MongoDB 性能优化

作者：黄金华

2017/03/22

1 MongoDB 简介

1.1 引言

MongoDB 是一个面向文档的开源 NoSQL 数据库，主要是解决面对大数据时关系型数据库所面临的“可扩展性”、“性能”以及“高可用性”问题。

一个 MongoDB 数据库包含一个或多个集合，而每个集合包含一个或多个文档。其中每个文档（相当于 MySQL 中一行）为 JSON 数据格式，并且支持任意模式扩展。

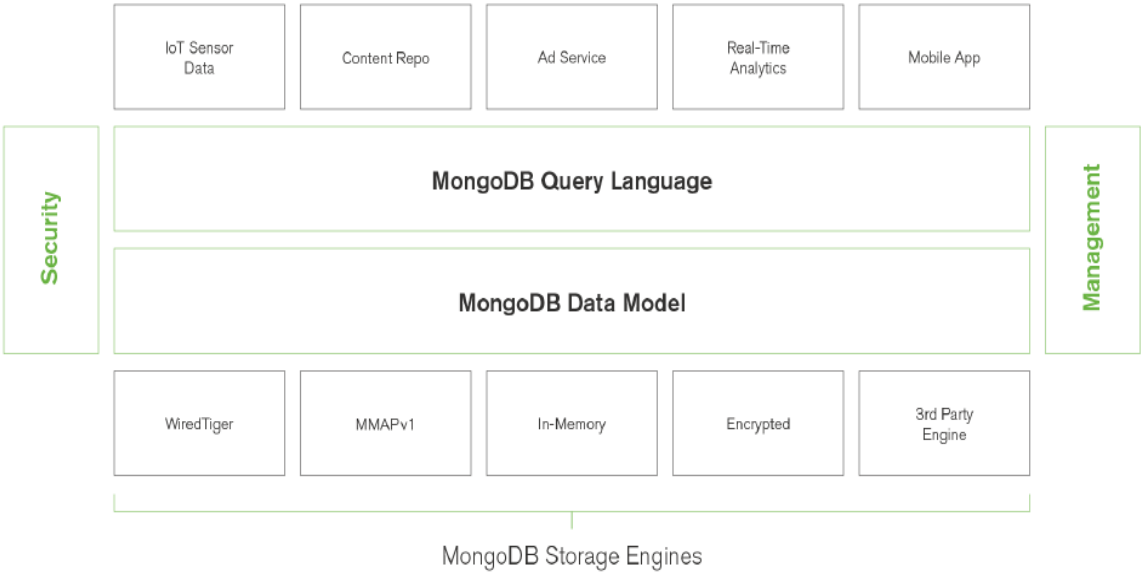
与大多数 NoSQL 数据库类似，基本上不支持 JOIN 功能和事务概念。不过支持文档级别的原子更新、二级索引，以及更为重要的是副本集和分片功能。

1.2 关系型与 NoSQL 数据库对比

	关系型数据库	非关系型数据库
类型	支持 SQL 标准	支持多种类型例如文档、键值对、多列数据库、图等
数据存储模型	数据以行列形式存放某个表中，同时一般符合某种范式。并且支持 JOIN 功能从多个表中获取数据	存储是灵活的，既可能是根据文档、键值对，也可能根据图中节点和邻边关系等存储数据
模式	表创建时其模式已确定，并且后续修改较困难	动态模式，也就是说也模式不是固定，可任意增加或删减
可扩展性	通常倾向购买更大更强的单个服务器	倾向水平扩展，也就说将数据分布多个普通服务器来扩展读写能力
事务	支持事务的 ACID 属性	一般支持 CAP 理论中分区和可用性，而牺牲一致性，即一般不支持事务。
一致性	强一致性	一般支持最终一致性
检索能力	因较成熟，一般可用 GUI 界面进行检索	一般需要编程能力，更加关注编程接口和功能

2 MongoDB 内部机制

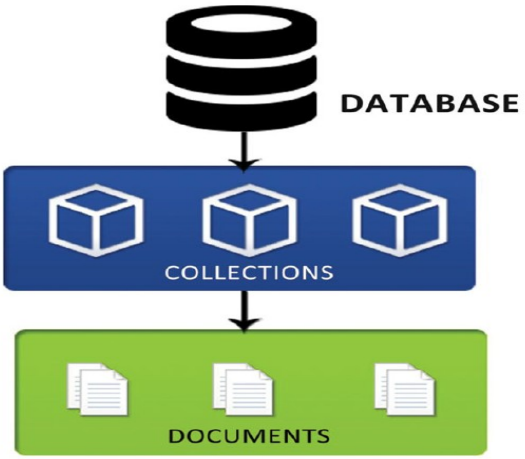
2.1 架构总览



存储引擎	主要特点	主要应用场景
WiredTiger	● 目前默认的存储引擎 ● 共享内存由 WiredTiger 引擎自身分配和控制，可最大化利用缓存； ● 高吞吐、高可靠、可扩展性： 1） 支持文档级别（此处文档对 MySQL 中一行）多版本并发控制，提升吞吐量； 2） 异步更新日志（默认 100ms 一次）减少 I/O 请求； 3） 支持自动故障恢复的副本集(与传统的主从模式不同，支持 Raft 协议自动选举主节点） 4） 支持良好的水平扩展（Shard）性，提升读写能力； ● 支持文档压缩从而减少 I/O 操作 ● 多 core 扩展性：利用 lock-free 算法等减少锁冲突，支持多 core 扩展 ● Write/Read-Concern: 允许单独设置（可支持 session 或 query 级别）读写关注级别（例如数据必须要写到多少几个节点才算完成，或者才能被读取）；	● 适合大多数应用场景，例如 IoT、客户数据管理、产品类目管理等； ● 不支持 JOIN 和事务操作 ● 日志一般 100ms 才更新一次，存在数据丢失的风险（不过一般可采用副本集来实现高可靠性）

Encrypted	● 基于 WiredTiger 同时支持对文档加密	● 特别适合对信息安全高度敏感的行业，例如金融、零售、政府部门等；
In-Memory	● 所有数据均在内存操作，不涉及磁盘 I/O ● 文档级别并发控制	● 特别适合对时延要求高的应用场景，例如交易系统、实时监控、信用卡验证、交易订单处理等； ● 可同时在副本集中采用 WiredTiger 存储引擎来备份数据
MMAPV1	● 早期的默认存储引擎操作 ● 所有文档文件通过 MMAP 映射到内存 ● 采用 WAL(Write-ahead-log)保证一致性 ● 集合级别（集合对应 MySQL 中的表，包含多个文档）并发控制	● 一般只是在老版本中才建议使用 ● 文档缓存由 OS 控制，从而 MongDB 无法真正控制缓存替换，因此读写性能一般

2.2 数据模型



如上图所示，一个 MongoDB 数据库包含一个或多个集合（Collection），同时每个集合包含一个或多个文档（Document）。其中每个文档为 JSON 格式，例如：

```
{
"_id" : 1,
"name" : { "first" : "John", "last" : "Doe" },
"publications" : [
            {
              "title" : "First Book",
              "year" : 1989,
              "publisher" : "publisher1"
            },
            { "title" : "Second Book",
              "year" : 1999,
              "publisher" : "publisher2"
            }
]
}
```

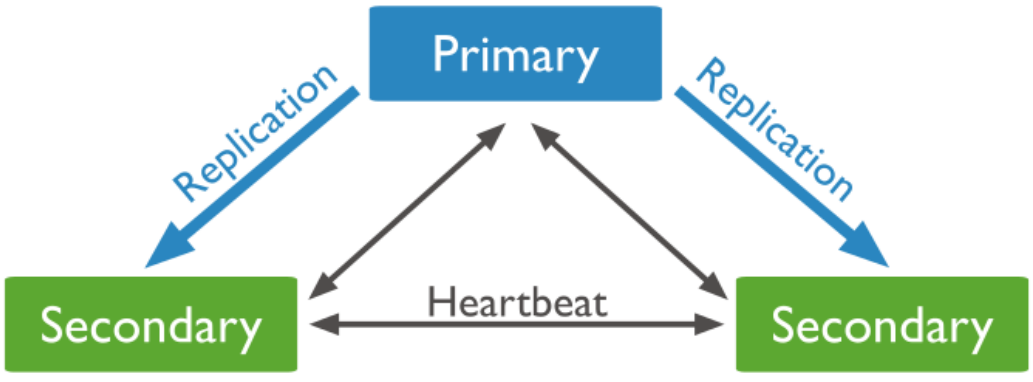
2.3 索引

总体而言，MongoDB 提供下列索引：

- 复合索引：在多个属性字段上建立索引；
- 唯一索引：通过将索引指定为唯一，可以确保文档的唯一性；
- 数组索引：对于包含数组的属性字段，将为每个数组值都存储为单独的索引项；另外，创建数组索引不需要特殊的语法，即与普通索引建立方法相同；
- TTL(Time to Live)索引：允许指定数据库自动删除数据前等待的时间段；
- 地理空间索引：提供地理空间的索引，以优化与二维空间内的位置相关的查询；
- 稀疏索引：稀疏索引仅包含含有指定字段的文档条目；
- 部分索引：稀疏索引更灵活的版本，可以指定某种约束条件判断某个文档是否需要建立索引；
- 哈希索引：根据哈希值建立索引；主要用于基于哈希的分片（Sharding）中；
- 文本搜索索引：提供词干提取、切词分析、停用词使用等功能的文本检索；

2.4 复制

2.4.1 复制基本形式



一般通过两种方式让 Slave 节点保持与 Master 节点同步:

- 初始化同步：刚启动的时候，Slave 节点通过“init sync”复制 Master 节点（或其他可用节点）复制所有数据库数据；
- 增量数据同步：Master 节点维护一个固定集合（也就是 oplog）用来存放历史数据的写操作，而所有 slave 节点通过读取并重放 Master 节点中 oplog 集合数据保持与 Master 节点一致；

值得一提的是，一般 Slave 节点与 Master 节点存在延迟，因此有可能在 Master 节点已经写入的数据并不会立即在 Slave 节点上能检索到。

2.4.2 主/从模式

比较典型的复制模式，通常由一个主节点和一个从节点组成。但是不支持自动故障恢复，而且一般还需要人工干预设置主从节点。因此该模式在 MongoDB 中逐渐不推荐使用，取而代之的是使用副本集（Replicate Set）。

2.4.3 副本集

与主/从模式相比，副本集中一个重要功能是支持 Master 故障时自动选举新的 Master 节点从而实现故障自动恢复。

一个副本集包含以下节点：

- **主节点：**
  - 一个副本集中有且仅有一个主节点，且所有写请求由主节点来处理，而从节点可以根据应用需要让主、从节点处理；
- **从节点：**
  - 每个普通节点都设定一个优先级（用于选举新的主节点），同时可以参与选择投票、有机会成为主节点和可以处理读请求；
  - 除了普通的从节点外，还可能存在多个特殊的从节点：
    - 优先级为 0 的从节点：不能选举为主节点；
    - 隐藏节点（Hidden）：既不能选举为主节点，也不能处理读请求；
    - 延迟节点（Delayed）：故意让从节点与主节点的数据同步延迟一段时间（特别系统升级时因各种意外导致数据库损坏时恢复原来数据）（因数据与主节点不是实时同步，因此既不能选举为主节点，也不能处理读请求）
    - 仲裁节点：该节点仅参与投票，并不保存数据，因此也无需与主节点同步
    - 非投票节点：具有普通节点其他功能，除了不参与主节点选举投票外（但也有可能选举为主节点）；

在组建副本集时，通常建议：

- 副本集中节点个数为奇数（当网络分区发生时，确保有且仅有一个节点获得超半数票数选举为主节点），同时尽量所有节点都能选举为主节点；
- 可考虑需要时设定隐藏节点（仅用于备份，并可与其他业务节点隔离）和延迟节点（可用于意外导致的数据库损坏恢复）；

2.4.4 读写关注级别

在读写请求时，允许客户接口为每个读写请求设置关注级别（Read/Write Concern）。

- **写关注级别（Write Concern）**
  - Write Acknowledge：默认关注级别；仅当写操作完成（但不包括写日志）时才向客户端确认写请求完成；
  - Journal Acknowledged：仅当主节点将日志写到磁盘后才向客户端确认写请求完成；
  - Replica Acknowledged：可以指定多少个从节点将日志写回到磁盘后才向客户端确认写请求完成；
  - Majority：仅当大多数从节点将日志写回到磁盘后才向客户端确认写请求完成；
  - Data Center Awareness：可以设置特殊策略，仅当某些数据中心的节点将日志写回到磁盘后才向客户端确认写请求完成；

- **读关注级别（Read Concern）**
  - local：默认级别；该读取返回当前节点的最新数据（不保证该节点最新数据与其他节点已经同步）
  - majority：读取该节点已经确认被多数节点确认写成功的最新数据；
  - linerizable：确保当前数据被读取时该节点依旧是主节点，而且即使其他节点重新选举为主节点时该数据依旧不会被回滚；该操作对时延有较大影响，因此需要设置 maxTimeMS 防止长时间等待

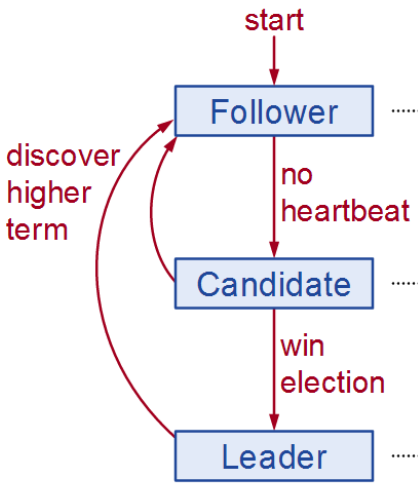
- **读偏好（Read Preferences）**
  - primary：默认级别；仅从主节点读取数据；
  - primaryPreferred：优先从主节点读取；当主节点不可用时，从从节点读取；
  - secondary：从从节点读取；
  - secondaryPreferred：优先从从节点读取；如果所有从节点不可用，则从主节点读取；
  - nearest：从最近的节点（根据时延判别）读取；

2.4.5 选举算法

选择算法基于 Raft，并对此进行改进。

- Raft 一致性算法：

Server States and RPCs

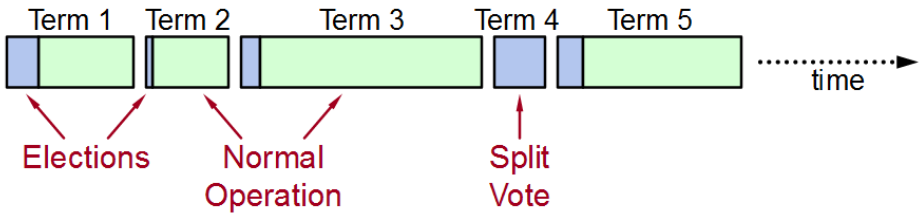


Passive (but expects regular heartbeats)

Issues **RequestVote** RPCs to get elected as leader

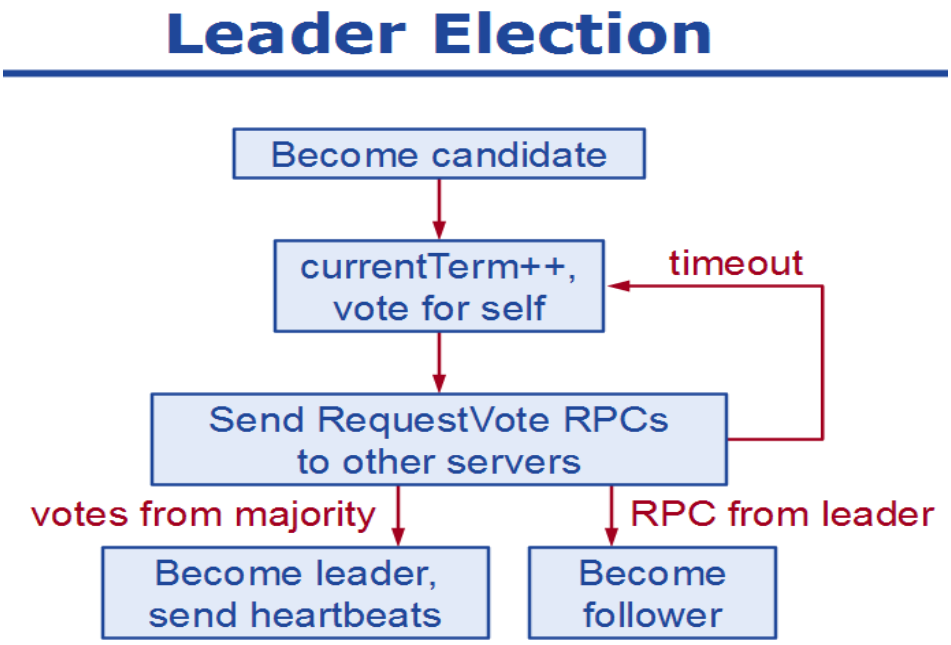
Issues **AppendEntries** RPCs:
- Replicate its log
- Heartbeats to maintain leadership

Terms



- **At most 1 leader per term**
- **Some terms have no leader (failed election)**
- **Each server maintains current term value (no global view)**
  - Exchanged in every RPC
  - Peer has later term? Update term, revert to follower
  - Incoming RPC has obsolete term? Reply with error

Terms identify obsolete information

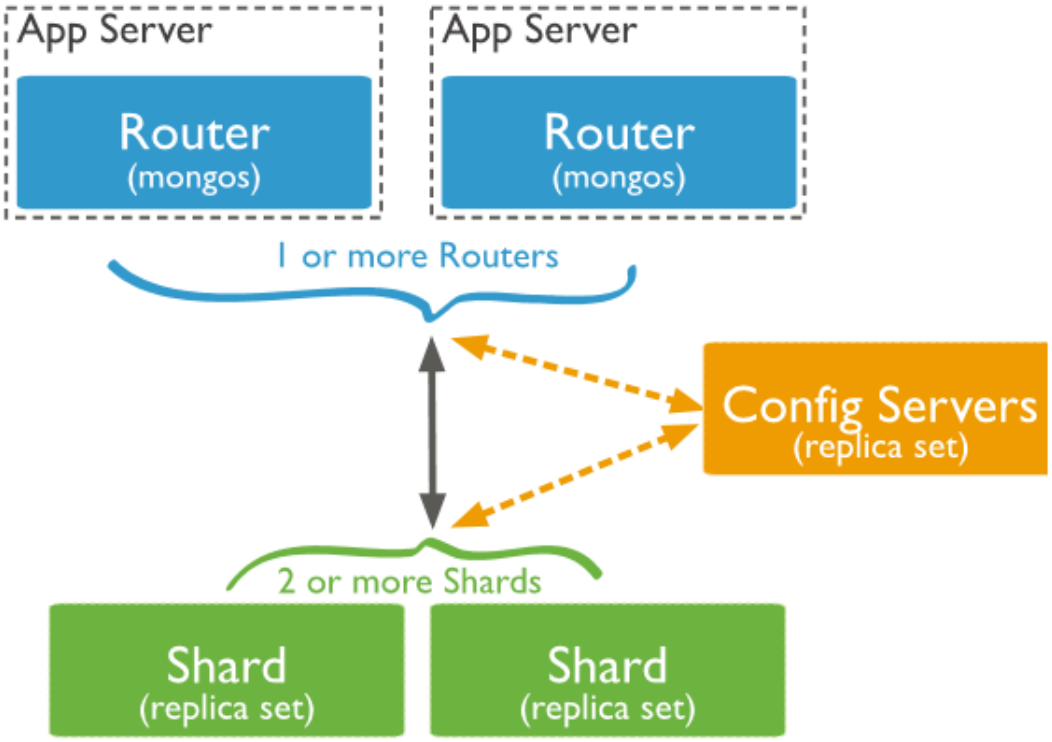


- MongoDB 选举算法
  - 基于 Raft，其重点在于选举的时候应该投票给谁，因为谁赢得了大多数选票谁就成了新的主节点（Leader）,而其他从节点（Followers）只是永远与主节点保持同步。其主要规则如下：
    1) 何时发起投票：
       - 副本集刚创建，或者无法收到（10s 内）主节点心跳信息；
    2) 如何投票：
       - 同一个任期内（Term，或者说一段时间内，用一个递增的数字表示，而非绝对时间），有投票权的节点有且仅能投票一次；
       - 仅当收到请求投票的信息的 Term 大于或等于当前 Term 时，且其操作日志与自己一样新或者比自身还新时，才给该节点投票（且更新当前自己的 Term）；
    3) 谁成为新的主节点：
       - 同一个任期内，赢得超半数节点的票数则为主节点；
  - 扩展
    1) 从节点链（Chaining）：
       - 减少主节点负载；
       - 支持不投票的从节点；
    2) 节点优先级
       - 选举协议版本 0：直接使用优先级参与选举，也就是优先级高的更有机会赢得选举；
       - 选举协议版本 1（目前默认版本）：
         - 只考虑自己与当前主节点的优先级，其他节点的优先级不决定选举与否；
         - 可以为新选举的节点设置“catchUpTimeoutMillis(默认 2000)”，尽量让新选举的主节点能与旧主节点的日志保持一致（虽然旧的主节点不可用，但其他节点可能包含最新的日志，同时从节点会

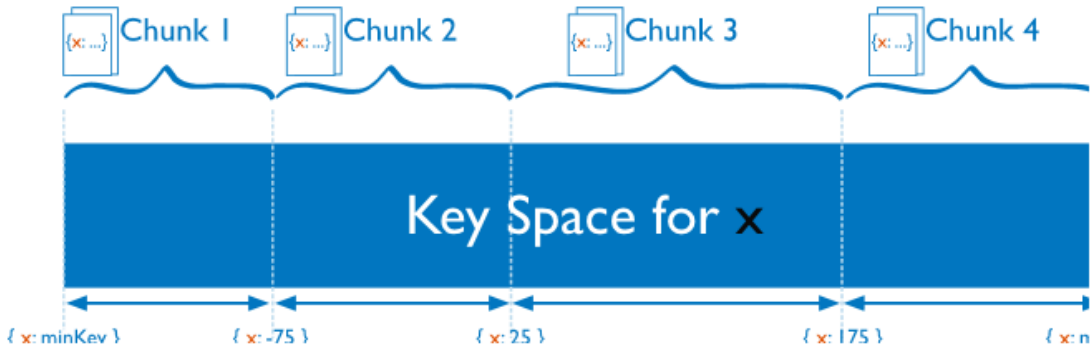
根据 ping 时延以及其他节点复制状态自动选择新的节点同步日志），从而尽量避免旧的主节点写回滚；

2.5 分片（Sharding）

2.5.1 基本架构



- **Shard**:
  1) 根据 Shard key 将数据分成若干 Chunks（默认大小 64M，可配置）存储到 Shard 中；



- **config servers**:
  1) 存储 Sharded 集群的元数据和配置信息（本身也是 mongod 进程，只是启动的时候配置成 configsvr）
  2) 其 balancer 线程会根据每个 Shard 节点 Chunks 分布情况自动在各个 Shard 之间迁

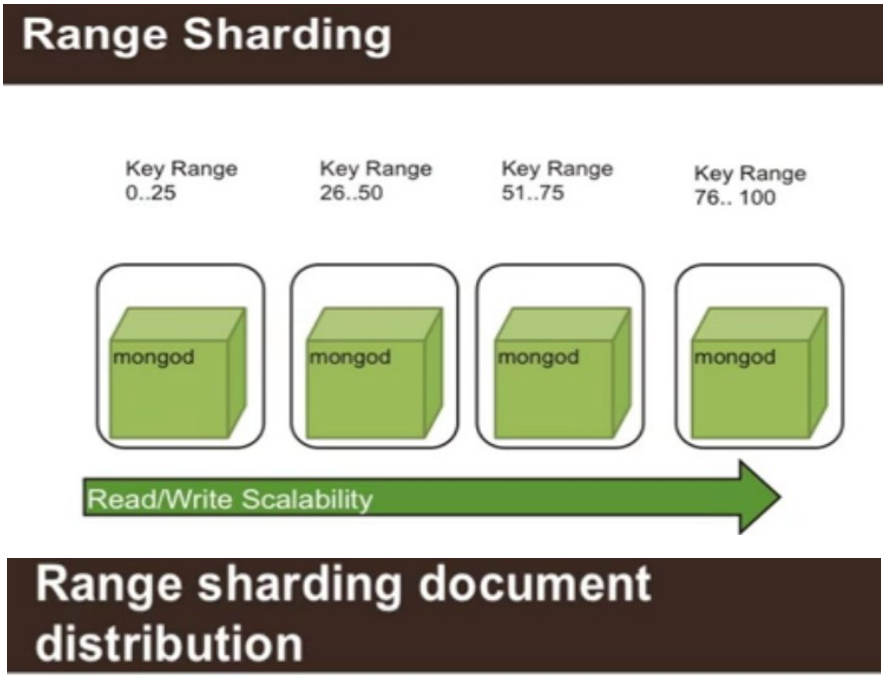
移 Chunk，以便实现负载均衡；

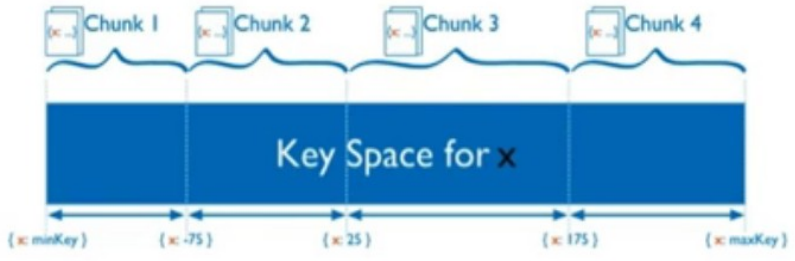
- **mongos**:
  1) 位于客户端与 Sharded 集群之间，承担路由功能（与 mongod 单独分离的一个进程）；
  2) 所有信息常驻内存，同时一般仅当初始化阶段或者发生配置信息更改时（例如数据在 Shard 之间自动迁移）才与 config servers 同步配置信息

2.5.2 Shard 类型

一般来说，分片有以下几种类型：

- Range Sharding：



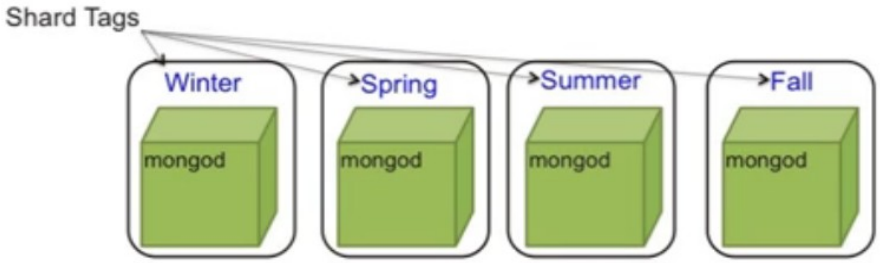


- Tag-Aware（Zones） Sharding:

Tag-Aware Sharding

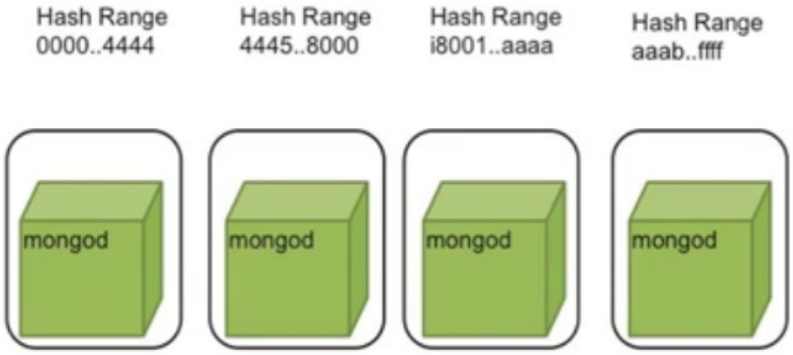
Tag Ranges

Shard Tag	Start	End
Winter	23 Dec	21 Mar
Spring	22 Mar	21 Jun
Summer	21 Jun	23 Sep
Fall	24 Sep	22 Dec

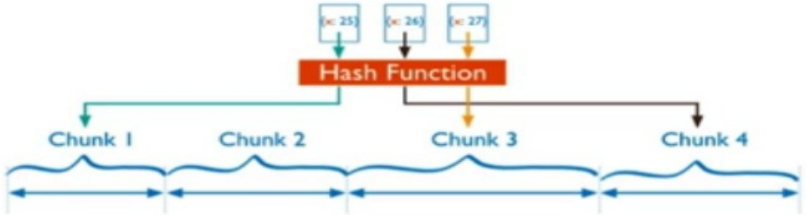


- Hash Sharding:

Hash-Sharding



Hashed sharding document distribution



2.5.3 分片建议

- 选择一个好的 Shard key:
  1) Cardinality（基数）：尽量选择 cardinality 大作为 shard key，这样可以避免大量数据存储到一个 chunk 所带来 chunk 切分开销以及多个 shard 服务器之间 chunk 迁移开销；
  2) 写扩展性：要尽量让写操作均衡分布到 Shard 服务器上从而提升写性能；
- 事先规划好足够容量；
- 一般运行三个或更多 config servers 以便实现高可靠性；
- 每个 Shard 本身结合使用副本集来高可靠性；
- 启动多个 mongos 实例；
- 批量插入：批量插入时事先已切分好数据从而避免数据插入时导致 chunk 切分以及 chunk 迁移；

3 性能优化

3.1 基本方法论

对于性能优化的基本方法论，以及基本工具可参阅：https://zhuanlan.zhihu.com/p/25013051

3.2 基准测试

在性能优化之前需要衡量数据库系统性能，而对数据库而言，其性能的两个主要指标是：时延和吞吐量。为此，常需要借助基准测试来衡量系统性能，并且将时延或吞吐量作为优化目标。

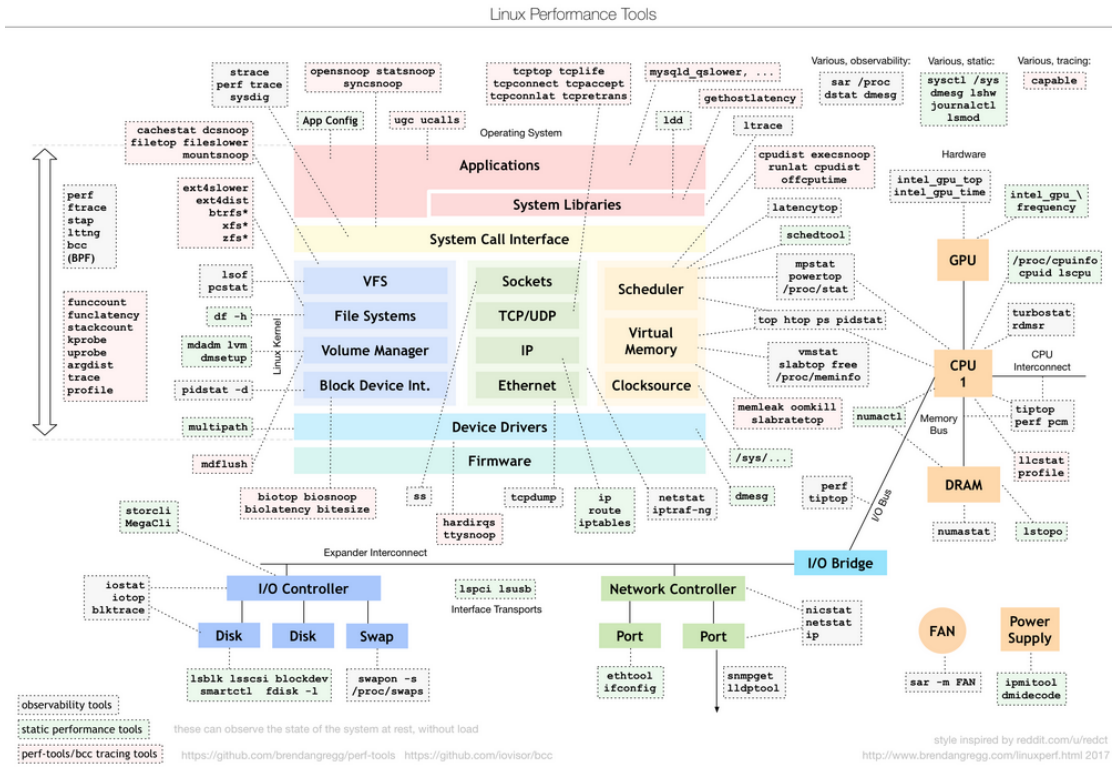
关于基准测试用例配置、部署等，可参考自动化部署脚本。

3.3 性能监控

3.3.1 基础性能监控

对 CPU/Memory/Disk/Networking/Kernel 等监控，可采用 perf/ftrace/bcc/top/dstat 等多种工具，既可以支持静动跟踪，也支持动态跟踪。

具体工具可参考 http://github.com/sjtuhjh/perftools，在此不再赘述。



3.3.2 应用内部监控

总体性能分析：

- mongostat/mongotop(https://github.com/mongodb/mongo-tools)：实时监控MongoDB 服务器、副本集以及分片集群（或者直接查看 db.serverStatus()）；
- 内部 profiler 工具：
  1) 通过设置 mongo shell 客户端设置 db.setProfilingLevel（）以及通过 db.system.profile.find()查看查询时延等；
  2) 同时可通过 pt-mongodb-query-digest <IP>:<PORT>/<Database>(例如：pt-mongodb-query-digest 127.0.0.1:27017/ycsb) 查看总体时延分布情况；

单个 Query 性能分析：

- 内部 Explain 工具（支持三种形式）：
  1) queryPlanner（默认）：
     例如 db.testdb.explain(). find()或 db.testdb.explain(“queryPlanner”). find()，仅返回执行计划，并不会实际执行
  2) executionStats：
     例如 db.testdb.explain(“executionStats”).find()，不仅返回执行计划，也会返回执行情况；
  3) allPlansExectuion:
     例如 db.testdb.explain(“allPlansExectuion”).find()，不仅返回所有执行计划，也会返回所有执行计划执行情况；

更多分析工具，可参考 https://github.com/sjtuhjh/perftools/tree/master/apptools/mongodb/

3.4 基础优化

3.4.1 BIOS/CPU/Memory

在对应用优化之前，要确保 BIOS/CPU/Memory 配置恰当，否则后续优化往往事倍功半。为此，需要借助一些测试工具来验证硬件功能是否工作正常（或者最优）。值得注意的是，开始测试的时候常常无法根据数据来确定是否工作正常，因此可将数据与历史优化后的数据或者对应的 X86 测试数据进行对比。常见的测试工具如下：

- **CPU**:
  SpecInt2006(https://github.com/sjtuhjh/applications/tree/master/apps/cpu/spec_cpu2006)

- **Memory**: lmbench/Stream
- **DISK : I/O** : fio 测试

Location	Disks	RAID Level	Purpose
/ (root)	2	1	Operating system
$PGDATA	6+	10	Database
$PGDATA/pg_xlog	2	1	WAL
Tablespace	1+	None	Temporary files

  - RAID1：将两块硬盘构成 RAID 磁盘阵列，其容量仅相当于一块磁盘，但另一块磁盘总是保持完整的数据备份。一般支持“热交换”。
  - RAID10：首先建立两个独立的 RAID1，然后将两个独立的 RAID1 再组成一个 RAID0

  建议配备 RAND10，同时支持 BBU(Battery Backup Unit)和 WRITE-BACK 功能的存储设备。这样既可以有良好的读写性能，又能防止意外情况下（例如突然断电）存储内容丢失。

- **BIOS 电源管理**：将电源管理设置成“性能”模式以便最大发挥 CPU 性能；

常见的基准测试工具也可参考 https://github.com/sjtuhjh/applications/tree/master/apps。

3.4.2 OS/Kernel

3.4.2.1文件系统

通常建议采用 EXT4 或 XFS 文件系统。

- Barriers I/O (Barrier 之前的所有 I/O 请求必须在 Barrier 之前结束，并持久化到非易失性介质中。而 Barrier 之后的 I/O 必须在 Barrier 之后执行。如果本身存储设备自带电池足以预防在突然掉电时其缓存内容依然可以被正确持久化，则可以关闭

Barrier 一般提高性能)。一般文件系统挂载时提供选项可以关闭 barriers,例如：mount –o nobarrier /dev/sda /u01/data

- 打开 noatime，减少不必要的 I/O 操作，例如：mount –o nobarrier,noatime …
- 调整 vm.overcommit (将其设置为 2，尽量避免 OOM(Out of Memory killer))
- 写缓存优化（需要根据需要调整参数，当脏页刷写太频繁，会导致过多 I/O 请求，而刷写频率过低，又可能导致所占内存过多，而在不得不刷写脏页时导致应用性能波动）：
  - vm.dirty_background_ratio: 当脏页数量达到一定比例时，触发内核线程异步将脏页写入磁盘；
  - vm.dirty_ratio: 当脏页数量达到一定比例时，系统不得不将脏页写入缓存。此过程可能导致应用的 I/O 被阻塞。
- I/O 调度
  - 对于普通磁盘，建议使用 deadline 调度方式，而固态硬盘各种调度方式影响并不大；
- Read-ahead 调整：
  - blockdev –getra /dev/sda 查看 sda
  - blockdev –setra <sectors> /dev/sda （设置预读缓冲大小）
- vm.swappiness=0 （尽量避免使用 SWAP）

3.4.2.2NUMA

一般建议关闭 NUMA（或者使用交织调度并关闭 NUMA BALANCING）性能会更好。主要原因是，MongoDB 后台进程使用共享机制共同访问一大块内存，而该块共享内存一般无法在一个 NUMA 内存节点分配。因此必须在多个内存节点分配共享内存。同时，MongoDB 后台进程可能是随机访问共享内存内某块数据，而在随机访问模式下，关闭 NUMA（包括 NUMA BALANCING）性能会更好。

3.4.2.3调度策略

适当增大 kernel.sched_migration_cost_ns 和 关闭 kernel.sched_autogroup_enabled （=0）。主要原因是 autogroup 会根据终端的响应时间要求自动对进程分组并优先处理响应要求更高的分组，而这种机制对于与伪终端相关的 MongoDB 后台进程而言，有可能会影响其调度从而降低性能。

3.4.2.4HugePage

鉴于 MongoDB 文档大小过小，同时其 I/O 访问时随机的，因此关闭 HugePage 性能会更好。

3.4.2.5资源限制(ulimit)

一般修改/etc/security/limits.conf，包括：

- -f (file size): unlimited
- -t (cpu time): unlimited
- -v (virtual memory): unlimited
- -n (open files) : 大于 20,000
- -m (memory size): unlimited
- -u (processes/threads): 大于 20,000

3.4.2.6存储设备设置优化

3.4.2.6.1 分区对齐

分区对齐的目的是让逻辑块与物理块对齐从而减少 I/O 操作。通常可以借助 parted 命令来自动对齐，例如：

device=/dev/sdb

parted –s –a optimal $device mklable gpt

parted –s –a optimal $device mkpart primary ext4 0% 100%

其中 optimal 表示要尽量分区对齐从而达到最好性能。

3.4.2.6.2 SSD 优化

SSD 支持 TRIM 功能，打开该功能可以防止未来读写性能恶化。通过 Btrfs, EXT4, JFS 和 XFS 文件系统都支持该功能。可通过下列命令进行配置：

- 普通磁盘: 在 mount 的时候添加 discard 选项，例如：
  /dev/sda / ext4 rw,discard 0 1
- LVM (Logical Volumes)： 在/etc/lvm/lvm.conf 中添加 issue_discards=1；

3.4.2.6.3 I/O 调度算法

一般建议将/sys/block/<设备名>/queue/scheduler 设置为 deadline 或 noop。尤其是对普通机械硬盘，不要将其设置为 cfq 算法。

3.4.2.6.4 SWAP

尽量禁止使用 SWAP，从而避免性能波动。可在/etc/sysctl.conf 中添加 vm.swappiness=0 并通过 sysctl –p 使其生效。

3.4.2.6.5 Readahead

使用 blockdev –setra <value> 设置为较少的值(例如 32 或 64),这是因为文档本身大小较小,同时 I/O 访问时随机的,所以其值过大会导致不必要的 I/O 开销(以及内存开销)。对于 SSD,一般可先设置 16，并且根据测试不断调整为其他值（例如 32 或 64 等)。

3.5 MongoDB 配置调优

相对于关系型数据库配置型而言，MongoDB 本身可配置项较少。其中 mongod.conf 关键配置项有以下几个：

- storage.journal : 一般需要日志功能，因此建议打开；
- wirtedTiger. engineConfig .cacheSizeGB：一般建议为可用内存的 60%左右（防止设置过大，从而导致 OS 文件系统缓存没有足够内存导致系统性能变差)；
- wiredTiger.engineConfig.journalCompressor: 一般建议使用压缩功能从而减少 IO(不过会增加 CPU 开销)
- 跨网络节点时一般将 networkMessageCompressors 设置为 snappy；
- 使能 wiredTiger.engineConfig.directoryForIndexes，同时条件允许可以为不同数据库选择不同存储设备；
-

同时，还有其他配置还有：

- 同时使用 numactl --interleave=all 方式启动 mongod；
- 尽量采用 XFS 文件系统而非 ext 文件系统(相比 XFS，目前已知 WiredTiger 采用 ext4 会有性能损耗）

3.6 模式以及索引优化

- 尽量使用一个文档存储该记录的所有信息；
- 避免大文档（最大文档不允许超过 16M，否则需要使用多个文档来存储)；
- 避免文档无限增长（MMAPv1 引擎）
- 避免大的数组索引（不过在多个属性上建立组合索引更高效）
- 避免较长的属性名；
- 删除不需要的索引；
- 避免在基数较小的属性上建立索引（因为基数较小，即使有索引其效果依然类似全文档检索）
- 删除那些是其他索引前缀的索引（因为重复了）
- 必要是建立组合索引
- 必要时仅在部分属性建立索引
- 避免带有通配符的正则表达式（因为这有可能导致全文档检索）
- 尽量采用索引优化（WiredTiger)，例如：压缩，采用其他单独磁盘等；

3.7 应用模式

- 仅仅更新所需要的文档（以及相关属性）（而不是检索所有文档，更新相关属性后再写会）
- 仅可能有覆盖索引；
- 在检索中避免求反（因为这有可能导致扫描所有文档）
- 可事先用 explain()检验每个 query 的性能；
- 选择合适写关注级别（write concern）
- 如果不能容忍最终一致性，则仅从主节点读取数据；
- 选择合适的读关注级别（read concern）
- 尽量选择能数据均匀分布的 shard keys

3.8 算法代码优化

与具体 CPU 相关的优化，待补充。

4 附录

[1] www.mongodb.com