1. **数据库存储引擎**
2. 定义：数据库引擎是用于存储、处理和保护数据的核心服务。利用数据库引擎可控制访问权限并快速处理事务，从而满主企业内大多数需要处理大量数据的应用程序的要求。使用数据库引擎创建用于联机事务处理或联机分析处理数据的关系数据库。这包括创建用于存储数据的表和用于查看、管理和保护数据安全的数据库对象（如索引、视图、存储过程）。
3. 为什么要合理选择数据库存储引擎：MYSQL中的数据用各种不同的技术存储在文件（或）内存中。这些技术中的每一种技术都使用不同的存储机制、索引技巧、锁定水平并且最终提供广泛的不同的功能和能力。通过选择不同的技术，能够获得额外的速度或功能，从而改善应用的整体功能。
4. 存储引擎的作用：
5. 设计并创建数据库以保存系统所需的关系或xml文档。
6. 实现系统以访问和更改数据库中存储的数据。
7. 为单位和客户部署实现的系统。
8. 提供日常管理支持以优化数据库的性能。
9. MYSQL的各大存储引擎：
10. **InnoDB**：InnoDB是MYSQL默认的事务型引擎，也是最重要、最广泛的存储引擎。它的设计是用来处理大量短期事务，短期事务大部分是正常提交的，很少回滚。InnoDB的性能和自动崩溃恢复特性，使得它在非事务型存储的需求中，也很流行。InnoDB的数据存储在表空间中，表空间是由InnoDB管理的黑盒文件系统，由一系列系统文件组成。InnoDB可以将每个表的数据和索引存放在单独的文件中。InnoDB也可以使用裸设备作为表空间存储介质。InnoDB通过间隙锁防止幻读的出现。InnoDB是基于聚簇索引建立，与其他存储引擎有很大区别，聚簇索引对主键查询有很高的性能，不过它的二级索引必须包含主键列，所以如果主键列很大的话，索引会很大。
11. MyISAM：在5.1之前，MyISAM是默认的引擎，包括全文索引、压缩、空间函数等特性，但是不支持事务和行级锁，而且在崩溃后无法安全恢复，即使后续版本中支持了事务，但是很多人的概念中依然是不支持事务的引擎。MySAM是对整个表加锁，而不是行锁，读取的时候对表加共享锁，写入的时候加排他锁，但是在表有读取查询的同时，也可以往表内写入数据，由于没有行锁机制，所以在海量写入的时候，会导致所有查询出路Locked状态。对于MyISAM，即使是Blob、Text等长字段，也可以基于前500字符创建索引，支持全文索引，这是一个基于分词创建的索引，也可以支持复杂的查询。MyISAM可以选择延迟更新索引键，在创建表的时候指定delay\_key\_write选项，在每次执行修改完成时，不会立刻将修改的索引数据写入磁盘，而是写道缓存区，只有在清理缓存区或者关闭表的时候才会将索引写入磁盘，可以极大的提升写入性能，但是在主机崩溃时会造成索引损坏，需要执行修复操作。MyISAM支持压缩表，如果数据在写入后不会修改，那么这个表适合压缩表，可以使用myisampack对表经行打包，压缩表是不可以修改数据的。压缩表可以极大的减少磁盘占用，因此可以减少磁盘IO，提升性能，压缩表也支持索引，但是索引也是只读的。对于一些只读数据，或者表空间较小，可以忍受恢复操作，可以使用MyISAM引擎。
12. Archive：Archive引擎支持insert、select操作，现在支持索引。Archive引擎会缓存所有的写，并利用zlib对写入行进行压缩，磁盘io很少，但是每次在select查询都需要全表扫描。Archive引擎支持行级锁和专用的缓存区，所有可以实现高并发写入，在查询开始到返回表存在的所有行数之前，会阻止其他select执行，用来实现一致读，另外也实现了批量写入结束前对读操作不可见，这种机制模仿了事务和MVCC的特性，但不是事务型引擎，而是针对高写入压缩做了优化的的简单引擎。Archive适合日志和数据采集应用，这类应用在分析时往往需要全表扫描或者更快的insert操作场景。0
13. Blackhole：Blackhole没有实现任何存储机制，它会舍弃所有写入数据，不做任何保存，但是服务器会记录Blackhole表的日志，用于复制数据到备库，或者只是简单的记录到日志，这种特殊的存储引擎可以在一些特殊的复制架构和日志审核时发挥做用，但是不推荐。
14. CVS：CVS引擎可以将普通的CVS文件作为MYSQL表来处理，但是这种表不支持索引，CVS可以在数据库运行时拷贝或者拷出文件，可以将Excel等电子表格中的数据存储为CVS文件，然后复制靠MYSQL中，就能在MYSQL中打开使用。如果将数据写入导一个CVS引擎表，其他外部程序也可以从表的数据文件中读取CVS的数据，因此可以作为数据交换机制。
15. Federated：Federated引擎是访问其他MYSQL服务器的一个代理，他会创建一个到远程MYSQL服务器的客户端连接，并将查询传输到远程服务器执行，然后提取或者发送需要的数据。最初设计该存储引擎是为了和其他数据库类似特性竞争，可以说是一种市场行为，因经常带来问题，默认是禁用的。
16. Memroy：如果需要快速的访问数据，并且这些数据不会被修改，重启以后丢失也没关系，那么使用Memroy表是非常有用的。Memroy表的结构在重启以后还会保留，但数据会丢失。Memory表支持Hash索引，因此查快操作非常快，但是是表级锁，并发写入的性能较低。它不支持Blob和Text类型的列，并且每行的长度是固定的，所以即使指定了varchar列，实际存储时也会转化城char，这可能导致部分内存的浪费。
17. Merge：Merge引擎是MyISAM引擎的一个变种，是由多个MyISAM表合并而来的虚拟表。如果将MYSQL用于日志或者数据仓库类应用，该引擎可以发挥做用，但是引入分区功能后，该引擎已经被放弃。
18. NDB集群引擎：NDB集群存储引擎，作为SQL和NDB原生协议之间的接口。MYSQL服务器、NDB集群存储引擎，以及分布式的、share-nothing的、容灾的、高可用的NDB数据库的组合，被称为MYSQL集群。
19. PERFORMANCE\_SCHEMA：该引擎是5.5新增的，主要用于收集数据库服务器性能参数。MYSQL用户不能创建存储引擎为PERFORMANCE\_SCHEMA的表。该引擎主要做用：提供进程等待的详细信息，包括锁、互斥变量、文件信息；保存历史的事件汇总信息，为提供MYSQL服务器性能做出详细的判断；对于新增和删除监控事件点都非常容易，并可以随意改变mysql服务器的监控周期。
20. 数据库存储引擎的修改：
21. 修改配置文件，在配置文件后面添加default-storage-engine-InnoDB，重启服务，数据库默认的引擎就修改为InnoDB了。
22. 在创建表的时候指定create table mytbl(id int primary key,name varchar(50))type=MyISAM；
23. 建表后修改：alter table table\_name type = InnoDB；
24. 查看是否修改成功：show table status from table\_name；show create table table\_name;
25. 通过数据库管理工具也可以实现以上功能，例如navicat
26. **索引是什么：首先我们可以举个例子，字典大家都使用过，我们可以使用目录快速定位到所要查找的内容，那么索引跟目录的作用类似，在数据库表记录中，利用索引，可以快速过滤查找到数据记录。**
27. 索引类型分类：
    1. 普通索引：加快查询速度，允许为空
    2. 唯一索引：加快查询速度，并且可以限制列值的唯一性，允许为空；唯一索引包含联合唯一索引，多列形成的唯一索引，这些列可以唯一确定一条表的记录，可以为空。
    3. 全文索引：主要针对文本的内容进行分词，加快查询速度。
    4. 联合索引：多列组成的索引，查询效率提升高于多个单列索引合并的效率。
    5. 主键索引：主键唯一性，加快查询速度，不能为空。
28. 索引结构分类：
    1. 哈希索引：哈希索引基于哈希表实现，对于没一行数据，存储引擎会对所有的索引列计算一个哈希码，然后存储引擎会基于这个哈希码来查找数据，只能在=，< = >, in条件中生效。
    2. 全文索引：全文索引在几种结构中比较特殊，他查找的是文本中关键词。
    3. B-tree索引：该索引普遍用于存储引擎中，使用B-tree数据结构存储数据，非常方便查询，适用于全键值，键值范围，或者前缀查找。
29. 建议不要使用索引的几种情况：
    1. 区分度不是很大的字段，例如：性别、状态
    2. 频繁更新的字段
    3. 不在where列中出现的索引
30. 索引失效的几种情况：
    1. 查询列中有函数计算
    2. 如果查询条件中有or，索引会失效，除非所有条件都加上所有
    3. Is null 或者 is not null条件会使索引失效
    4. 字符串不加引号，会导致索引失效
    5. 联合索引中会遵循最左原则，如果要使用联合索引，那么前面的索引列一定要包含，否则索引会失效
    6. Like条件中 % 前置会使索引失效，后置不会使索引失效
31. 聚簇索引和非聚簇索引：
    1. 聚簇索引：将数据存储与索引放到一块，索引结构的叶子节点保存了行数据
    2. 非聚簇索引：将数据与索引分开存储，索引结构的叶子节点指向了数据对应的位置。
    3. 在InnoDB中，主键索引是聚簇索引，在聚簇索引上创建的索引是辅助索引，辅助索引都是非聚簇索引。辅助索引叶子节点存储的是主键值，不是行的物理位置，辅助索引访问数据总是需要二次查找。
    4. 聚簇索引具有唯一性，由于聚簇索引是将数据和索引结构放到一块，因此一个表只有一个聚簇索引。
    5. 聚簇索引默认是主键，如果表中没有定义主键，InnoDB会选择一个唯一且非空的索引代替。如果没有这样的索引，InnoDB会隐式定义一个主键来作为聚簇索引。如果已经设置了主键为聚簇索引又希望再单独设置聚簇索引，必须先删除主键，然后添加我们想要的聚簇索引，最后恢复设置主键即可。
    6. 为什么辅助索引叶子节点存储的是主键值而不是数据存放的地址？如果存放的是数据地址，当行数据发生变化时，辅助索引树的节点也需要分裂变化；存放主键的的话，我们只需维护聚簇索引树就好。另一个好处时，辅助索引存放主键值，减少了辅助索引占用内存空间的大小。
32. 联合索引为什么要遵循最左原则？

索引是一棵B+Tree，其非叶子节点存储的是关键字的索引，叶子节点存储的数据，多列组合索引，是按左右顺序组成的关键字，所以需要遵循最左原则。

1. ss
2. **Explain执行计划包含的信息：id、select\_typ、type、key、rows、extra**
3. id：select查询的序列号，包含一组数字，表示查询中select子句或操作表的顺序，可分为三种情况：
4. id相同：执行顺序由上至下。
5. id不同：如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行。
6. id相同又不同（两种情况同时存在）：id如果相同，可以认为是一组，丛上往下顺序执行；在所有组中，id值越大，优先级越高，越先执行。
7. select\_type：查询的类型，主要是用于区分普通查询、联合查询、子查询等复杂的查询
8. SIMPLE：简单的select查询，查询中不包含子查询或者union。
9. PRIMARY：查询中包含任何复杂的子部分，最外层查询则被标记为primary。
10. SUBQUERY：在select或where列表中包含子查询。
11. DERIVED：在from列表中包含的子查询被标记为derived（衍生），mysql或递归执行这些子查询，把结果放在零时表里。
12. UNION：若第二个select出现在union之后，则被标记为union；若union包含在from子句的子查询中，外层select将被标记为derived。
13. UNION RESULT：从union表获取结果的select。
14. type：访问类型，sql查询优化中一个很重要的指标，结果值从好到坏依次是：

**system > const > eq\_ref > ref > fulltext > ref\_or\_null > index\_merge > unique\_subquery > index\_subquery > range > index > all**

1. system：表只有一行记录（等于系统表），这是const类型的特例，平时不会出现，可以忽略不记。
2. const：表示通过索引一次就找到了，const用于比较primary key或者unique索引。因为只需要匹配一行数据，所以很快。如果将主键置于where条件中，myslq就能将该查询转化为一个const。
3. eq\_ref：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常用于主键和唯一索引扫描。
4. ref：非唯一性索引扫描，返回匹配某个单独值的所有行。本质也是一种索引访问，它返回所有匹配某个单独值的行，然而他可能会找到多个符合条件的行，所以它应该属于查找和扫描的混合体。
5. range：只检索给定范围的行，使用一个索引来选择行。一般就是在where语句中出现了between、<、>、in等查询，这种索引列上的范围扫描比全索引扫描要好。只需开始于某个结点，结束于另一个点，不用扫描全部索引。
6. index：index扫描全部索引树，从索引中读取，索引文件通常比数据文件小。
7. all：遍历全表找到匹配的行，从硬盘读取。
8. table：显示这一行的数据是关于哪张表的
9. possible\_keys：查询涉及到的字段上存在索引，则该索引将被列出，但不一定被查询实际使用
10. key：实际使用的索引，如果为null，则没有使用索引。查询中如果使用了覆盖索引，则该索引仅出现在key列表中。
11. key\_len：表示索引中使用的字节数，查询中使用的索引的长度（最大可能长度），并非实际使用长度，理论上长度越短越好。ken\_len是根据表定义计算而得，不是通过表内检索出的。
12. ref：显示索引的那一列被使用了，如果可能，是一个常量。
13. rows：根据表统计信息及索引选用情况，大致估算出找到所需的记录所需要读取的行数。
14. extra：不适合在其他字段中显示，但是十分重要的额外信息
15. using filesort：mysql对数据使用了一个外部的索引排序，而不是按照表内的索引进行排序读取。也就是说mysql无法利用索引完成的排序操作成为“文件排序“。
16. using temporary：使用临时表保存中间结果，也就是说mysql在对查询结果排序使用了临时表，常见于order by和group by。
17. using index：表示相应的select操作中使用了覆盖索引，避免了访问表的数据行，效率高。如果同时出现using where，表明索引被用来执行索引键值的查询。如果没有出现using where，表明索引用来读取数据而非执行查找动作。**覆盖索引就是select列表中的字段，只用从索引中就能获取，不必根据索引再次读取数据文件，换句话说查询列要被所建的索引覆盖**。
18. using where：使用了where过滤。
19. using join buffer：使用了链接缓存。
20. impossible where：where子句的值总是false，不能用来获取任何元组。
21. select tables optimized away：在没有gourp by子句的情况下，基于索引优化MIN/MAX操作或者对于myisam存储引擎优化count（\*）操作，不必等到执行阶段在进行计算，查询执行计划生成的阶段即可完成优化。
22. distinct：优化distinct操作，在找到第一个匹配的元组后即停止找同样值的动作。
23. not exists：mysql优化left join，一旦它找到匹配left join标准的行，就不再搜索了。
24. **Mysql主键uuid和自增主键区别优劣**
25. uuid的做主键的优缺点：
26. 有点：出现数据拆分、合并存储的时候，能达到全局的唯一性。
27. 缺点：影响插入速度，并且造成硬盘使用率低；比较大小相对数字慢不少，影响查询速度；占用空间大，建的索引越多，影响越严重。
28. 自增主键的优缺点：
29. 有点：存储和读取速度都是最快的，而且占的存储空间也是最小。
30. 缺点：两张自增id做主键的表合并时，id一定会有冲突，但如果各自的id还关联了其他表，这就很不好操作了。
31. 综合可得：
32. 不和业务关联，只做新增和查询，为了存储和查询性能应该使用自增长id做主键，例如业务操作展示日志。
33. 由于uuid的无序性，InnoDB会产生巨大的IO压力，此时不适合用uuid做物理主键，可以把它作为逻辑主键，物理主键依然使用自增id，为了全局的唯一性，用uuid做索引关联其他表或做外键。
34. 如果是主从（M-S）模式，最好是不使用mysql自带函数uuid（）来生成唯一主键，因为主表生成的uuid要再关联从表时，需要再去数据库查出这个uuid，需要多进行一次数据库交互，而且在这个时间差里面主表很可能还有数据生成，这样就很容易导致关联的uuid出错。如果真要使用uuid，可以在java中生成后，直接存储到DB里，这时主从的uuid就是一样的了。
35. **SQL语句的执行过程**
36. Select执行过程：

（8）**select**

（9）**distinct** <select list>   
（1）**from** <left table>   
（3）<join type> **join** <right table>   
（2）**on**<join condiction>   
（4）**where**<where condition>   
（5）**group by** < group by list >   
（6）**with {CUBE|ROLLUP}**   
（7）**having** <having condition >   
（10）**order by**<order by list>   
（11）**limit**<limit condition>

(1)、from：对from的左边表和右边表计算笛卡尔积，产生虚表VT1

(2)、on：对虚拟表VT1进行on条件赛选，记录在虚拟表VT2中

(3)、join：如果指定了join方向（left join、right join），那么保留行中未匹配的行就会作为外部行添加到虚拟表VT2中，产生虚拟表VT3。多表关联重复1~3步骤，直到处理完所有表为止。

(4)、where：对虚拟表VT3进行where条件过滤，符合条件的添加到虚拟表VT4中

(5)、group by：根据group列对虚拟表VT4进行分组操作，产生虚拟表VT5

(6)、with cube|rollup：对虚拟表VT5进行cube或rollup操作，产生虚拟表VT6

(7)、having：根据having条件对虚拟表VT6过滤，产生虚拟表VT7

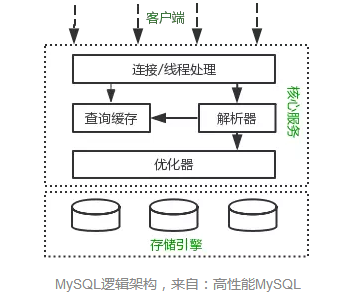
(8)、select：在虚拟表VT7中选择select指定列，插入到虚拟表VT8

(9)、distinct：对虚拟表VT8表进行去重，产生虚拟表VT9

(10)、order by：根据order列对虚拟表VT9进行排序，产生虚拟表VT10

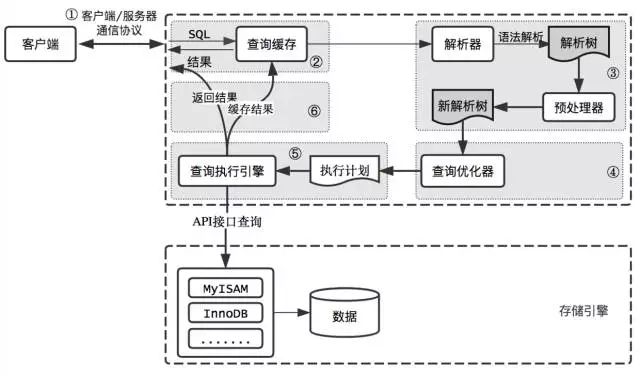
(11)、limit：根据limit条件取出指定行记录，产生虚拟表VT11

1. **MySQL的优化策略**
2. Mysql的逻辑架构图(总共分为三层)：



* 1. 最上层为客户端，并非mysql独有，诸如：连接处理、授权认证、安全等功能均在这一层处理。
  2. Mysql大多数核心服务均在中间这一层，包括查询解析、分析、优化、缓存、内置函数。所有的跨存储引擎的功能也在这一层实现：存储过程、触发器、视图等。
  3. 最下层为存储引擎，其负责mysql中的数据存储喝提取。喝linux下的文件系统类似，每种存储引擎都有其优势和劣势。中间的服务层通过API与存储引擎通信，这些API接口屏蔽了不同存储引擎间的差异。

1. Mysql的查询过程



1. 客户端/服务端通信协议
2. Mysql客户端/服务端通信协议是“半双工”的：在任一时刻，要么是服务器向客户端发送数据，要么是客户端向服务器发送数据，这2个动作不能同时发生。一旦一端开始发送消息，另一端要接收完整个消息才能响应它，所以我们无法也无须将一个消息切成小块独立发送，也没办法进行流量控制。
3. 客户端用户一个单独的数据包将查询请求发送给服务器，所以当查询语句很长的时候，需要设置max\_allowed\_packet参数。但是需要注意的是，如果查询实在是太大，服务端会拒绝接收更多数据并抛出异常。
4. 与之相反的是，服务器响应给用户的数据通常会很多，由多个数据包组成。但是当服务器响应客户端请求时，客户端必须完整的接收整个返回结果，而不能简单的只取前面几条结果，然后让服务器停止发送。因而在实际开发中，尽量保持查询简单且只返回必需的数据，减少通信间数据包的大小和数量是一个非常好的习惯，这也是查询中尽量避免使用select \* 以及加上limit 限制的原因之一。
5. 查询缓存
6. 在解析一个查询语句前，如果查询缓存是打开的，如果查询缓存是打开的，那么mysql会检查这个查询语句是否命中查询缓存中的数据。如果当前查询恰好命中查询缓存，在检查一次用户权限后直接返回缓存中的结果。这种情况下，查询不会被解析，也不会生成执行计划，更不会被执行。
7. Mysql将缓存存放在一个引用表（可以认为是类似于HashMap的数据结构），通过一个hash索引，这个哈希值通过查询本身、当前要查询的数据库、客户端端协议版本号等一些可能影响结果的信息计算得来。所以两个查询任何字符上的不用（例如：空格、注释），都会导致缓存不会命中。
8. 如果查询中包含任何用户自定义函数、存储函数、用户变量、临时表、mysql库中的系统表，其查询结果都不会被缓存。比如函数now（）或者current\_date（）会因为不同的查询时间，返回不通的查询结果，再比如包含current\_user或者connecion\_id（）的查询语句会因为不同的用户而返回不同的结果，将这样的查询缓存起来没有任何意义。
9. Mysql的查询缓存系统会跟踪查询中涉及的每个表，如果这些表（数据或结构）发生变化，那么和这张表相关的所有缓存数据都将失效。正因为如此，在任何写操作时，mysql必须将对应的所有缓存都设置为失效。如果查询缓存非常大或者碎片非常多，这个操作就可能带来很大的系统消耗，甚至导致系统僵死一会。而且查询缓存对系统的额外消耗也不仅仅在写操作，读操作也不例外：任何查询语句在开始之前都必须经过检查，即使这条SQL语句永远不会命中缓存；如果查询结果可以被缓存，那么执行完成后，会将结果存入缓存，也会带来额外的系统消耗。
10. 缓存和失效都会带来额外消耗，只有当缓存带来的资源节约大于其本身消耗的资源时，才会给系统带来新能提升。写密集型应用，最好不要轻易打开查询缓存，可以设置query\_cache\_type=DEMAND，这时只有加入SQL\_CACHE的查询才会走缓存，其他查询则不会，这样可以非常自由的控制哪些查询需要被缓存。
11. 如果系统确实存在一些性能问题，可以尝试打开缓存，并在数据库设计上做一些优化：用小表代替大表，注意不要过度设计；批量插入代替循环单条插入；合理控制缓存空间大小，一般来说设置为几十兆比较合适；通过SQL\_CACHE和SQL\_NO\_CACHE来控制某个查询语句是否需要进行缓存。
12. 语法解析和预处理

Mysql通过关键字将SQL语句进行解析，并生成一颗对应的解析树。这个过程解析器主要通过语法规则来验证和解析。比如SQL中是否使用了错误的关键字或者关键字的顺序是否正确等等。预处理则会根据mysql规则进一步检查解析树是否合法。比如检查要查询的数据表和数据列是否存在等。

1. 查询优化
2. 经过前面的步骤生成的语法树认为是合法的了，并且由优化器将其转化成查询计划。多数情况下，一条查询可以有很多种的执行方式，最后都会返回相应的结果。优化器的作用就是找到这其中最好的执行计划。
3. Mysql使用的是基于成本的优化器，它尝试预测一个查询使用某种执行计划时的成本，并选择其中成本最小的一个。在mysql可以通过查询当前会话的last\_query\_cost的值来得到其计算当前查询的成本。计算成本的信息包括：每张表或者索引的页面个数、索引的基数、索引和数据行的长度、索引的分布情况等等。
4. 有非常多的原因会导致mysql选择错误的执行计划，比如统计信息不准确、不会考虑不受其控制的操作成本（用户自定义函数、存储过程）、mysql认为的最优和我们想的不一样（我们希望执行时间尽可能短，但mysql值选择它认为成本最小的，但成本最小不一定执行时间最短）等等。
5. Mysql的查询优化器是一个非常复杂的部件，它使用了非常多的优化策略来生成一个最优的执行计划：--重新定义表的关联顺序；--优化MIN（）和MAX（）函数（如果该列有索引，最小值只需要查找B+Tree索引最左端，最大值只需查询B+Tree索引最右端）；--提前终止查询（比如：使用limit时，查询到满足数量的结果集后会立即终止查询）；--优化排序（老版本会使用两次传输排序，即先读取行指针和需要排序的字段在内存中对其进行排序，然后再根据结果集去读取数据行，而新版本单次传输排序，也就是一次读取所有的数据行，然后根据给定的列排序。对于I/O密集型应用，效率会高很多）。
6. 查询执行引擎

在完成解析和优化阶段后，mysql会生成对应得执行计划，查询执行引擎根据执行计划给出得指令逐步执行得出结果。整个执行过程得大部分操作均是通过调用存储引擎实现得接口来完成，这些接口统称为handler API。查询过程中的每一张表由一个handler实例表示。Mysql在查询优化阶段就为每一张表创建了一个handler实例，优化器可以根据这些实例的接口来获取表的相关信息，包括表的所有列名、索引统计信息等。

1. 返回给客户端
2. 查询执行的最后一个阶段就是将结果返回给客户端。即使查询不到数据，mysql仍然会返回这个查询的相关信息，比如该查询影响到的行数以及执行时间等。
3. 如果查询缓存被打开，并且这个查询可以被缓存，mysql也会将结果放到缓存中。
4. 结果集返回给客户端是一个增量且逐步返回的过程。有可能mysql在生成第一条结果时，就开始向客户端逐步返回结果集了。这样服务端就无须存储太多结果而消耗过多内存，也可以让客户端第一时间获得返回结果。结果集的每一行都会以满足上面所描述的通信协议的数据包发送，再通过TCP协议进行传输，在传输过程中，可能对mysql的数据包进行缓存然后批量发送。
5. Mysql整个查询过程总的来说分为6个步骤
6. 客户端向mysql服务器发送一条查询请求
7. 服务器首先检查查询缓存，如果命中缓存，则立刻返回存储在内存中的结果。否则进入下一阶段。
8. 服务器进行SQL解析、预处理、再由优化器生成对应的执行计划
9. Mysql根据执行计划，调用存储引擎的API来执行查询
10. 将结果返回给客户端，同时缓存查询结果
11. 性能优化建议（优化没有绝对的真理，应该再实际的业务场景下通过测试来验证关于执行计划和响应时间的假设）
12. Scheme设计与数据类型优化

选择数据类型只要遵循小而简单的原则就好，越小的数据类型通常会更快，占用更少的磁盘、内存，处理时需要的CPU周期也更少。比如，会使用整形来存储ip地址，使用datetime来存储时间，而不是使用字符穿。几个容易错误理解的技巧

1. 通常来说把可为NULL改为NOT NULL不会对性能提升有多少帮助，只是如果计划在列上创建索引，就应该将该列设置为NOT NULL。
2. 对整数类型指定宽度，比如int（11），没有任何用。Int使用32位（4个字节）存储空间，它表示的存储返回已经确定，所以int（1）和int（20）对于存储和计算是相同的。
3. Unsigend表示不允许负值，大致可以使正数的上限提高一倍。
4. 通常来讲，没有太大必要使用decimal数据类型。即使是在需要存储财务数据时，仍然可以用bigint。这样可以避免浮点数计算不准确和decimal精确计算代价高的问题。
5. Timestamp使用4个字节存储空间，datetime使用8个字节存储空间，而且timestamp的值因时区不同而不同。
6. 大多数情况下没有使用枚举类型的必要，其中一个缺点是枚举的字符串列表是固定的，添加和删除字符串必须使用alter table（只是在列表末尾追加元素，不需要重建表）。
7. Schema的列不要太多。存储引擎的API工作时需要在服务器层和存储引擎层之间通过行缓冲格式拷贝数据，然后再服务器层将缓冲内容解码成各个列，这个转化过程的代价是非常高的。如果列太多而实际使用的列又很少的话，有可能导致cpu占用过高。
8. 大表的alter table非常耗时，mysql执行大部分修改表结构操作的方法是创建一张新表，从旧表中查出数据插入新表，然后再删除旧表。
9. 创建高性能索引

索引是提高mysql查询性能的一个重要途径，但是过多的索引可能会导致过高的磁盘使用率以及过高的内存占用，从而影响应用程序的整体性能。应当尽量避免时候才想起添加索引，因为事后可能需要监控大量的SQL才能定位到问题所在，而且添加索引的时间肯定是远远大于初始添加索引的时间

1. 独立的列（索引列不能是表达式的一部分，也不能是函数的参数）mysql才会使用索引。
2. 如果列很长，通常可以索引开始的部分字符，这样可以有效节约索引空间，从而提高索引效率。
3. 当出现多个索引做相交操作时（and拼接），通常来说一个包含所有列的索引要优于多个独立索引，选择性更高的字段放到索引的前面，这样通过第一个字段旧可以过滤大部分不符合条件的数据。
4. 避免多个范围条件，多个范围条件时，mysql能使用一个索引列，但是无法同时使用它们。
5. 如果一个索引包含或者覆盖所有需要查询的字段的值，那么久没有必要再回表查询，这就称为覆盖索引。索引条目远小于数据行大小，如果只读取索引，极大减少数据访问量。索引是有按照列值顺序存储的，对于I/O密集型的范围查询要比随机从磁盘读取每一行数据的IO要少的多。
6. 在设计索引时，如果一个索引既能够满足排序，有满足查询是最好的。只有当索引的列顺序和order by子句的顺序完全一致，并且所有列的排序方向也一样时，才能够使用索引来对结果做排序。如果查询需要关联多张表，则只有order by子句引用的字段全部为第一张表时，才能使用索引做排序。Order by子句和查询的限制是一样的，都要满足最左前缀的要求，其他情况下都需要执行排序操作，而无法利用索引排序。
7. 冗余索引是指在相同的列上按照相同的顺序创建的相同类型的索引，应当避免这种索引，发现后立即删除。大多数情况下都应该尽量扩展已有的索引列而不是创建新索引。但有极少情况下出于性能方面的考虑需要冗余索引，比如扩展已有索引而导致其变得过大，从而影响到其他使用该索引的查询。
8. 定期删除一些长时间未使用过的索引是一个非常好的习惯。索引不总是最好的工具，只有当索引帮助提高查询速度带来的好处大于其带来的额外工作时，索引才是有效的。对于非常小的表，简单的全表扫描更高效。对于中到大型的表，索引就非常有效。对于超大型表，建立和维护索引的代价随之增长，这时候其他技术也许更有效，比如分库分表。
9. 特定查询类型优化
10. 优化count（）查询
11. Count（）有两种不同的作用，其一时统计某个列值的数量，其二是统计行数。统计列值时，要求列值是非空，它不会统计null。如果确认括号中的表达式不可能为空时，实际上就是在统计行数。
12. 有时候某些业务场景不需要安全精确的count值，可以用近似值来代替，explain出来的行数就是一个不错的近似值，而且执行explain并不需要真正的去执行查询，所有成本非常低。执行count（）都需要扫描大量的行才能获取到精准的值，因此很难优化，mysql层面还能做到得也就只有覆盖索引了。如果还是不能解决，只有从架构层面解决了，比如添加汇总表，或者使用redis这样得外部缓存系统。
13. 优化关联查询
14. Mysql的关联查询是嵌套循环关联操作，即先在一个表中循环取出单条数据，然后再嵌套循环到下一个表中寻找匹配的行，依次下去，直到找到所有表中匹配的行为止。然后根据各个表匹配的行，返回查询中需要的各个列。因此只需要再关联顺序中的第二张表的相应列上创建索引即可
15. 确保on和using字句中的列上有索引。
16. 确保group by和order by中的表达式只涉及到一个表中的列，这样mysql才有可能使用索引来优化。
17. 优化limit分页
18. 当需要分页操作时，通常会使用limit加上偏移量的办法实现，同时加上适合的order by字句。如果有对应得索引，通常效率会非常不错，否则myslq需要做大量得文件排序
19. 优化这种查询一个最简单得办法就是尽可能得使用覆盖索引扫描，而不是查询所有得列。然后根据需要做一次关联查询再返回所有得列。对于偏移量很大时，这样做得效率会提升非常大。
20. 有时候如果可以使用书签记录上次取数据得位置，那么下次就可以直接从该书签记录得位置开始扫描。
21. 其他优化办法还包括使用预先计算的汇总表，或者关联到一个冗余表，冗余表中只包含主键列和需要做排序的列。
22. 优化union
23. Mysql处理union的策略是先创建临时表，然后再把各个查询结果插入到临时表中，最后再来做查询，因此union查询中很难有优化策略。经常需要手动将where、limit、order by等字句下推到各个子查询中，以便优化器可以充分利用这些条件先优化。
24. 除非确实需要服务器去重，否则就一定要用union all，如果没有all关键字，mysql会给临时表加上distinct选项，这会导致整个临时表的数据做唯一性检查，这样做的代价非常高。
25. **的执行过程**