一范式就是无重复的域。

在1NF的基础上，非码属性必须完全依赖于码[在1NF基础上消除非主属性对主码的部分函数依赖]

第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。第二范式（2NF）**要求数据库表中的每个实例或记录必须可以被唯一地区分。**选取一个能区分每个实体的属性或属性组，作为实体的唯一标识。例如在员工表中的身份证号码即可实现每个一员工的区分，该身份证号码即为候选键，任何一个候选键都可以被选作主键。在找不到候选键时，可额外增加属性以实现区分，如果在员工关系中，没有对其身份证号进行存储，而姓名可能会在数据库运行的某个时间重复，无法区分出实体时，设计辟如ID等不重复的编号以实现区分，被添加的编号或ID选作主键。（该主键的添加是在ER设计时添加，不是建库时随意添加）

第二范式（2NF）**要求实体的属性完全依赖于主关键字**。所谓完全依赖是指**不能存在仅依赖主关键字一部分的属性**，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。为实现区分通常需要为表加上一个列，以存储各个实例的唯一标识。简而言之，第二范式就是在第一范式的基础上属性完全依赖于主键。

第三范式（3NF）

在1NF基础上，**任何非主属性不依赖于其它非主属性**[在2NF基础上消除传递依赖]

第三范式（3NF）是第二范式（2NF）的一个子集，即满足第三范式（3NF）必须满足第二范式（2NF）。简而言之，第三范式（3NF）要求一个关系中不包含已在其它关系已包含的非主关键字信息。例如，存在一个部门信息表，其中每个部门有部门编号（dept\_id）、部门名称、部门简介等信息。那么**在员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。**如果不存在部门信息表，则根据第三范式（3NF）也应该构建它，**否则就会有大量的数据冗余**。简而言之，第三范式就是属性不依赖于其它非主属性，也就是在满足2NF的基础上，任何非主属性不得传递依赖于主属性。

数据库约束

主键约束、非空约束、唯一约束（unique）、检查约束（check (age between 0 and 150)）、主外键约束（增强表的完整性，删除要先删除子表）删除的时候要级联删除

**数据库触发器**

CREATE TRIGGER `<databaseName>`.`<triggerName>`

< [ BEFORE | AFTER ] > < [ INSERT | UPDATE | DELETE ] >

ON [dbo]<tableName> //dbo代表该表的所有者

FOR EACH ROW

BEGIN

--do something

END |

**Sql性能优化**