使用MongoDB进行开发分享一二

BBD 钟秋

为什么要使用MongoDB?

- schema-less（动态模式、自由模式）
  - 关系型数据库的时候，需要预先定义，需求变动，线上环境增减字段没有好的方案（保留扩展字段，格式化的字段，列转行..）
  - 使用MongoDB后，不需要预先定义，由程序端决定，同一个集合不要求统一模式，线上环境随时可增减字段
- 高可用
  - 复制，副本同步
  - 选举，自动故障转移
- 可扩展
  - 分布式，解决单机瓶颈，支撑海量数据
  - 路由分发，数据自动迁移均衡
- 高性能
  - 在读写性能上满足我们大部分场景需求
- 丰富功能
  - 二级索引，聚合操作
  - 全文检索，定容集合，TTL索引，GridFS等
- All-in-one
  - 对于大多数需求，我们需要的功能和特性MongoDB都已经提供，不需要部署多种DB，架构，开发和运维都变得简单

如何用好MongoDB?

- 做好设计（开发）
  - 根据读写场景，做好数据结构设计
    - 要一起返回的是否内嵌冗余在一个文档?
    - 频繁更新的数据和静态的数据是否要拆分集合?
    - 是否使用数组?
    - 能否在插入数据时，通过计算并冗余字段，解决一些简单的聚合统计需求?
    - ...
  - 数据量大的集合，为查询和排序设计好索引
  - 写入数据是否重要，读写一致性要求如何，我能在数据没有持久化或同步之前返回吗?
  - 业务对数据一致性要求，是否能够读写分离?
  - 预估数据规模和负载，考虑集合是否分片。分片集合常用查询是哪些，片键如何设计才能避免写热点同时兼顾范围查询?
  - ...
- 做好监控（运维）
- 调优（DBA）

反范式设计：冗余，冗余

- 使用MongoDB时，我们最重要的设计技巧就是充分利用冗余：
  - 内嵌文档内嵌数组
    - 一次检索，减少io开销
    - 单文档更新原子性，避免分开更新事务问题
  - 插入时提前计算冗余字段：
    - 比如数组的大小（如果我们后面需要过滤数组大小在某个范围的文档时，就可以在这个字段建索引快速查询，而不是用$where + js去全表筛选）
    - 冗余倒排字段，变 *abc查询变为cba*查询，让正则匹配能利用上索引
    - ...

是否使用数组?

- 应该使用的场景
  - 数组元素的上限不大（记住bson文档有16M的大小限制）
  - 数组需要和文档其他字段一起返回的需求场景
  - 不需要或者说极少需要随机更新或删除数组中元素
  - 不需要对数组排重
- 案例：（评论系统，引用评论列表）
  - 需要一起展示评论和引用评论的内容
  - 评论的回复通常数量很有限
  - 评论了基本不会修改

```
{
    "_id": ObjectId("..."), //评论id, 主键
    "resource_id": //评论所属的文章或其他主题资源id
    "content": "...", //评论内容
    "refs": [ //引用评论
                {
                    "_id": Object("..."),
                    "content": "...",
                    ...
                },
                {
                    "_id": Object("..."),
                    "content": "...",
                    ...
                },
                ...
    ],
    ...
}
```

是否使用数组?

- 不应该使用的场景
  - 数组元素的上限很大且不可控
  - 数组和文档其他字段没有一起返回的需求
  - 会频繁随机更新或删除数组中元素
  - 需要对数组元素排重
- 案例：（用户交互系统：关注和粉丝列表）
  - 频繁的关注取消关注操作
  - 偶像的粉丝很多，数量不可控
  - 需要避免重复
  - 数组中冗余的粉丝信息会经常修改

```
{
    "_id": ..., //用户id
    "username": //用户名
    "fans": [ //粉丝列表
                {
                    "_id": ...,
                    "username": "...",
                    "icon": "...",
                    ...
                },
                {
                    "_id": Object("..."),
                    "username": "...",
                    "icon": "...",
                    ...
                },
                ...
    ]
}
```

关于分页查询

- 一般的分页方式：
  - db.coll.find({...})
    .sort({...})
    .skip(offset)
    .limit(pageSize)
  - 但只适合小数据量，翻页不深的情况（offset不能太大），否则需要遍历很多文档性能会很差
- 前后页方式：
  - db.coll.find({"_id":{"$gt":lastPageMaxId}})
    .sort({"_id":1})
    .limit(pageSize)
  - 适合大数据量，需要翻页很深的情况，采取前一页后一页的翻页方式
  - 有很多限制：过滤字段和排序字段必须是同一字段，且只能是单字段，并且是唯一值字段。字段上要有索引用于过滤和排序

关于_id

- 默认的ObjectId
  - 推荐，并且能满足我们绝大部分需求
  - 注意保存在集合中的_id并不能保证严格自增(客户端生成，分片集群环境)

- 自定义整数_id
  - 可以利用findAndModify操作，

  这个操作可以返回修改后的值，

  同时因为单文档操作原子性，

  保证了并发安全

```
robotComment@undefined> db.ids.find()
{ "_id" : "robotComment", "cnt" : 43336000 }
{ "_id" : "robotCrawlerTask", "cnt" : 6 }
{ "_id" : "robotStat", "cnt" : 112000 }
{ "_id" : "robotUrlMappingInfo", "cnt" : 8524000 }
```

```
@Override
public long getId() {
    return TypeUtil.getLong(collIds.findAndModify(query, fields, null, false,
                            new BasicDBObject("$inc", new BasicDBObject("cnt", 1l)), true, false)
                            .get("cnt"));
}
```

关于_id

```
//生产者
this.job = new Runnable() {
    public void run() {
        while(!stop){
            long n = TypeUtil.getLong(collIds.findAndModify(query, fields, null, false,
                    new BasicDBObject("$inc", new BasicDBObject("cnt", queueSize)), true, false)
                    .get("cnt"));
            long low = n - queueSize + 1;
            for(long i=low; i<=n; i++){
                try{
                    idQueue.put(i);//blocking queue
                }catch(InterruptedException e){
                    Thread.currentThread().interrupt();
                }
            }
        }
    }
};
```

```
//消费者线程调用
@Override
public long getId(){
    try{
        return idQueue.take();
    }catch(InterruptedException e){
        Thread.currentThread().interrupt();
    }
    return -1;
}
```

索引

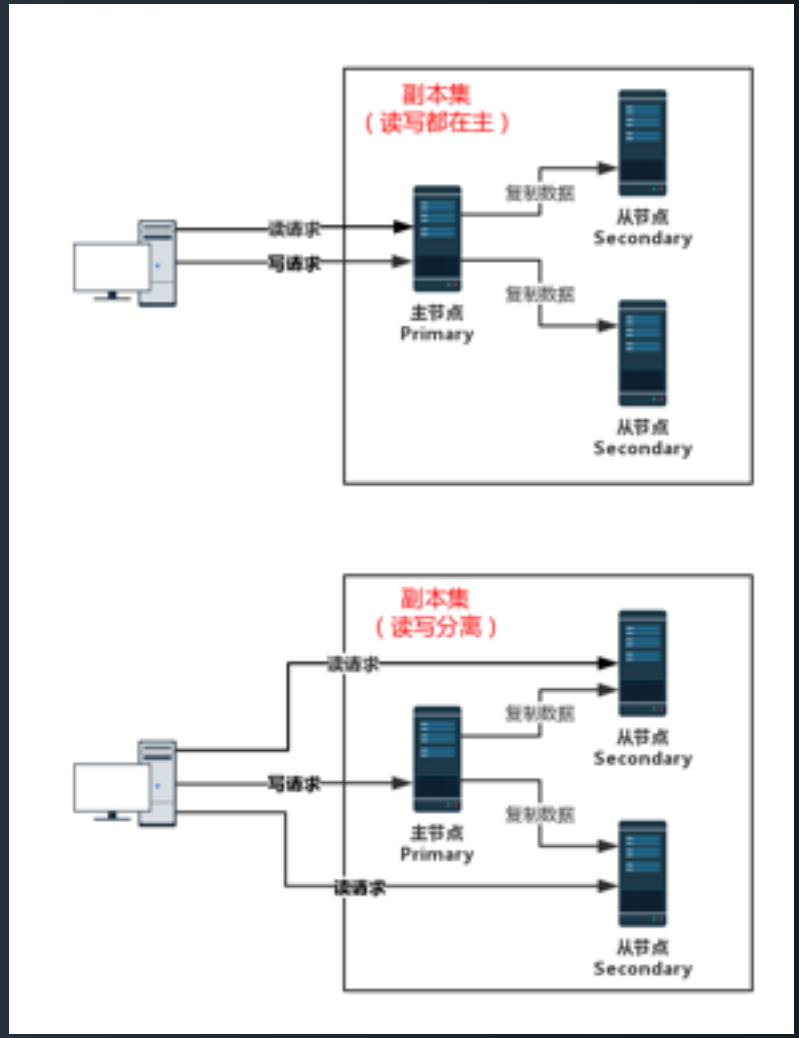
- 能够索引覆盖查询是最好的方式，直接通过索引就能得到结果，不需要加载文档获取数据。
- 创建和使用复合字段索引，不要使用多个单字段索引。通过多个索引交集的方式匹配查询的性能并不好，而且通常很难出现用上索引交集的情况。
- 复合索引创建时，要注意索引字段顺序，选择性高的尽可能放前面，范围查询放最右侧。同时要注意索引能覆盖到哪些查询组合。
- 创建索引时要考虑排序需求，如果索引顺序不能满足返回顺序，需要加载数据到内存进行排序。
- 对字符串建索引时可以考虑进行如下转换后再建索引：
  - 长字符串可以考虑通过一些冲突率比较低的哈希函数，比如md5进行转换，或者提取能保证比较好选择性的前缀进行索引创建。
  - ipv4字符串的话可以转换成int，ip+port可以转换成long(32bit + 16bit)
  - 需要*abc正则表达式匹配的，冗余一个倒排字段创建索引

事务(ACID)

- MongoDB不支持事务，但是MongoDB对单文档的写操作满足事务的原子性。
- 为了保证多个字段更新的并发安全问题，可以考虑将他们放在一个文档中，保证修改的原子性。
- 可以使用findAndModify操作实现find->update->get这样多步操作的原子性。

读写分离与读写一致性

- 副本集部署情况下，可以通过配置读写分离来分担读负载以及减轻读写锁争用。
- 由于主从复制同步的延迟，读写分离可能造成读写不一致的情况，需要在一致性和性能之间进行取舍。



复制幂等操作

- 为了保证从节点重放同步的写操作幂等性，主节点会将批量的update或remove操作，拆成多条针对单个文档的操作。
- 所以应该尽量避免批量更新或删除大量数据的操作，或者在业务低峰期进行。

片键

- 避免自增的片键导致写热点
- 避免小基数导致无法分裂
- 优化范围查询性能

优化读写性能

- 优化读性能
  - 足够的缓存大小设置（减少缺页引起的IO）
  - 索引（大数据量下主要手段，索引覆盖查询，复合索引字段选择性，为排序设计索引）
  - 内嵌冗余，一次获取
  - 读写分离（利用多个从节点分担读压力，减少读写锁争用，以及缓存，cpu等其他系统资源争用）

- 优化写性能
  - 缩简响应流程（WriteConcern设置(w, j参数)，性能与安全写入权衡）
  - 减少请求次数（批量写，写数组，bulk insert）
  - 合理的并发连接数（并不是越大越好，连接和线程占用大量资源，以及线程切换开销）
  - 优化IO（SSD固态硬盘替代机械磁盘优化随机IO性能，定容集合顺序IO）
  - 减少并发读写锁竞争（使用wt引擎，优化锁粒度，提高并发写性能。读写分离，减小读写锁竞争。）
  - 使用分片集群分担写负载

谢谢