**2-MongoDb**

**MongoDb 基础**

**存储文件**

**数据存储文件**

MongoDb 是面向文档的 NoSql 数据库，底层存储原理是基于 Bson 格式存储的文档。由于 Bson 对象限制，最大 16M，所以使用了 GridFS 机制，将大文件分割成多个小文档来存储

**Bson 和 Json 主要区别：**

* Bson 是二进制编码（UTF-8），Json 是文本编码
* Bson 遍历可以直接跳到某一位置开始读取，Json 需要扫描整个文档并做完格式匹配后再开始
* Bson 是二进制数据，并且有类型，在整形数据从 1 变化到 10 时候，数据大小不会发生变化。Json 数据存储是无类型的，会导致大小增加
* Bson 新增了二进制数组类型，在存储二进制数据不需要 Base64 转码后再转 Json，这样减少了计算开销以及存储空间

**系统存储文件**

* **数据文件：**MongoDb 将数据存储在一系列的数据文件中，当数据大小达到阈值就会分裂新的数据文件来存储，这样可以提高数据的读写性能
* **索引：**索引用来加速查询操作，是 B+ 树
* 索引在创建的时候必须指定顺序，所有集合默认有一个索引：\_id
* 索引类型：单字段索引，复合索引，多 key 索引，hash 索引，地理位置索引，文本索引（性能比较低）
* 索引常见问题：
* 复合索引遵循前缀匹配原则：{userid:1, score:-1} 包含了 {userid:1} 的索引
* 避免内存排序
* 索引交集：在查询两个字段的时候，优化器会自动做索引交集
* 后台创建索引：创建的时候不能立马执行下一个，会导致大量写操作和数据复制，从而占用大量 cpu，要确定第一个创建完再创建第二个
* **内存映射：**可以使应用程序直接访问内存数据，不用经过系统的文件系统操作，减少数据拷贝和转换操作，提高数据的访问速度
* **日志文件：**保证数据的持久性和一致性，在发生故障的时候可以进行数据恢复
* **副本集：**提供数据的高可用性和容错性

**文档**

**数据限制：**

文档是 MongoDb 的最小数据单位，最大大小是 16M，操作是原子性的

**id 生成：**

文档的 \_id 由 12 个字节组成，前 4 个是 unix 秒，3 个是机器标识，2 个是进程 id，3 个字节是计数器数字

**事务**

MongoDb 中支持事务，事务最大的运行时间是 60s

**数据库组成**

MongoDb 存在 3 个数据库， admin（存用户角色信息）， local（存在本地数据库的内容），config（保存分片相关信息）

**存储过程**

MongoDb 支持存储过程，通过 javascript 来写，保存在 db.system.js 中

**聚合处理**

* **聚合管道：**将一系列的数据处理划分成不同阶段，每个阶段之间通过管道来传输数据，主要用来处理一些简单的固定的聚集操作，处理结果不能超过 16 M，聚合过程中内存不能超过 100M
* **mapreduce：**通过 map 函数和 reduce 函数来处理，这些函数都由 js 来编写。主要用来处理复杂的，大量数据集的并行计算

**MongoDb 底层存储引擎**

底层引擎使用了 Wired Tiger

**底层数据结构**

使用了 B+ 树，中间节点只存索引，叶子节点存数据，数据基本单元是 page，page 有三个 list，WT\_ROW（从磁盘加载的数组），WT\_UPDATE（记录数据加载后到下个 checkponit 之间修改的数据），WT\_INSERT（记录数据加载后到下个 checkpoint 之间新增的数据）

**内存管理**

引擎内部的 cache（最大内存减一后除以二，如果分配不出来会导致数据库宕掉，是解压后的 b+ 树，可以直接使用），索引信息（默认是 500M），文件系统 cache（利用操作系统的文件缓存，减少和磁盘的交互，这里数据是没有解压的，不能直接使用）

**内存淘汰机制**

由内存使用量以及内存脏数据量来控制，如果淘汰运作太过频繁，会导致占用用户线程来进行淘汰操作，会影响查询（一般是因为大量的写入导致磁盘 I/O 撑不住）

**死锁检查**

引擎支持死锁检查，当事务等待资源超过一定阈值，引擎会尝试检测是否存在死锁情况，死锁后会选主一个或者多个事务回滚。（案例：高并发写入引起死锁，导致某些操作长时间占用写锁，连接可能长时间被占用，无法及时归还连接池，导致可用连接耗尽，可以使用批量写入或者异步写入来解决，同时还要检查连接池是否设置合理）

**MongoDb 锁机制**

**文档锁**

锁定单个文档，允许多个并发读写操作，写操作会互斥

**MVVC 机制**

主要用于事务，为每个事务创建数据的多个版本，读操作不受写操作的影响

**范围锁**

某些情况会锁定一个文档的范围，防止其他操作修改该范围内的数据

**乐观锁**

在读写操作的时候，依赖文档版本控制，文档都有一个内部版本号，写操作之前先读取版本号，写入时检查版本号是否改变，如果没有改变则成功，否则重试，这样减少了锁的使用。

**意向锁**

在锁父资源的时候，如果子资源有检索，则不能对父资源加锁。意向锁就是判断这个父资源有没有子资源被锁，如果没有意向锁则需要遍历所有子资源是否加锁

**锁矩阵**

对不同锁之间的权限进行整合

**锁的让渡和释放**

在某些情况下，读写操作可以释放它们持有的锁，防止长时间的阻塞。长时间运行的读取和写入操作，比如查询、更新和删除，在许多情况都会释放锁。MongoDb 操作也可以在影响多个文档的写入操作，在单个文档修改之间释放锁。对于支持文档级并发控制的存储引擎，比如 Wired Tiger，在访问存储时通常不需要释放

**不同数据库操作会使用的锁**

* select：库级别的意向读锁(r)，表级别的意向读锁(r)，文档级别的读锁(r)
* update/insert：库级别的意向写锁(w)，表级别的意向写锁(w)，文档级别的写锁(w)

**MongoDb 集群**

**日志文件**

* Journal：用于 MongoDb 的预写日志，然后 100ms 之后将该日志刷盘
* Oplog：用于主从复制，MongoDb 的写日志放在 Oplog，从节点通过拉回 Oplog 来实现数据同步
* CheckPoint：将内存数据进行磁盘持久化，每 60s 操作一次，并记录当前持久化点

**复制集群**

一个副本集最少有三个节点

* 最少一个节点（Primary）：复制集群的写操作，主节点挂了后会自动选新的主节点
* 一个或者多个从节点（Secondary）：从节点同步数据，主节点挂了后选举新的节点
* 零个或者多个仲裁节点（Arbiter）：只负责投票，不存数据

**分片集群**

**Chunk**

由一组文档组成的逻辑数据单元

**分片集群模块**

* Config：配置模块，存储分片集群元数据和配置，如分片地址，chunks等
* Mongos：路由服务，不存具体数据，从 Configs 获取集群配置转发到特定分片，并整合分片结果返回给客户端
* Mongod：具体的数据存储单元，每一个分片都是一个单独的复制集群

**数据均衡实现方式**

每个分片单元 MongoDb 都会有数据管理单元 chunk，大小默认 64M，范围 1-1024M，集群有多少个 chunk，每个 chunk 范围，chunk 是在哪个分配都存在 Config 中，当 chunk 内部包含数据超过阈值会分裂成两个，或者 chunk 内部超过 25W 条数据也会分裂，并且会定时检查 chunk 个数，如果分片的 chunk 个数不均匀，会启动 chunk 迁移，迁移比较消耗资源，一般要把它的执行时间设置到业务低峰期

**分片算法**

* 区间分配：按照 shardkey 做区间查询的分片算法
* hash 分片：按照 shardkey 的 hash 值来分片

**数据均衡 （rebalance）**

当不同分片 chunk 数目差异较大，就会将 chunk 从数据多的分片迁移到数据少的分片实现数据均衡。可以采取定时手动迁移，否则会等到出现差异较大才会执行

**实际部署考虑**

需要考虑多少 Mongos，需要多少发分配，分片键和分片算法用什么

**节点选举**

节点选举是基于 raft 协议修改

**选举触发条件**

* 复制集群初始化
* 主节点挂掉
* 主节点脱离副本集（网络原因）
* 参与选举的节点数必须大于副本集总结点数量的一半，否则所有节点只会保持可读状态

**选举流程**

* 副本集中的主节点必须满足副本中一半以上的成员投了赞同票，仲裁节点不保存数据，如果偶数个节点，则需要加一个仲裁节点
* 副本集成员节点被投了反对票，本轮不能成为主节点
* 集群中优先级为 0 的节点不能成为主节点，并且不能出发选举，但是具有投票权，拥有和主节点一致的数据集

**MongoDb 读写策略**

**写策略**

* w 参数：数据同步到多少个节点才算成功，参数包括：0-n/majorty，majorty 表示同步大于一半才返回
* j 参数：单个节点是否完成 oplog 持久化到磁盘才返回成功

**读策略**

* primary：读主节点
* primartPreferred：优先读主节点，不存在读从节点
* secondary：优先从节点，不存在读主节点
* secondaryPreferred：优先读从节点，不存在读主节点
* nearest：就近读，不区分主从，只考虑延迟

**读级别**

* local：直接从查询节点返回，不关注同步多少节点，读本地分配数据且属于当前分片的数据，重新选主的时候，读到的数据可能回滚
* available：适用于分片集群，和 local 差不多，读取本地所有可用数据，分片集群下也会读取到不属于当前分片的数据
* majority：返回被大多数确认过的数据，保证读取的数据不会回滚，但是不能保证读的数据是最新的
* lineraizable：适用于事务，等在他之前提交的事务完成才返回，能保证读到最新的数据，也保证数据不会回滚
* snashit：适用于事务，快照隔离，与 lineraizable 一致，但是可能读不到最新的数据，直接从快照里面取

**MongoDb 数据压缩**

* snappy：默认算法，压缩比 3-5 倍
* zlib：压缩比 5-7 倍
* 前缀压缩：索引使用
* zstd：有更好的压缩率

**MongoDb 常见性能优化**

**MongoDb 写性能（4核8G）**

* 瓶颈在单个分片上
* 数据量小，读写性能好，数据增加后性能急剧下降，最后稳定在 3k qps
* 分片集群和逐条写性能差不多，复制集群比分片集群性能好

**MongoDb 读性能（4核8G，2亿以上数据）**

* shardkey 查询：2ms，3w qps
* 索引查询：需要把数据转发到所有分片，然后聚合所有结果返回。瓶颈在 Mongos 上。10 分片1400 qps，10ms
* 其它查询：会全表扫描

**预分片**

数据分布已经比较随机了，但是插入性能还是比较差，chunk 分裂和 chunk 迁移都比较消耗资源，可以提前将分片的 chunk 创建好，同时设置 chunk 的分裂阈值

**内存排序**

查询数据时候排序方式要和索引的相同，否则会导致内存排序，数据量超过阈值会导致数据库返回错误

**链式复制**

在开启的时候，各个从节点不是直接从主节点去拉去 oplog，而是排成一条队伍去复制。优点是可以避免影响主节点性能，缺点是如果写策略是需要同步大量节点，会导致些时间增长

**排查性能问题方法**

可以通过开启 MongoDb 的慢日志检查，记录请求超过 100ms 的请求记录，记录包含请求内容，加锁相关信息。同时可以使用 expain 工具查询语句执行情况