数据库优化方案整理

1. 一：优化说明
2. A：有数据表明，用户可以承受的最大等待时间为8秒。数据库优化策略有很多，设计初期，建立好的数据结构对于后期性能优化至关重要。因为数据库结构是系统的基石，基础打不好，使用各种优化策略，也不能达到很完美的效果。
3. B：数据库优化的几个方面
4. 这里写图片描述
5. ​​
6. 可以看出来，数据结构、SQL、索引是成本最低，且效果最好的优化手段。
7. C：性能优化是无止境的，当性能可以满足需求时即可，不要过度优化。
8. 二：优化方向
9. 1. SQL以及索引的优化
10. 首先要根据需求写出结构良好的SQL，然后根据SQL在表中建立有效的索引。但是如果索引太多，不但会影响写入的效率，对查询也有一定的影响。
11. 2. 合理的数据库是设计
12. 根据数据库三范式来进行表结构的设计。设计表结构时，就需要考虑如何设计才能更有效的查询。
13. 数据库三范式：
14. 第一范式：数据表中每个字段都必须是不可拆分的最小单元，也就是确保每一列的原子性；
15. 第二范式：满足一范式后，表中每一列必须有唯一性，都必须依赖于主键；
16. 第三范式：满足二范式后，表中的每一列只与主键直接相关而不是间接相关(外键也是直接相关)，字段没有冗余。
17. 注意：没有最好的设计，只有最合适的设计，所以不要过分注重理论。三范式可以作为一个基本依据，不要生搬硬套。
18. 有时候可以根据场景合理地反规范化：
19. A：分割表。
20. B：保留冗余字段。当两个或多个表在查询中经常需要连接时，可以在其中一个表上增加若干冗余的字段，以 避免表之间的连接过于频繁，一般在冗余列的数据不经常变动的情况下使用。
21. C：增加派生列。派生列是由表中的其它多个列的计算所得，增加派生列可以减少统计运算，在数据汇总时可以大大缩短运算时间。
22. 数据库五大约束：
23. A：PRIMARY key:设置主键约束；
24. B：UNIQUE：设置唯一性约束，不能有重复值；
25. C：DEFAULT 默认值约束
26. D：NOT NULL：设置非空约束，该字段不能为空；
27. E：FOREIGN key :设置外键约束。
28. 字段类型选择：
29. A：尽量使用TINYINT、SMALLINT、MEDIUM\_INT作为整数类型而非INT，如果非负则加上UNSIGNED
30. B：VARCHAR的长度只分配真正需要的空间
31. C：使用枚举或整数代替字符串类型
32. D：尽量使用TIMESTAMP而非DATETIME
33. E：单表不要有太多字段，建议在20以内
34. F：避免使用NULL字段，很难查询优化且占用额外索引空间
35. 3. 系统配置的优化
36. 例如：MySQL数据库my.cnf
37. 4. 硬件优化
38. 更快的IO、更多的内存。一般来说内存越大，对于数据库的操作越好。但是CPU多就不一定了，因为他并不会用到太多的CPU数量，有很多的查询都是单CPU。另外使用高的IO（SSD、RAID），但是IO并不能减少数据库锁的机制。所以说如果查询缓慢是因为数据库内部的一些锁引起的，那么硬件优化就没有什么意义。
39. 三：优化方案
40. 代码优化
41. 之所以把代码放到第一位，是因为这一点最容易引起技术人员的忽视。很多技术人员拿到一个性能优化的需求以后，言必称缓存、异步、JVM等。实际上，第一步就应该是分析相关的代码，找出相应的瓶颈，再来考虑具体的优化策略。有一些性能问题，完全是由于代码写的不合理，通过直接修改一下代码就能解决问题的，比如for循环次数过多、作了很多无谓的条件判断、相同逻辑重复多次等。
42. 举个栗子：
43. 一个update操作，先查询出entity，再执行update，这样无疑多了一次数据库交互。还有一个问题，update语句可能会操作一些无需更新的字段。
44. ​
45. 我们可以将表单中涉及到的属性，以及updateTime，updateUser等赋值到entity，直接通过pdateByPrimaryKeySelective，去update特定字段。
46. ​​
47. 定位慢SQL，并优化
48. 这是最常用、每一个技术人员都应该掌握基本的SQL调优手段（包括方法、工具、辅助系统等）。这里以MySQL为例，最常见的方式是，由自带的慢查询日志或者开源的慢查询系统定位到具体的出问题的SQL，然后使用explain、profile等工具来逐步调优，最后经过测试达到效果后上线。
49. SqlServer执行计划：
50. 通过执行计划，我们能得到哪些信息：
51. A：哪些步骤花费的成本比较高
52. B：哪些步骤产生的数据量多，数据量的多少用线条的粗细表示，很直观
53. C：每一步执行了什么动作
54. 具体优化手段：
55. A：尽量少用（或者不用）sqlserver 自带的函数
56. select id from t where substring(name,1,3) = ’abc’
57. select id from t where datediff(day,createdate,’2005-11-30′) = 0
58. 可以这样查询：
59. select id from t where name like ‘abc%’
60. select id from t where createdate >= ‘2005-11-30’ and createdate < ‘2005-12-1’
61. B：连续数值条件，用BETWEEN不用IN：SELECT id FROM t WHERE num BETWEEN 1 AND 5
62. C：Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗
63. D：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
64. E：不建议使用 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。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
65. F：表与表之间通过一个冗余字段来关联，要比直接使用JOIN有更好的性能
66. G：select count(\*) from table；这样不带任何条件的count会引起全表扫描
67. 连接池调优
68. 我们的应用为了实现数据库连接的高效获取、对数据库连接的限流等目的，通常会采用连接池类的方案，即每一个应用节点都管理了一个到各个数据库的连接池。随着业务访问量或者数据量的增长，原有的连接池参数可能不能很好地满足需求，这个时候就需要结合当前使用连接池的原理、具体的连接池监控数据和当前的业务量作一个综合的判断，通过反复的几次调试得到最终的调优参数。
69. ​​
70. 合理使用索引
71. 索引一般情况下都是高效的。但是由于索引是以空间换时间的一种策略，索引本身在提高查询效率的同时会影响插入、更新、删除的效率，频繁写的表不宜建索引。
72. 选择合适的索引列，选择在where，group by，order by，on从句中出现的列作为索引项，对于离散度不大的列没有必要创建索引。
73. 主键已经是索引了，所以primay key 的主键不用再设置unique唯一索引
74. 索引类型
75. 主键索引 （PRIMARY KEY）
76. 唯一索引 （UNIQUE）
77. 普通索引 （INDEX）
78. 组合索引 （INDEX）
79. 全文索引 （FULLTEXT）
80. 可以应用索引的操作符
81. 大于等于
82. Between
83. IN
84. LIKE 不以 % 开头
85. 不能应用索引的操作符
86. NOT IN
87. LIKE %\_ 开头
88. 如何选择索引字段
89. A：字段出现在查询条件中，并且查询条件可以使用索引
90. B：通常对数字的索引和检索要比对字符串的索引和检索效率更高
91. C：语句执行频率高，一天会有几千次以上
92. D：通过字段条件可筛选的记录集很小
93. ​​
94. 无效索引
95. A：尽量不要在 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
96. B：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。
97. C：应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描
98. select id from t where num=10 or Name = ‘admin’
99. 可以这样查询：
100. select id from t where num = 10
101. union
102. select id from t where Name = ‘admin’
103. union all 返回所有数据，不管是不是重复。 union会自动压缩，去除重复数据。
104. D：不做列运算
105. where age + 1 = 10，任何对列的操作都将导致表扫描，它包括数据库教程函数、计算表达式等
106. E：查询like，如果是 ‘%aaa’ 不会使用到索引
107. 分表
108. 分表方式
109. 水平分割（按行）、垂直分割(按列)
110. 分表场景
111. A： 根据经验，mysql表数据一般达到百万级别，查询效率就会很低。
112. B： 一张表的某些字段值比较大并且很少使用。可以将这些字段隔离成单独一张表，通过外键关联，例如考试成绩，我们通常关注分数，不关注考试详情。
113. 水平分表策略
114. 按时间分表：当数据有很强的实效性，例如微博的数据，可以按月分割。
115. 按区间分表：例如用户表 1到一百万用一张表，一百万到两百万用一张表。
116. hash分表：通过一个原始目标id或者是名称按照一定的hash算法计算出数据存储的表名。
117. 读写分离
118. 当一台服务器不能满足需求时，采用读写分离【写: update/delete/add】的方式进行集群。
119. 一台数据库支持最大连接数是有限的，如果用户的并发访问很多，一台服务器无法满足需求，可以集群处理。mysql集群处理技术最常用的就是读写分离。
120. 主从同步：数据库最终会把数据持久化到磁盘，集群必须确保每个数据库服务器的数据是一致的。从库读主库写，从库从主库上同步数据。
121. 读写分离：使用负载均衡实现，写操作都往主库上写，读操作往从服务器上读。
122. 缓存
123. 缓存分类
124. 本地缓存：HashMap/ConcurrentHashMap、Ehcache、Guava Cache等
125. 缓存服务：Redis/Tair/Memcache等
126. 使用场景
127. 短时间内相同数据重复查询多次且数据更新不频繁，这个时候可以选择先从缓存查询，查询不到再从数据库加载并回设到缓存的方式。此种场景较适合用单机缓存。
128. 高并发查询热点数据，后端数据库不堪重负，可以用缓存来扛。
129. 缓存作用
130. 减轻数据库的压力，减少访问时间。
131. 缓存选择
132. 如果数据量小，并且不会频繁地增长又清空（这会导致频繁地垃圾回收），那么可以选择本地缓存。具体的话，如果需要一些策略的支持（比如缓存满的逐出策略），可以考虑Ehcache；如不需要，可以考虑HashMap；如需要考虑多线程并发的场景，可以考虑ConcurentHashMap。
133. 其他情况，可以考虑缓存服务。目前从资源的投入度、可运维性、是否能动态扩容以及配套设施来考虑，我们优先考虑Tair。除非目前Tair还不能支持的场合（比如分布式锁、Hash类型的value），我们考虑用Redis。
134. 缓存穿透
135. 一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比
136. 如DB）。如果key对应的value是一定不存在的，并且对该key并发请求量很大，就会对后端系统造
137. 成很大的压力。这就叫做缓存穿透。
138. 对查询结果为空的情况也进行缓存，缓存时间设置短点，或者该key对应的数据insert了之后清理缓存。
139. 缓存并发
140. 有时候如果网站并发访问高，一个缓存如果失效，可能出现多个进程同时查询DB，同时设置缓存的情况，
141. 如果并发确实很大，这也可能造成DB压力过大，还有缓存频繁更新的问题。
142. 对缓存查询加锁，如果KEY不存在，就加锁，然后查DB入缓存，然后解锁；其他进程如果发现有锁就
143. 等待，然后等解锁后返回数据或者进入DB查询。
144. 缓存雪崩(失效)
145. 当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)
146. 带来很大压力。
147. 不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀.
148. 防止缓存空间不够用
149. ① 给缓存服务，选择合适的缓存逐出算法，比如最常见的LRU。
150. ② 针对当前设置的容量，设置适当的警戒值，比如10G的缓存，当缓存数据达到8G的时候，就开始发出报警，提前排查问题或者扩容。
151. ③ 给一些没有必要长期保存的key，尽量设置过期时间。
152. 我们看下图，在WebServer（Dao层）和DB之间加一层cache，这层cache一般选取的介质是内存，因为我们都知道存入数据库的数据都具有持久化的特点，那么读写会有磁盘IO的操作，内存的读写速度远比磁盘快得多。（选用存储介质，提高访问速度：内存>>磁盘；减少磁盘IO的操作，减少重复查询，提高吞吐量）
153. 这里写图片描述
154. ​​
155. 常用开源的缓存工具有：ehcache、memcache、redis。
156. ehcache 是一个纯Java的进程内缓存框架，hibernate使用其做二级缓存。同时，ehcache可以通过多播的方式实现集群。本人主要用于本地的缓存，数据库上层的缓存。
157. memcache是一套分布式的高速缓存系统，提供key-value这样简单的数据储存，可充分利用CPU多核，无持久化功能。在做web集群中可以用做session共享，页面对象缓存。
158. redis高性能的key-value系统，提供丰富的数据类型，单核CPU有抗并发能力，有持久化和主从复制的功能。本人主要使用redis的redis sentinel，根据不同业务分为多组。
159. redis注意事项
160. A：在增加 key 的时候尽量设置过期时间，不然 Redis Server 的内存使用会达到系统物理内存的最大值，导致 Redis 使用 VM 降低系统性能；
161. B：Redis Key 设计时应该尽可能短,Value 尽量不要使用复杂对象；
162. C：将对象转换成 JSON 对象（利用现成的 JSON 库）后存入 Redis；
163. D：将对象转换成 Google 开源二进制协议对象（Google Protobuf，和 JSON 数据格式类似，但是因为是二进制表现，所以性能效率以及空间占用都比 JSON 要小；缺点是 Protobuf 的学习曲线比 JSON 大得多）；
164. E：Redis 使用完以后一定要释放连接。
165. 读取缓存中是否有相关数据，如果缓存中有相关数据，则直接返回，这就是所谓的数据命中“hit”
166. 如果缓存中没有相关数据，则从数据库读取相关数据，放入缓存中，再返回。这就是所谓的数据未命中“miss”
167. 缓存的命中率 = 命中缓存请求个数/总缓存访问请求个数 = hit/(hit+miss)
168. ​​
169. NoSQL
170. 与缓存的区别
171. 先说明一下，这里介绍的和缓存不一样，虽然redis等也可以用来做数据存储方案（比如Redis或者Tair），但NoSql是把它作为DB来用。如果当作DB来用，需要有效保证数据存储方案的可用性、可靠性。
172. 使用场景
173. 需要结合具体的业务场景，看这块业务涉及的数据是否适合用NoSQL来存储，对数据的操作方式是否适合用NoSQL的方式来操作，或者是否需要用到NoSQL的一些额外特性（比如原子加减等）。
174. 如果业务数据不需要和其他数据作关联，不需要事务或者外键之类的支持，而且有可能写入会异常频繁，这个时候就比较适合用NoSQL（比如HBase）。
175. 比如，美团点评内部有一个对exception做的监控系统，如果在应用系统发生严重故障的时候，可能会短时间产生大量exception数据，这个时候如果选用MySQL，会造成MySQL的瞬间写压力飙升，容易导致MySQL服务器的性能急剧恶化以及主从同步延迟之类的问题，这种场景就比较适合用Hbase类似的NoSQL来存储。
176. 视图/存储过程
177. 普通业务逻辑尽量不要使用存储过程，定时任务或报表统计函数可以根据团队资源情况采用存储过程处理。
178. GVM调优
179. 通过监控系统（如没有现成的系统，自己做一个简单的上报监控的系统也很容易）上对一些机器关键指标（gc time、gc count、各个分代的内存大小变化、机器的Load值与CPU使用率、JVM的线程数等）的监控报警，也可以看gc log和jstat等命令的输出，再结合线上JVM进程服务的一些关键接口的性能数据和请求体验，基本上就能定位出当前的JVM是否有问题，以及是否需要调优。
180. 异步/多线程
181. 针对某些客户端的请求，在服务端可能需要针对这些请求做一些附属的事情，这些事情其实用户并不关心或者用户不需要立即拿到这些事情的处理结果，这种情况就比较适合用异步的方式处理这些事情。
182. 异步作用
183. A：缩短接口响应时间，使用户的请求快速返回，用户体验更好。
184. B：避免线程长时间处于运行状态，这样会引起服务线程池的可用线程长时间不够用，进而引起线程池任务队列长度增大，从而阻塞更多请求任务，使得更多请求得不到技术处理。
185. C：线程长时间处于运行状态，可能还会引起系统Load、CPU使用率、机器整体性能下降等一系列问题，甚至引发雪崩。异步的思路可以在不增加机器数和CPU数的情况下，有效解决这个问题。
186. 异步实现
187. A：额外开辟线程，这里可以采用额外开辟一个线程或者使用线程池的做法，在IO线程（处理请求响应）之外的线程来处理相应的任务，在IO线程中让response先返回。
188. B：使用消息队列（MQ）中间件服务
189. 搜索引擎
190. 例如：solr，elasticsearch