Wukong 是知乎反作弊系统，它负责了 SPAM 的召回和处理。从 2016 年 8 月开始我们反作弊工程师对 Wukong 进行了局部重构（下称 Wukong 3.0）。在这里分享一下重构过程中对于缓存的设计和优化思路，主要解决了重复的查询和大数据量的查询造成 Mongodb 读压力太大的问题。

**名词解释**

**Action**

指用户的一次写行为，如回答创建，问题创建，点赞等。JSON 结构，一般可解释为“谁，什么时间，对谁，做了什么”， 包含 ActionType，UserID，UserIP， UserAgent，Created（行为产生时间） 等信息。

**Policy**

策略，指 SPAM 的召回策略。主要包含触发 (Trigger)，执行 (Handle) 两个部分，触发是指判定某行为是否为 SPAM 的触发条件，执行是指 SPAM 召回后的处理(如封号，禁言等)。两者都为一个 python 表达式。如下图所示，第一栏为策略名， 第二栏为 Trigger，第三栏为 Handle。



作者：曾涛  
链接：https://zhuanlan.zhihu.com/p/23509238  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

## Recent\_events

Policy 触发条件中的主要函数。主要用于在 Policy 的触发中召回最近一段时间内(如 30 分钟，1 小时)与该策略定义的相关维度(如 ActionType 相同，IP相同，UserID 相同)的 Action 集。如上图中的 same\_type\_events(60) 表示召回近 60 分钟内与该行为的 ActionType 相同的 Action 集。

# 诊断瓶颈

在 Wukong 系统中，所有 Action 的数据都存储在 MongoDB 中，并做了相关的索引提高查询效率。所有的 recent\_events 操作都从 MongoDB 中读取。

目前，Wukong 处理的主要逻辑就是对于线上实时的用户行为流中的每一个用户的写行为，通过多条 Policy 来从多角度的判断其是否是 SPAM 行为，并做相关处理。

本次 Wukong 的局部重构主要目的是解决如下问题：

1）减轻对 MongoDB 存储的读压力

2）放开对 MongoDB 存储读的条数限制。

对 MongoDB 的读压力主要体现在两方面：很多接近于重复的查询和大数据量的查询。

接近重复的查询是指，对于相隔时间很短内的多个相同类型的 Action，在某些 Policy 中（如图一的 Policy），对每一个 Action 都会调用 MongoDB 查询近 60 分钟内相同类型的事件（same\_type\_events(60))。这样的相隔时间段的多次 MongoDB 查询，其返回结果的一大部分是相同的，例如对于相隔10秒钟发生的两个 ActionType 为 ANSWER\_CREATE 的行为，在调用条件为“过去 60 分钟内 ActionType 相同“的 Action 集时，会 2 次调用 MongoDB 进行查询，而两次查询的返回结果大概率是 80% 以上的 Action 相同，只有小部分 Action 不同，造成大量不必要的IO。而一分钟内这样的查询可能存在上千次。

注: 这里只是对某些策略存在这些问题。对于其它策略如查询维度增加相同 ip 或相同 UserID 时，因为这类查询针对性强，且后端对 MongoDB 做了合理的 index，MongoDB 的性能完全靠谱，所以对这些查询并不是本次重构中所关注的重点。

大数据量的查询是指，recent\_events 调用取回条数大于等于 1000 条的查询，过多的这些查询导致 MongoDB 服务器负载过高。为了避免 SPAM 爆发时由于 MongoDB 的读性能问题造成处理队列阻塞，我们对 MongoDB 的每次查询条数设了上限，默认为 1000 条，这在某种程度上造成某些策略 recent\_events 中 Action 集的返回结果严重失真，但如果去除这个限制，则会导致 MongoDB 的响应时间线性增长，同时随着返回结果集限制等放开，每条策略的平均处理时长会大大增加，系统的的处理性能会大幅下降。

## 优化方案

为了解决上述的两个问题，我们的想法是采用的「Redis 缓存 + 本地缓存的方法」，开发一个缓存组件 BigCache，将接近重复的查询和大数据量查询从 MongoDB 中剥离，MongoDB 中只负责数据量小和针对性强的查询。

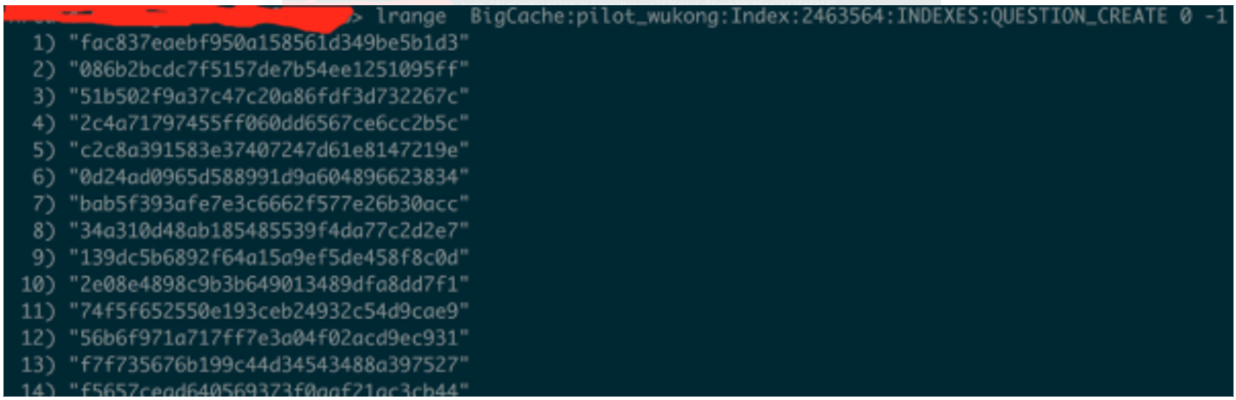
## BigCache

Redis 缓存数据库由 MetaCache 和 IndexCache 两部分组成，数据都实时更新。

MetaCache 存储 Action 的原始信息，key 为原始信息的 md5 值，value 为 Action 的详细信息。

IndexCache 存储 Action 的索引信息，按照时间维度进行分片（默认 10 分钟）存储, key 由 time\_sheet 和 index 维度组成，value 为该时间段内所有发生的该 index 维度的 Actions 的 md5 值，存储为双向链表。如下图所示。  


MetaCache 数据结构



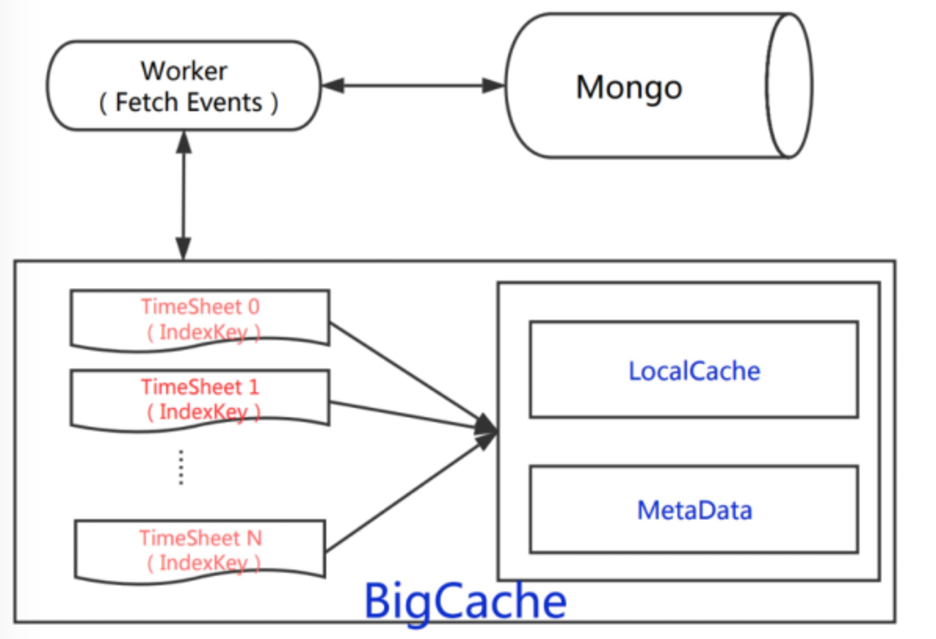
IndexCache 数据结构

本地 worker 中的 LocalCache 采用2层的 dict 实现，第一层的 key 为时间分片，第二层存储具体的 cache 信息，这样做的好处是方便定时的清理过期缓存。

在本地同时缓存近 2 小时的 Meta 数据和近 2 小时内的 Index 数据(如以 10 分钟作为分片依据，那么查询近 1 小时数据（精确到秒）可分为 7 片查询。我们的Action数据流是基于时间序的，这里假设时间延迟不会超过10分钟，因此 7 个分片中只有第 0，1 分片的 Index 数据是存在动态更新的可能，需从 Redis 中获取， 而对第 2~6 片的 Index 数据不会再改变，可以存储到 LocalCache 中。这样大部分的 Index 查询和 Meta 数据查询可以在 LocalCache 中完成，只有 LocakCache 未命中的才会查询 Redis 的 IndexCache 和 MetaCache。在我们的应用场景下，LocalCache 的命中率极高，大大减少了对Redis的重复请求。

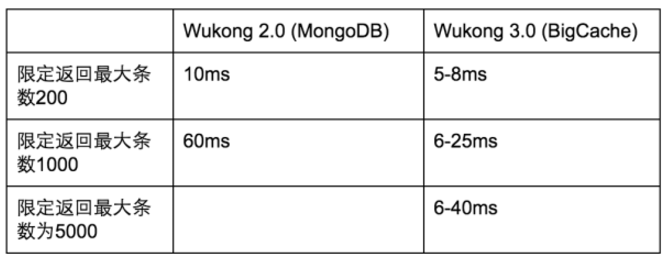
**使用 BigCache**

采用该设计后，对于一次 recent\_events 查询，首先根据查询条件判断是否通过 BigCache 查询，如果是，我们将根据给定的查询 duration 和 Action Created 的时间拆分成若干个时间分片，然后分片查询 BigCache 的 Index 表 (Redis+Local) 来获取 Action 的 MD5 列表，然后通过基于该列表查询 Meta 表 (Redis+Local) 获取到具体的 Action 信息并返回。而对于其它查询，依然通过MongoDB 返回如下图所示。



# 性能对比

使用该缓存方案后 recent\_events 的执行时间对比如下表：

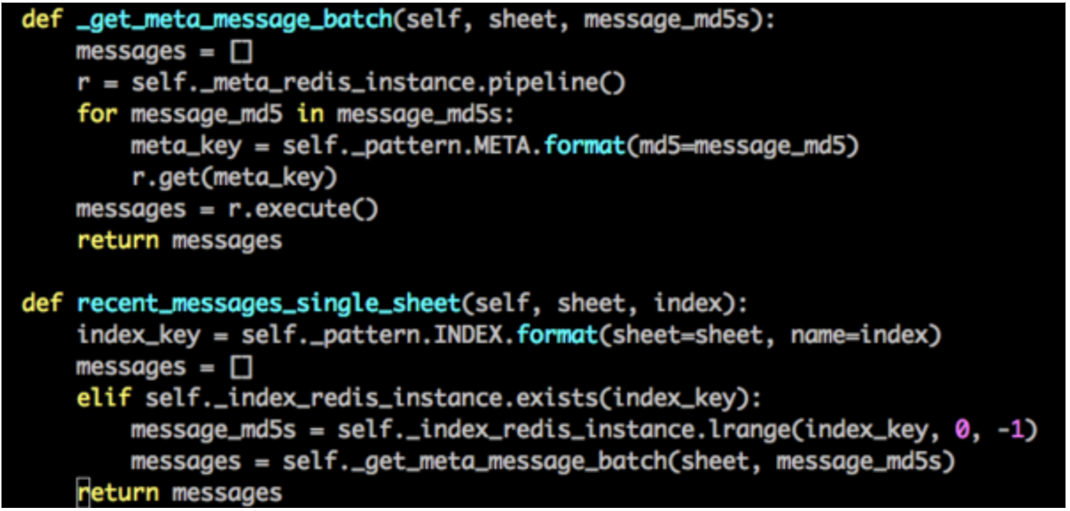


从上表可以发现采用 BigCache 后性能有很大提升，尤其体现当返回条数上限越大时，性能并未线性下降。主要原因在于将热数据存储在本地减少了对同一数据 70% 以上的重复网络传输和查询，且本地缓存命中率高。

# 再优化

从上面的性能对比表可以发现 Wukong3.0 的查询返回平均时间是一个范围，经过对以上方法的再思考和热点的再分析，为了进一步降低 recent\_events 的最大平均返回时间，我们还采取了以下两个措施：

* 采用Redis Pipeline，减少了未在 LocalCache 中命中的 Action 原始信息查询 Redis 的次数，减少 Round-Trip。代码如下：

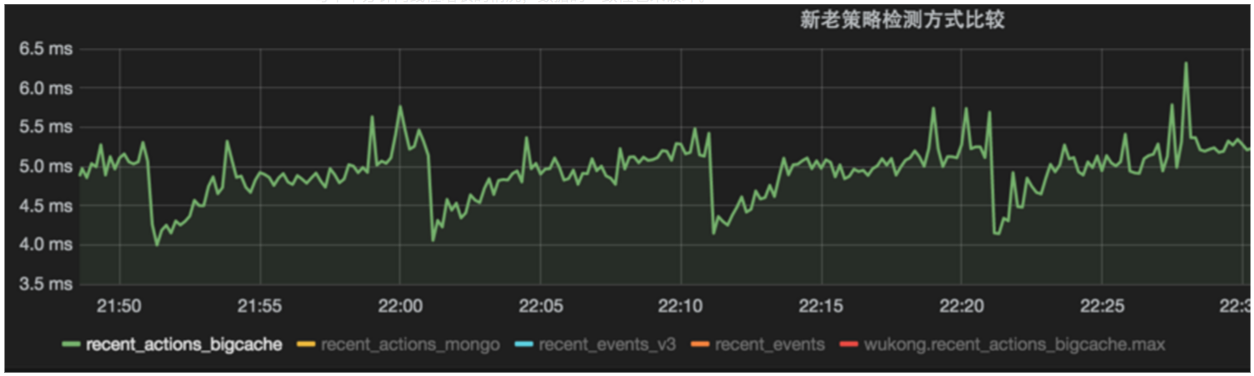


作者：曾涛  
链接：https://zhuanlan.zhihu.com/p/23509238  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

IndexCache 增量更新。我们在 LocalCache 中并未存取当前时间分片 (即第0分片) 的 IndexKey，因此每次查询时都需从 Redis 的 IndexCache 中获取该分片的全量 MD5 列表，其处理时间与其所存储的当前分片的 MD5 列表的长度有关。经测试，对于一个 MD5 列表长度为 1000 的 key，全量取回的查询平均耗时为 8ms，而长度为 5000 的 key，平均需要 40ms 左右。而该列表的大小会在每十分钟的第 0 分钟开始线性增长直到该分片的第 10 分钟，造成 recent\_events 平均处理时间在每个十分钟内也线性增长。对于第 2-6 分片，由于 IndexKey 已经在存储在 LocalCache 中，并不影响处理时间。

对于这个问题，我们使用增量更新的方式优化了 BigCache IndexKey 的存储。在 LocalCache 中也缓存第 0 和 1 分片的 MD5 列表，并由 BigCache 负责增量更新。具体的做法是这样每次向 Redis IndexCache 查询时，不取全量，取而代之的是 LocalCache 以该分片已有 IndexKey 的信息为依据向 Redis 请求该时间段内的增量数据，并更新自己的缓存内容。这样每次向 Redis IndexCache 请求的数据更新量平均都在10以内，且不会出现每个十分钟内线性增长的情况，数据的一致性也未破坏。

经过以上两个优化方案后，recent\_events 的处理时间平均稳定为 4-6ms, 相较于只采 MongoDB 的最大返回条数限制为 1000 时的 60 ms 平均处理时间提升明显。如下图所示。



通过以上的步骤，基本解决了本文开头提到的很多接近于重复的查询和大数据量的查询对于 MongoDB 的查询压力问题。同时取消 recent\_events 返回条数的限制后，相关策略的召回量提升20% 以上，且并不会对系统的性能造成大的影响。

**作者**：曾涛，[翟峰](https://www.zhihu.com/people/zhaif)，[周奥特](https://www.zhihu.com/people/AoteZhou/)

同时感谢反作弊团队其他同学的帮助

作者：曾涛  
链接：https://zhuanlan.zhihu.com/p/23509238  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。