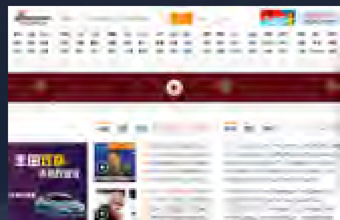
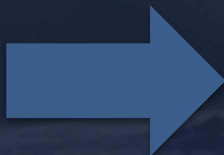
今日头条
推荐系统架构设计实践

金敬亭

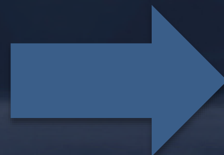
互联网时代内容分发的变革

门户



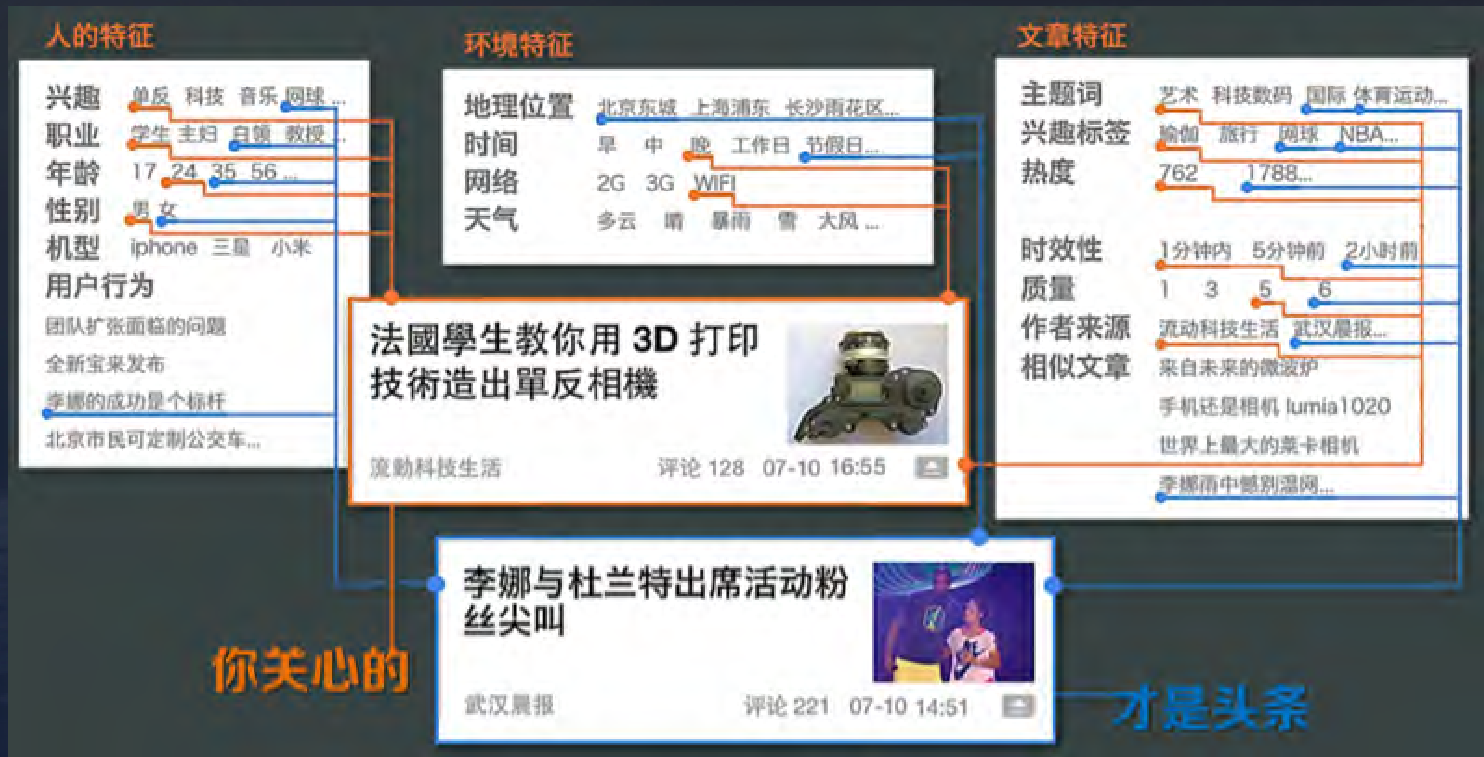


社交媒体&社交网络



推荐引擎

今日头条本质上是一个强大的智能推荐引擎



截至2016年12月底

- 头条DAU：7800w；
- 头条MAU：1.75亿；
- 单用户日平均使用时长： 76分钟；
- 用户行为峰值： 150w+ msg/s;
- 每天训练数据：300T+（压缩后）；
- 机器规模：万级；

系统架构

用户画像

面临挑战

- 期望快速反馈：10min内；
- feature 数量：200+;
- 存量用户数和每天的用户行为数据量巨大；
- 在线存储：读写吞吐高，要求延时低且可预期；

流式计算

- 实现Storm Python框架
  - 写MR的方式写Streaming Job；
  - Topology用Yaml描述，代码自动生成，降低编写job成本；
  - 框架自带KafkaSpout，业务仅关注拼接和计算逻辑；
  - Batch MR相关算法逻辑可以直接复用在流式计算中；
- Job数：300+；
- Storm集群规模：1000+；

在线存储-abase

基于Rocksdb的分布式存储系统：

- 基于文件的全量复制和基于rocksdb自身WAL的增量复制；
- 内建和back storage强一致的key级别LRU cache；
- 基于bucket的sharding和migration；
- 基于compaction filter的延迟过期策略；

在线存储-abase

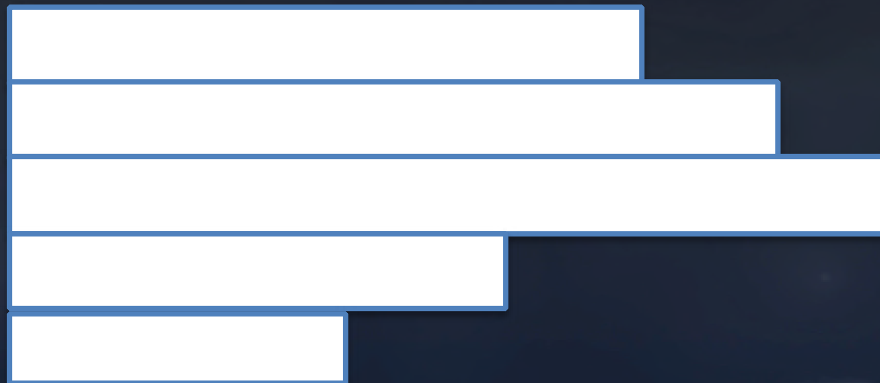
- 数据量：压缩后，单副本85T；
- QPS：读360w、写40w；
- 内建Cache命中率：66%；
- 延时：avg 1ms、pct99 4ms;
- 机器数：单副本40台，SSD容量瓶颈；

架构抽象

推荐召回-典型策略

用户兴趣标签

德甲	0.3
电商	0.2
O2O	0.2
搞笑	0.1
历史	0.1
军事	0.1



离线更新tag倒排索引

根据兴趣标签拉取相应文章并rank top 结果

以兴趣分类召回为例，实际上这里的tag可以是各种显式兴趣标签和隐式兴趣特征

推荐召回–抽象

- 离线倒排更新

  fid : $(gid_1, score_1) \rightarrow (gid_2, score_2) \rightarrow \ldots \rightarrow (gid_n, score_n)$

- 在线search

  $(fid_1, score_1)(fid_2, score_2)\ldots(fid_k, score_k) \rightarrow$

  $(gid_1, score_1)(gid_2, score_2)\ldots(gid_n, score_n)$

- 其他组件

  filter、merge、boost等；

正排–相关数据

- 推荐系环节，如：召回、过滤及预估等需要文章正排；
- 文章包括图文、视频、UGC内容，正排属性主要包括：

属性	内容
文章属性	• 创建时间； • 过期时间；
文本信息	• Keyword； • Topic；
动态属性	• 阅读数； • 展现数；

正排-痛点

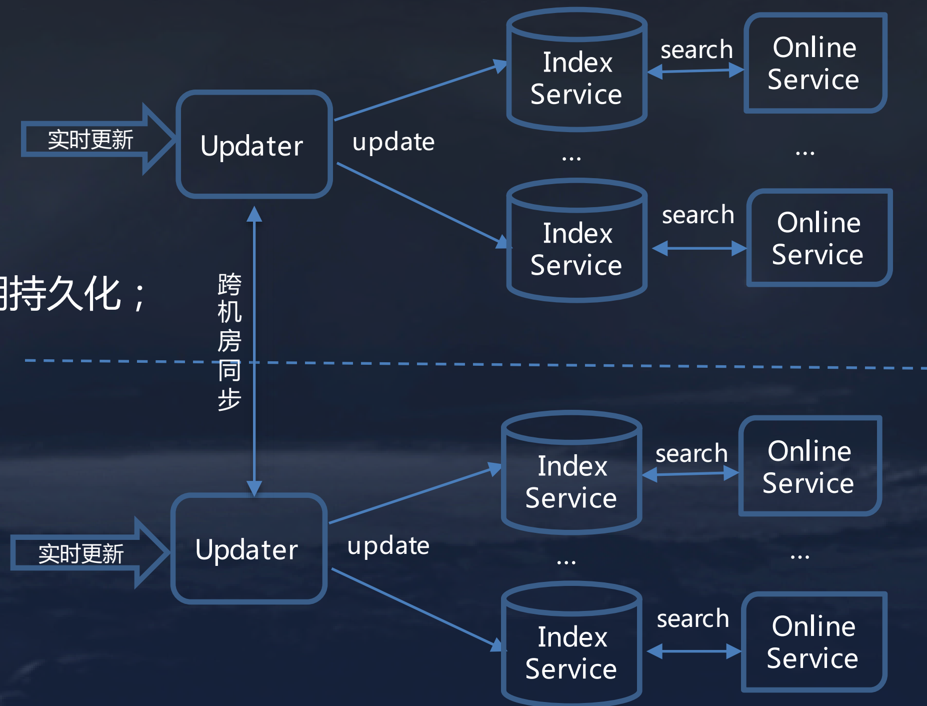
- 各模块各自维护相应的离线和在线，稳定性和时效性无法保证；
- 格式多样: json、msgpack、protobuf等；
- 字段重复、含义不一致 ，存储不统一：mc、redis等；
- Trouble-shooting成本比较高；

正排-统一方案

- 200+字段，由protobuf IDL描述，按”簇”存储；
- 统一离线刷新框架，保证高时效性和稳定性；
- 统一存储和对外接口；
- 提供完善的debug工具：查询正排内容、更新时间等；

存储方案-index_service

- 基于lmdb存储， MVCC、读写相互不阻塞；
- 内存映射，0拷贝，如：过滤；
- 存放在共享内存，实现TTL、compaction、定期持久化；
- 给召回提供算子：filter、merge、boost等；
- 单机1w QPS，pct99<1ms，没有明显长尾；



成本压力

架构1.0-局部优化

基于Thrift+Python多进程模型：

- 并行化：gevent、线程池；
- C++扩展 ： cython、boost.python;
- 解释器：pypy，需要适配依赖，主要用于离线；
- 基础库so版本： protobuf、thrift protocol、json；
- 服务处理时间和用户感知时间gap；

架构1. 0-痛点

痛点主要来自Python多进程模型：

- 性能无法满足策略优化，如：增加预估条数；
- 单机QPS低，内存瓶颈，CPU用不上去，浪费严重；
- 只能堆机器提高吞吐能力；

架构2. 0

完全重构，拥抱C++11：

- Thrift Nonblocking Server 多线程模型；
- 机器数减少**60%+**；
- 平均延时下降**30%+**，PCT99下降**50%+**；

延时控制

Cache问题

在推荐系统中Cache并非总是万金油：

- 一个用户刷新两次，不能重复，但搜索同一个query，短期内返回相同结果；
- 实时的用户Profile和模型特征，Cache会影响指标；
- 召回候选集、倒排拉链实时更新；

Cache应用

但Cache依然是系统的重要组成部分，降低延时，避免雪崩：

- LocalCache、分布式Cache、共享内存；
- 一些招数
  - 空值回填；
  - 异步刷新；
  - 写时更新 or 写时删除；

Pool化

- 内存池：tcmalloc、jemalloc；
- 对象池：复用对象，减少内存申请释放，实现warmup、shared_ptr deleter自动归还；
- 线程池：并发执行，降低延时；
- 连接池：多RPC副本连接管理、长连接复用；

并行化

- openmp
  - ➢ 一行代码串行变并行；
  - ➢ 滥用，频繁创建和销毁线程；

```
#pragma omp parallel for schedule(dynamic, 1) reduction(+:success_num)
for (size_t i = 0; i < clients_.size(); ++i) {
    ...
    client_[i]->fetch_parameter(response, request);
    ...
}
```

- 线程池
  - ➢ 方便监控：空闲线程、pending task；
  - ➢ 配合future/promise更灵活；

但不是并行度越高越好，多线程的overhead；

大扇出

- 获取feature需2000+的RPC调用，串行肯定是无法满足需求。
- 基于openmp的扇出调度：

```
#pragma omp parallel for schedule(dynamic, 1) reduction(+:success_num)
for (size_t i = 0; i < clients_.size(); ++i) {
    ...
    client_[i]->fetch_parameter(response, request);
    ...
}
```

- 问题：openmp线程数太多，CPU load太高，但利用率低。

大扇出-优化

- Thrift client使用同步模型，需要更多的线程来实现并发；
- 实现基于异步IO的thrift RPC fanout，将IO和序列化反序列化分离：

```
reactors_ = std::make_shared<xactor::FanoutHandlers>(threads_, nr_reactors);
for (size_t i = 0; i < shards.size(); ++i) {
    ...
    reactors_->addHandler(std::make_shared<xactor::ThriftHandler>(hosts, ports, ps_conn_timeout_));
    ...
}
auto decoder = [&] (int i, xactor::ThriftHandler *h) {
    ...
};
auto decoder = [&] (int i, xactor::ThriftHandler *h) {
    ...
}
auto action = reactors_->open();
action->parallel_encode(encoder_parallel, encoder);
action->fanout(fanout_timeoutms, retry_times - 1, fanout_max_fails);
action->parallel_decode(decoder_parallel, decoder);
```

- 用更少线程实现扇出调用，CPU load**下降20%+**；

大扇出

- 长尾概率： (1-0.999**2000)=86.5%；
- 传输大量小包，sys态占比高；
- 增加Proxy，减少扇出和小包；

框架层面

- thrift bugs： 10 patches，性能不高；
- grpc or fbthrift or 自研 ？

使用Fbthrift Server模块CPU使用率下降20%+：

- 传输协议&IDL向下兼容；
- IOBuf应用，减少copy、避免一次性申请大内存；
- 支持全异步调用：扇出、并发；
- ……

可用性

- 降级机制：自动降级和降级平台；
- 调度优化：
  - 封禁、探活、解禁；
  - 主动overflow：丢弃长时间pending task；
  - 熔断：降低后端压力，防雪崩；
  - 优雅退出：运维0拒绝；
  - 多机房调度：快速流量迁移；

未来挑战

- 数据、规模继续爆炸；
- 多IDC的挑战；
- 系统复杂性、ToubleShooting；
- 资源利用率、资源调度；
- ……

Q&A