应用系统体系架构 2021 至 2022

1. MongoDB 将 collection 分为小的 chunks 称为 shards
   数据库会为每个分片的 collection 创建一个 router 存储分片的元数据
   当添加或减少 shard 时 MongoDB 会自动迁移数据 使得数据均匀分布
   shard 是一个服务器节点上的分片 一个 shard 包含多个 chunk MongoDB 会限制每个 shard 之间的 chunk 数量不能超过一定的阈值
   用户选择用来分片的字段称为 shard key 按照 shard key 的值范围 数据会分为不同 chunk 当一个 chunk 的数据量超过阈值时 MongoDB 会自动分裂这个 chunk 并且把新的 chunk 迁移到其他 shard 上
   新添加一个 shard 时 MongoDB 会把一部分 chunk 迁移到新的 shard 上 删除同理
   有时不希望数据均匀存储 可以关闭 chunk 数量的限制 改用自己写的 balancer 保证数据访问的均衡
2. 空间放大是指数据库中存储数据占用的空间比实际数据量大很多 lsmt 数据库并不存在这种问题 因为额外的存储只有一些索引数据和已经被删除的数据 lsmt 数据库通常使用定期 gc 和合并来清除无用的数据
   写放大是指磁盘的 IO 操作比实际数据量大很多 lsmt 数据库会有写放大的问题 因为在某些操作触发多次 compaction 时会产生大量的 IO 操作
3. 使用 docker volume create 命令创建卷 并使用–mount source=xxx,target=xxx 来将创建好的卷挂载到容器中 Docker 会自动管理卷的存储位置 通常用于无需访问数据的持久化 比如数据库
4. mapper 在处理 split 时 会因为缓冲区不够大而进一步切分 split 然后处理一部分就写入一部分结果到磁盘 最后再合并结果 reducer 也是一样的问题 这就导致 IO 次数太多 速度受到限制 解决方法是在内存里处理数据
   另外 mapreduce 本身的中间结果文件也需要大量 IO
5. spark 中对于 rdd 的操作分为两种 transformation 和 action
   所有的 Transformation 都是 lazy 的 遇到 Action 时必须执行
   除此之外 Transformation 会被划分为 stage 其边界处必须执行
   stage 的划分有两个依据 遇到宽依赖的 shuffle 操作或是产生新的已缓存的分区
   因此可以减少内存占用 利用了窄依赖的高速计算 将操作尽可能推迟到宽依赖时才执行 使得窄依赖产生的 RDD 可以尽可能晚产生
6. hdfs 有机架策略 即其中至少有一个副本在一个不同的机架上 一般而言 hdfs 尽量寻找最近的副本 比如先找同一个机器的 再找同一个机架的 再找其他机架的
7. 列族使得数据变为半结构化的 存储比较灵活 支持列的动态增加 增强了单个表的表现能力 形成一个大表 另外通过列族相当于进行了垂直分表 支持了行列混合存储 动态应对不同场景的数据存储需求 比如 OLAP OLTP

应用系统体系架构 2022 至 2023 A 卷

一

1. 首先 数据库连接池的大小和用户数并无直接关系 数据库连接池指的是 mysql 提供多少连接给服务器的后端应用 理想情况下 连接占满了 CPU 核心与硬盘的连接是最快的 因此数据库连接池=核心数*2+有效硬盘数
   其次 过大的数据库连接池会让 mysql 花费大量的开销在维护连接池中的实例状态 以及上锁保证并发安全上 导致性能反而大大降低
2. 事务是用于保证操作原子性的 然而没有事务不一定意味着系统一定会出现异常 比如系统所有方法都正常的执行没有出现错误 自然每一个方法也就维持了原子性 并不会出现事务方面的异常

3. payOrder 可以不设置事务 视情况而定
   pay 与 shipping 都设置 progation 为 requirednew 使得无论如何都会新建一个独立事务
4. 第一 消息队列会比同步处理需要更多的开销 如果访问数量超出处理能力限制 只会导致服务器的负载更大 如果遇到这种情况 增加硬件性能
   第二 消息队列相比于同步处理 需要在后台异步返回操作的真正结果 这需要额外的代码逻辑与通信开销 一般可以使用 websocket 实现 让处理结果通过 ws 发送给前端
5. 无需对对象使用 sync java 中每个对象都具有一个锁 在 sync 修饰的代码块中 所有对象的锁会被获取 无需显式标明
6. 不合理 存在两种场景的问题
   第一 高并发场景下 订单放入 redis 后 放入数据库前的状态是不一致的 查询会查出这个订单 然而其并不存在于数据库中
   第二 容灾场景下 存入 redis 后 放入数据库前若服务器崩溃重启 redis 中订单仍存在 但数据库中不存在 导致查询一个不存在于数据库的订单
   一般为了保证一致性 都是先写入数据库 再写入缓存 就算在中间状态查询 也只会因为 redis 里不存在而把数据提取到 redis 中 只有性能影响而没有数据一致性问题
7. 单点认证是指用户只需要登录一次就可以访问所有的应用 在单台服务器上 session 足以做到 但是在分布式集群中 因为会话粘滞的问题 通常使用 jwt 来实现单点认证
   具体而言 jwt 是服务器根据用户信息生成的一个 token 发回给客户端 维护在客户端的 cookie 中 服务器通过验证这个 token 本身的正确和完整性来验证用户的身份 而无需维护任何状态 使得可以在多个服务器上实现单点认证
8. mysql 的全文索引是通过倒排索引的结构实现的 即从 term 到 document 的映射 维护一个 term 在哪些 doc 中出现过 当我们需要搜索某一个 term 时 只需要在倒排索引中找到这个 term 对应的 doc list 然后进行相关度排序返回即可
9. 注册中心会维护各个服务的 ip 与端口 当 gateway 需要转发给某种服务时 只需要在注册中心中寻找一个服务的实例即可 主要实现了服务实例与服务器实体间的解耦 隐藏了服务到对应 url 的映射 使得服务的调用更加简单
10. 并非正确 restful 风格的服务是通过 uri 定位资源 通过 http 请求方法来表明对资源的操作 使用注解并不代表其 api 是 restful 风格的 比如对于一本书的增加可以用 post 方法 但是如果使用 get 方法 参数全部写在 requestparam 中也可以实现 但是这样就不是 restful 风格的

二

1. 无损连接分解是指将一个表分解为多张表后 可以通过 join 得到一模一样的原表 目的是保证数据的完整性 使得查询更加高效 并且减少数据冗余
   最小化重复信息是指尽可能减少数据的冗余 一般是为了节省存储空间 并且保证数据的一致性
2. 缓冲池多实例主要是为了提高并发性能 建立多实例后 同一张表可以存储在不同实例中 进行并发的读写操作
3. char 类型是定长字符串 占用空间固定 varchar 则是变长 可以压缩存储
   使用 char 类型的好处是不需要维护额外的长度信息 并且查询时更快？适用于数据长度固定的场景 比如 id
   缺点是无法节省空间 并且对于变长数据会浪费空间
4. 物理备份是指将整个数据库的数据文件进行拷贝 逻辑备份是指记录数据库的所有操作逻辑 通常而言逻辑操作的备份空间更大 因为要记录很多语句 并且恢复速度也是物理备份快 只需要拷贝文件即可 逻辑备份则需要逐条执行语句
   缺点是适用范围小 通常不同数据库 甚至不同版本的数据库都不通用一个物理备份 而逻辑备份则可以

5. 分区除了扩展空间以外 还可以带来各种读写性能的提高 比如按照时间分区数据 当我需要按照时间范围查询时 mysql 会通过分区进行剪枝 减少了查询涉及的数据量 相当于初步的索引
   另外 分区表也可以用来维护父子数据 比如各个地区的数据 在统一维护时可以通过 mysql 的 exchange partition 来实现
6. 看情况 对于电子书店 有些数据适合 MongoDB 存储 比如书籍的评论回复 但是某些需要双向查询的数据 比如订单与书籍 需要在两边都维护外键关系 这样就不适合 并且 MongoDB 也不是万能的 事务更加复杂 而且也不能自动维护数据的一致性 当需要维护一致性时 MongoDB 需要手动找到所有的引用并且更新
7. influxdb 中 有两类字段 fields 和 tags 其中 fields 不参与索引 这告诉我们 如果需要在某一个列上进行查询 把它设为 tags 而不是 fields 因此不会影响数据访问的效率 fields 在语义上一般就是用于数据分析 需要全表扫描的字段
8. 是的 负载均衡会使得请求以一定比例转发给不同的实例 比如 1:3
9. 如果数据在写入时是以 raw 形式存储的 在进行数据分析时 需要先对数据进行清洗 以便提高数据的质量 使得数据分析更加准确 然后再进行转换与加载 确保数据最终以合适的格式存储到数据仓库中 方便进行分析操作 不进行处理的数据往往是非结构化的 难以进行分析
10. 在 worker 本地对 Map 的输出进行合并 比如(map,2),(map,3)合并为(map,5) 从而减少传输的数据量 注意不是所有任务都适合 Combine 比如求平均值就不能
    combine 与 reduce 的区别在于 前者是在 mapper 的本地上进行的 目的是减少传输的数据 后者是在 reduce 上进行的 目的是得到最终的结果

应用系统体系架构 2022 至 2023 B 卷

一

1. 正确
2. 影响不大 因为 log 是顺序写
3. 同 A 改为 required payorder 也需要改为 required
4. 错误 ajax 异步发送时对于前端响应速度的优化 和前端发送请求给后端没有关系哈 比如一个请求 通过 ajax 发送 可以使得页面不会阻塞 但是请求的结果仍要等待后端处理结束 消息队列解决的就是这个问题
5. 是指一个线程可以多次获取一个锁 用于递归与嵌套调用 防止死锁
6. 不合理 会导致数据不一致
7. 是的 尽管有 RSA 加密 如果传输被人截断 还是可能产生中间人攻击 ssl 保证了传输的安全
8. tf-idf 算法对于搜索结果文档进行了相关性评分 具体而言 关键词出现的越多 并且关键词本身在所有文档中出现的越少 说明这个文档的相关性越高 鉴于我不知道支持度和置信度是什么东西 回答不了
9. 不是 但是注册中心有很大的作用 如果没有 编码非常麻烦 gateway 需要维护所有服务的地址
10. 否 见 A

二

1. BCNF 虽然保证了更强的数据完整性 但是会导致表的拆分过于复杂 在实际考虑中 通常做权衡 第三范式已经足够了
2. 变种 LRU 策略 根据数据的热度来替换 会把新数据放到 3/8 的队列位置 队里的数据被访问时则会被放到队尾 如果是冷数据则会迅速的被挤到队头移出 还会做一定的数据迁移 把热数据放到其他地方
3. 同上 varchar 适合存储变长数据 节省空间 但是有额外的长度信息 查询更慢

4. 使用 flush logs 指令将 binlog 写入磁盘并切换到下一个 binlog 文件 恢复数据库时 先进行全量恢复 然后找到对应时间的 binlog 文件进行重放直到恢复到指定时间点
5. 子分区支持在分区里继续分区 可以使用不同的字段和策略 比如按照时间分区后再按照地区分区 优化细粒度的查询
6. 基于 R 树 快速查询空间数据
7. 见 A
8. 多主多从相对于一主多从 需要额外的开销确保多个主数据库的数据一致性 写入性能会更差 包括高并发 2pc 通信开销等
9. 数据分析 即 OLAP 的场景下 通常需要对一个列的数据进行聚合操作 因此数据湖中存储面向列存的数据更加适合进行数据分析 并且列存还支持更多的压缩算法
10. map 类 这个操作把一个 list 变为另一个 list 符合 map 的语义 另外 过滤操作尽早的进行 可以减少传输的数据以及 reduce 需要处理的数据量 减少无谓的计算

课上卷 A

一

1. 连接池=核心数*2+有效硬盘数=16*2+2=34 连接池本质上是线程 一个核通过超线程可以处理两个线程 因此 16*2=32 另外磁盘还需要异步的读写操作 因此需要 2 个连接（也即是说 CPU 在运行时 磁盘 IO 是在空转的）
2. 合理 通常维护一个 websocket 的连接需要的开销比较大 包括心跳状态的维护等 因此用户通过 ws 接收到结果后 可以删除此会话以减少服务器的负担
3. undolog 可以是逻辑日志或是逻辑物理日志 不可以是物理日志 因为 undolog 要求操作可逆性 即逆向执行日志可以撤销操作 物理日志不满足 因为数据页的偏移可能已经被修改 而物理逻辑日志可以通过页面前后指针来完成回滚 逻辑日志显然可以通过执行语句来回滚
4. 优点是保持了数据的一致性 即使在中间状态查询也不会出现数据不一致的情况 因为都会到数据库中查询数据 缺点是性能会有所下降 因为需要查询数据库 并且重新加载数据到缓存中
5. 当等待锁的时间非常短 以至于小于线程切换的开销时 使用自旋锁比 wait 更加高效 因为 wait 会导致线程切换的开销
6. RESTful 的风格是完全数据驱动的 无法细化描述用户的需求 比如只取出数据的某些字段 而 GraphQL 可以根据用户的需求返回不同的数据结构 更加细粒度的控制数据的返回
7. 微服务架构确实会影响系统性能 最主要的原因就是微服务架构中各个服务之间的通信从进程间通信变为了网络通信 还需要额外的序列化和反序列化的开销 以及网络传输的开销 但是使用微服务架构往往是一种权衡或是无奈之选 牺牲性能换取更好的可维护性和可扩展性 或者当单个服务器已经不足以支持业务需求时 必须扩展为分布式的微服务架构 另外 通过消息队列等异步通信可以减缓微服务架构的性能问题

二

1. 首先 使用 varchar 存储 base64 编码的图片数据是不合理的 因为图片数据本身较大 base64 编码后会更大 而 varchar 类型是直接存储在数据库中的 当一个数据行超过半页时 查询效率会非常低 甚至因此导致数据行 size 的限制 很有可能根本无法写入 更好的方法是使用 blob 类型存储图片数据 其次 是否需要专门用 MongoDB 存储图片数据 答案是不一定的 mysql 的 blob 也比较适合存储图片数据 但是当数据量非常大时 关系型数据库不再是最优的使用场景 可以考虑使用 MongoDB 存储图片数据 其对于大数据量的存储和查询有更好的优化
2. 以前考过 为了并发

3. 上课讲过 innodb 存储数据有限制 包括
   在声明 MySQL 中一张表的字段类型时 一行数据最多 65535 字节 其中 char 是严格按照声明的长度存储的
   在 InnoDB 引擎的实际存储中 一行数据最多存储不超过 innodb_page_size 的一半（实际会略小于一半）
   声明时 VARCHAR 类型的长度可以超过 innodb_page_size 的一半 因为不知道实际存储的数据大小 CHAR 类型由于是定长 声明多少就会占用多少空间 所以总共加起来不能超过 innodb_page_size 的一半
   第一条语句全部是 char 一个数据行已经大于了半页的大小了 因此报错 第二条语句是一个 varchar 其需要额外的长度信息 2byte 刚好 65535 字节 因此可以声明
4. 分区除了扩展空间以外 主要为了查询性能 我们可以根据查询场景进行合适的分区 比如经常需要查询最新的订单 可以按照时间分区 经常需要查询某个地区的订单 可以按照地区分区 这样可以减少查询的数据量 提高查询的效率

课上卷 B

一

1. 见 A 一样
2. 合理的 既然需要保持通信 那么就需要维护 ws 连接 相应的增加一些开销
3. redolog 需要幂等性 重复执行不会产生不同的结果 逻辑日志不满足
   失败可重做性 日志执行失败后 可以通过重做达成恢复 逻辑日志不满足 因为一条逻辑记录可能对应多项数据修改 比如数据表和索引 因此必须使用物理日志
4. 优点 分表之后 多读少写的数据放入了 redis 提高了读取的速度 并且完全不会有写入的开销 缺点是在返回完整的 book 数据时 需要同时查询两边的数据并进行 join 有一定的性能开销
5. 应对多读少写的场景 只有在修改数据时才进行拷贝
   应对低并发且需要保持数据库原子性的场景 在低并发情况下 不会有多个请求同时对一个对象进行写 因此可以通过读写分离进行原子性的写 另外 cow 也可以用于快照备份 传统的快照备份需要拷贝整个数据库的状态 然而 cow 技术可以只保存指针 并只为修改过的数据创建副本 减少了存储空间
6. 同 A 需要细粒度的控制数据的返回时
7. 如果数据量不大 不需要扩展 并且也不需要高容错性 与高并发性 或者磁盘的 IO 能力足以应对需求 那么是可以的 反之如果数据量大 需要高容错性 高并发性 那么就需要考虑使用分布式数据库 或是主从复制的数据库集群

二

1. 针对不同场景 见 A 通常数据量较大适用于 MongoDB
2. 把新数据放到 3/8 的队列位置 队里的数据被访问时则会被放到队尾 如果是冷数据则会迅速的被挤到队头移出 针对需要频繁访问热数据 数据访问不均匀的场景 解决了 LRU 策略的缺点 即冷数据在缓存中占用太久
3. 见 A 第二条语句是 TEXT 在数据库中只存指针 因此声明时只占用 8byte 显然可以声明
4. 见 A 根据查询场景进行合适的分区即可

1

缓存过期是指缓存数据在一定时间后失效，无法继续提供有效的内容，系统必须重新从数据源加载或计算数据。缓存过期机制在系统设计中有着重要意义，它直接影响系统的性能、数据的实时性和一致性等多个方面。

1. 提高系统性能

- **减少对数据库或其他服务的访问**：缓存的核心目的是提高数据访问速度，缓存过期机制能够确保缓存的数据在合理的时间内使用，避免缓存数据过期后依然被访问，导致系统性能降低。
- **避免过时的数据造成不必要的计算**：通过设置合适的缓存过期时间，可以减少不必要的数据重计算，进而提升响应速度和性能。

3. 避免内存占用过多

- **释放过期的缓存数据**：如果缓存数据无限期存储在内存中，会导致系统内存占用过高。设置缓存过期时间，能够自动清理不再使用的数据，确保系统内存资源的有效利用。
- **减少垃圾数据的积累**：缓存过期帮助避免无用数据的积累，减轻系统负担，保持缓存的精简和高效。

4. 提高系统的一致性和可靠性

- **避免缓存击穿、缓存雪崩**：缓存过期机制可以配合一些策略，如“异步更新缓存”或者“分布式锁”，从而避免缓存失效导致对后端数据源的频繁访问，确保系统的稳定性和一致性。
- **避免因缓存被修改而引起的问题**：通过合理的缓存过期时间，可以避免因某些场景缓存被非法修改或失效，导致系统中不同节点的数据不一致，造成错误的数据输出。