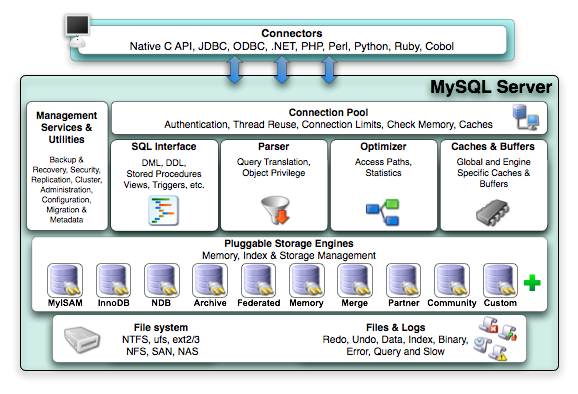
MySQL是一个开源的关系数据库管理系统。以前的MySQL不支持视图，存储过程，触发器等复杂功能，所以满足不了企业的性能要求；但自从5.0版本推出后，MySQL相继推出复制，集群，分区，分布式等高级特性，使其完全可以适应企业级的应用环境，而相较其他数据库，MySQL最大的优势还是灵活性，其可高度定制化地设置参数，能适应更多的业务需求。所以为了加深对于MySQL的理解与使用，便将最近所学习的相关知识做了总结，不足之处，敬请批评指正。

首先，PO出一张非常经典的结构图，让大家有个直观的感受：



乍一看，是不是还有点复杂，不要方，听我慢慢道来。

根据上图，我们大致可以将MySQL的架构分为四层：

第一层：线程/连接的处理，主要就是存在一些客户端和提供连接服务，说的通俗一点就是方便编程语言连接数据库，并且还提供了线程池，为客户端认证提供线程。

第二层：由SQL接口，查询缓存，解析器，优化器等组成，完成了MySQL大多数核心服务，例如是否查询缓存，解析查询，索引优化等操作。

第三层：存储引擎，真正负责数据的存取，服务器通过API与存储引擎通信，并且可以根据需求自由选择。

第四层：文件存储，将数据存储于设备的文件系统中，完成与存储引擎的交互，例如表文件，日志文件等都存储于此。

看完分层，是不是对MySQL的架构有一点模糊的概念了。很好，接下来我将对上面提到的一些知识点做进一步的分析，慢慢地为大家掀开MySQL的面纱…

第一层

由于第一层是封装接口，并通过通信协议(TCP…)与其连接，所以对这方面知识感兴趣的朋友，可查询相关资料进行了解，我在这里就不做赘述了。

第二层

接着，第二层又是在干嘛呢？给个场景，当第二层接收到一个select语句，MySQL首先会查询缓存，判断是否与该查询匹配，接着进行权限检查，若都能通过，就直接返回数据。当中间有任何一步出错，它都会经由解析器处理，交由查询优化器进行优化，最后返回结果集等一系列流程。而这条SQL的执行也就是第二层的大致功能了。下面就是我刚学习时有过的一些疑问：

1.什么是查询缓存？

答：缓存SELECT操作或预处理查询（5.1.17开始支持）的结果集和SQL语句， 新的SELECT语句或预处理查询语句，先去查询缓存，判断是否存在可用的记录集，判断标准：与缓存的SQL语句，是否完全一样，区分大小写。

2. 查询缓存的存取规则？

答：对于以下查询语句无法进行缓存：添加了SQL\_NO\_CACHE参数的语句，语句中含有特殊函数，对系统数据库(mysql)的查询，隔离级别为Serializable下的语句等都不会缓存其记录集。

3.查询缓存的配置：

答：query\_cache\_type：是否启用；

query\_cache\_size：缓存大小；

query\_cache\_limit：限制缓存的记录集；

query\_cache\_min\_res\_unit：分配的最小单位；

query\_cache\_wlock\_invalidate：对加锁内容的缓存策略

4.如何管理查询缓存？

答：FLUSH QUERY CACHE：缓存区的碎片整理；

RESET QUERY CACHE：清空查询缓存的数据

FLUSH TABLES：清空查询缓存的数据

5.什么是解析器？

答：其实跟其他编码语言的解析器一样，通过词法分析，语法分析生成相应的解析树，然后交由下一模块进行处理。

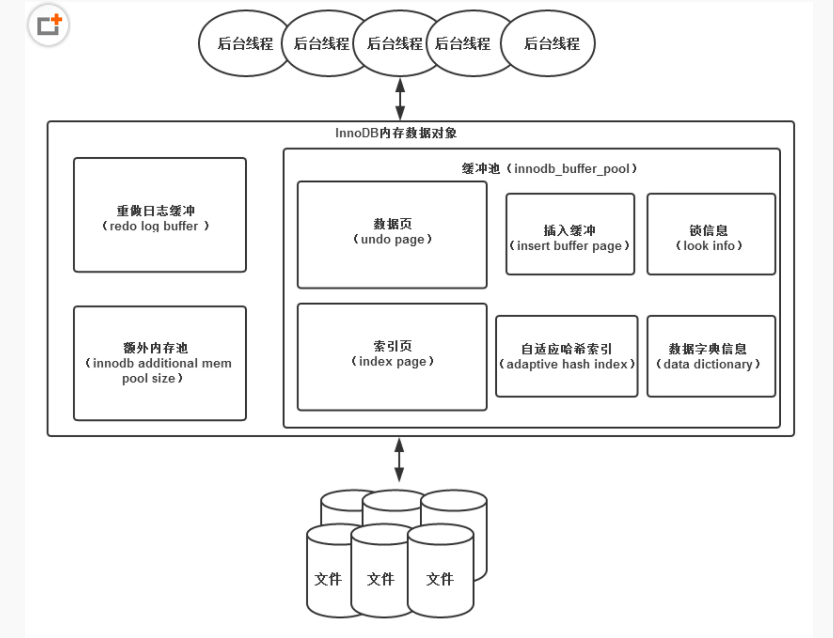
6.什么是优化器？

答：顾名思义，就是对解析后的SQL进行优化，我认为最主要的目的就是使用索引，通过使用条件最严格的索引来尽可能多且快地排除那些不符合索引条件的数据行，说通俗点就是选择怎样使用索引。另外，墙裂建议大家看一下explain关键字，它可以查看优化器的所作的操作，便于我们分析优化器到底做了何种优化，后面再做详解。

第三层

看完了第二层，是不是对MySQL又有了一点新的认识。很好，睁开眼睛，下面我将介绍MySQL最重要的东西——存储引擎。

这里，我以INNODB(默认引擎)为例进行介绍，照旧，先看图。



从上往下看，innodb由多个后台线程，内存数据对象以及相关文件组成。

它默认的后台线程有7组，分别是一个主线程，4组IO线程，1个锁线程，1个错误监控线程，在5.5之后，又新增了一个purge线程(其实是从主线程中分离出去的)。

\*主线程(master thread):主要负责脏页回刷，触发检查点，合并插入缓冲区等。

\*IO线程：

Insert buffer thread：负责插入缓冲区的合并操作

Read thread：负责数据库的读取操作

Write thread：负责数据库的写操作

Log thread:负责将重做日志刷新到logfile中

\*Purge线程：负责执行purge操作，而purge操作是指系统定期对DML操作(insert,update,delete)产生的undo页进行清理。那undo页又是什么呢？首先，我们得明白undo页是为回滚而用的，具体内容就是Copy事务前的数据库内容(行)到undo buffer,在适合的时间把undo buffer中的内容刷新到磁盘。

\*lock thread:负责锁控制和错误处理。

\*error monitor thread:主要负责错误监控和错误处理。

接下来，让我们看看innodb的内存的组成。

\*缓存池：不仅缓存索引块，还缓存数据块

Free list:空闲缓存块列表

Flush list:需要刷新到磁盘的缓存块列表

LRU list(核心):正在使用的缓存块

Index page(索引页):

Insert buffer(插入缓存)：

AHI(自适应哈希索引)：

\* Redo log buffer(重做日志缓冲): 记录innodb存储引擎的事务日志,并按照一定的频率刷新到文件中(redo log)。

\*额外内存池: 一些数据结构本身分配内存时，需要从额外内存池中分配。

最后就是文件(磁盘)，它主要存储表文件，日志等信息。

下面，我就一些重要的点进行分析(不然就太多了T.T)：

1. Insert buffer: 对于非聚簇索引的插入或更新操作，不是每一次直接插入到索引页中，而是先判断插入的非聚簇索引页是否在缓冲池中，若在，则直接插入； 若不在，则先放入到一个Insert  Buffer对象中，从表面上看，数据库这个非聚簇索引的已经插入到叶子节点中，但实际上没有，只是存放在了另一个位置。然后再以一定的频率和情况进行insert和辅助索引叶子节点的merge操作，这样就可以把多个插入操作合并到一个操作中，大大提高对非聚簇索引的插入性能。注意：它的使用应满足非聚集索引和索引非唯一两个条件。
2. LRU list flush list free list？
   1. LRU list全称为Latest Recent Used,意为最近最少使用的链表，说简单点就是缓存热度较高的数据。innodb将LRU list分为新子链与旧子链，当数据从磁盘中读出时，会将该缓存块插入到LRU list的中点，即旧子链的头部,经过一定时间的访问后，该数据块会从旧子链转移到新子链的头部，也就是整个LRU list 的头部。随着时间的推移，新子链和旧子链中较少被访问的缓存会从各自的头部逐渐向尾部移动；所以需要淘汰数据块时，优先从尾部淘汰，而被淘汰的数据会立刻放到Free list中。这样设计也避免了偶然访问的数据块淘汰掉访问频繁的热块。
   2. Free list：数据库刚启动的时候，LRU 列表为空，此时需要用到的时候直接将Free列表中的页删除，在LRU列表中增加相应的页，维持页数守恒。:
   3. Flush list: 当LRU列表中的页被修改后，称该页为脏页（dirty page），即缓冲池中的页和磁盘上的页数据产生了不一致。这时候数据库会通过checkpoint机制将脏页刷新回磁盘，而Flush 列表中的页即为脏页列表。注意脏页也存在于LRU列表中。
3. 检查点(checkpoint):
4. 解决的问题：

缩短数据库的恢复时间: 当数据库发生宕机时，数据库不需要重做所有的日志，因为Checkpoint之前的页都已经刷新回磁盘。数据库只需对Checkpoint后的重做日志进行恢复，这样就大大缩短了恢复的时间。同样，事务也是基于检查点的，这就是无论多大的事务，提交都会很快。(这里有很多问题需要分析，望自行了解)

当缓冲池不够用时，将脏页刷新到磁盘: 当缓冲池不够用时，根据LRU算法会溢出最近最少使用的页，若此页为脏页，那么需要强制执行Checkpoint，将脏页也就是页的新版本刷回磁盘。

当重做日志不可用时，刷新脏页: 当重做日志出现不可用时，因为当前事务数据库系统对重做日志的设计都是循环使用的，并不是让其无限增大的，重做日志可以被重用的部分是指这些重做日志已经不再需要，当数据库发生宕机时，数据库恢复操作不需要这部分的重做日志，因此这部分就可以被覆盖重用。如果重做日志还需要使用，那么必须强制Checkpoint，将缓冲池中的页至少刷新到当前重做日志的位置。

1. checkpoint的两种机制

sharp: 当数据库关闭时，将所有的脏页刷回磁盘，默认的工作方式。但数据库若还在使用时，采用这种机制，会大大降低数据库的可用性。

fuzzy: InnoDB存储引擎内部使用的方式，即只刷新一部分脏页。可以实现异步刷新，保证LRU list有空闲页以及redo log不可用时，强制刷新。

1. doublewrite:

在讲doublewrite前，我得先提一下redo log(重做日志),它是存储在磁盘上，对InnoDB存储引擎的事务操作进行备份的日志文件，防止宕机造成事务未正常提交而无法启动实例。但是若数据还未写到redo log就发生断电等异常情况时，就得需要doublewrite来提供这些数据恢复操作。

Doublewrite:在进行脏页刷新时，InnoDB采用双写策略，之所以这么做，是因为MySQL的数据页大小与操作系统的IO数据页大小不一致，无法保证缓冲中的数据能够完整，一致地刷新到磁盘。因此，InnoDB用系统表空间中一块连续磁盘空间(100个连续数据页，2MB)作为doublewrite buffer，在进行脏页刷新时，先将脏页的副本写到doublewrite buffer中，然后通过调用fsync()刷新到系统IO缓存中，确保副本被真正写入磁盘，然后InnoDB后台IO线程将脏页刷新到数据文件中。在恢复时，如果发现不一样的页，InnoDB会使用系统表空间doublewrite buffer区的相应副本来恢复数据页。

1. 自适应哈希索引(AHI)：

一个表几乎大部分都在缓冲池中，那么建立一个哈希索引能够加快查询，通过将btree的索引值转换成一个排序指针(需要了解索引的数据结构)。InnoDB有这个机制，可以监控索引的搜索情况，如果它注意到有些查询通过建立哈希索引可以优化查询，那么它会自动建立，所以说是自适应的。

以上，就是我对MySQL架构学习的大致历程，我给大家补充一下事务的相关知识

4.索引的优化,失效以及explain的使用：

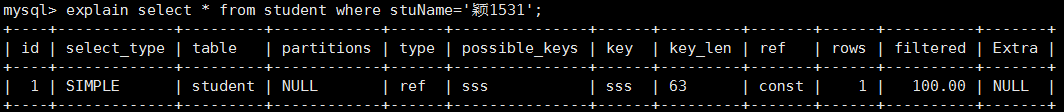
Explain输出列:



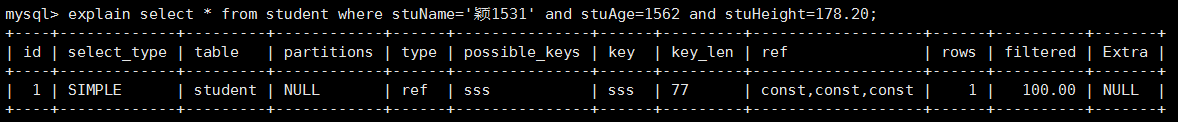
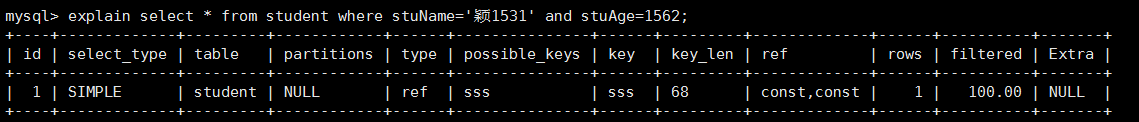
**这里我建了一个复合索引：**

*create index sss on student(stuName,stuAge,stuHeight);*

\***全值匹配**：当你建立了复合索引时，就尽量都写上，且按照顺序书写，可以加快SQL的执行速度。





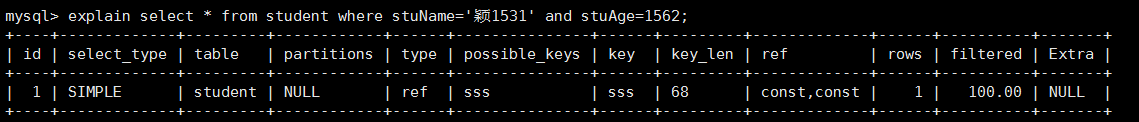




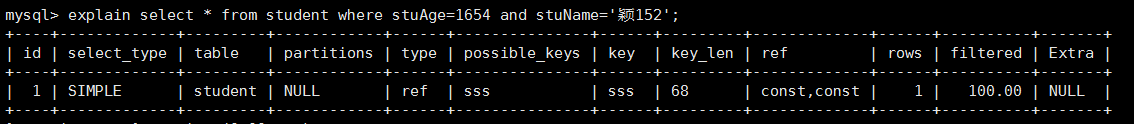
\***最佳左前缀**：当有多个索引时，应尽量把限定最强的放在左边（按照定义时的顺序），并且中间不要间隔使用非索引列。

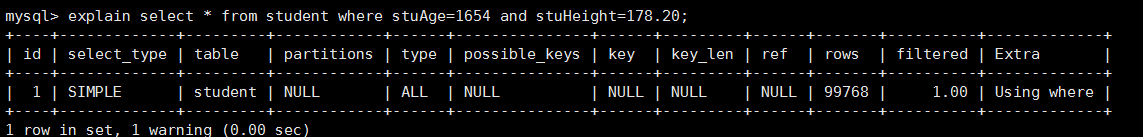


**@必须出现复合索引的首个字段**







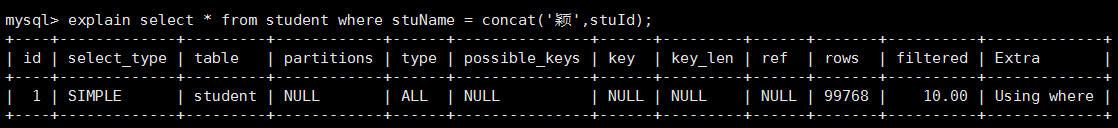


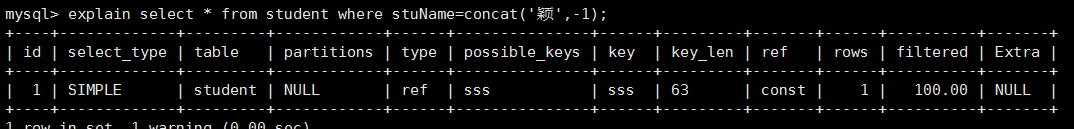


\*尽量避免在索引列上进行操作(计算，函数，转换类型)，不然会导致索引失效从而全表查找。

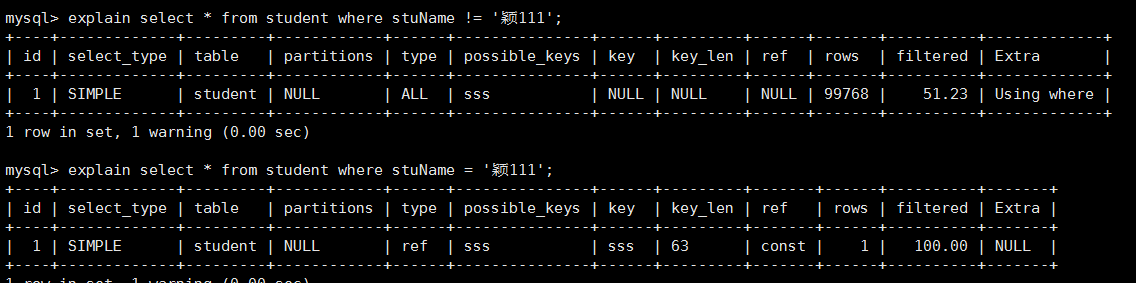
因为我在cancat()里写了一个stuId，进行了列之间的计算，导致索引失效，不是因为用了该函数。

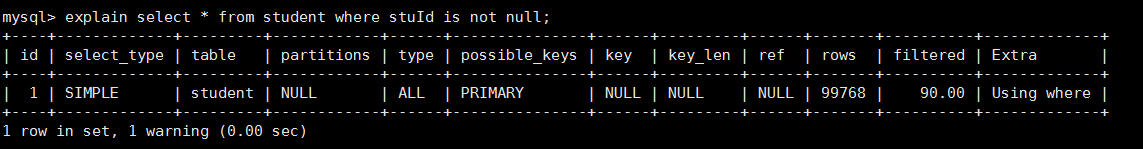






\*索引上使用！=，<>,is null,is not null会导致索引失效





\*使用通配符like会导致索引失效

