

Gamma Go

郭振兴 郭嘉梁 李豪
中科院计算所



挑战双十一实时计算



TIANCHI天池

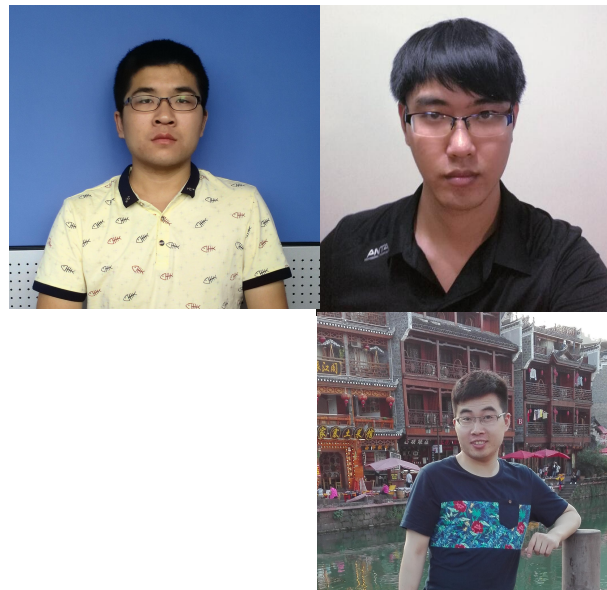
团队介绍

队名：Gamma Go

口号：Gamma Go你比Alpha多两Go

来自：计算所高性能中心

成员介绍：



| 成员 | 主要职责 |
|---------|----------------|
| 郭振兴（队长） | 初赛，复赛缓存模块 |
| 郭嘉梁 | 初赛，复赛IO控制模块 |
| 李豪 | 复赛架构搭建，排序和索引模块 |

目录

问题分析

系统架构

关键问题





优化方案

提升空间

技术展望

问题分析

对100GB的订单信息建立索引，支持后续四种查询

- 按照orderId查询订单信息  单条查询
- 按照buyerID查询订单信息  顺序查询
- 按照goodID查询订单信息 
- 对指定goodID的某字段求和  聚集查询

查询：采用何种索引方式？

- Hash索引，**构建快速**，不适合顺序查询和聚集查询；
- B+Tree索引，构建速度慢，**适合顺序查询和聚集查询**；
- LSM Tree (Log Structured Merge Trees) 索引，构建速度慢，**牺牲部分读性能来提升写性能**。

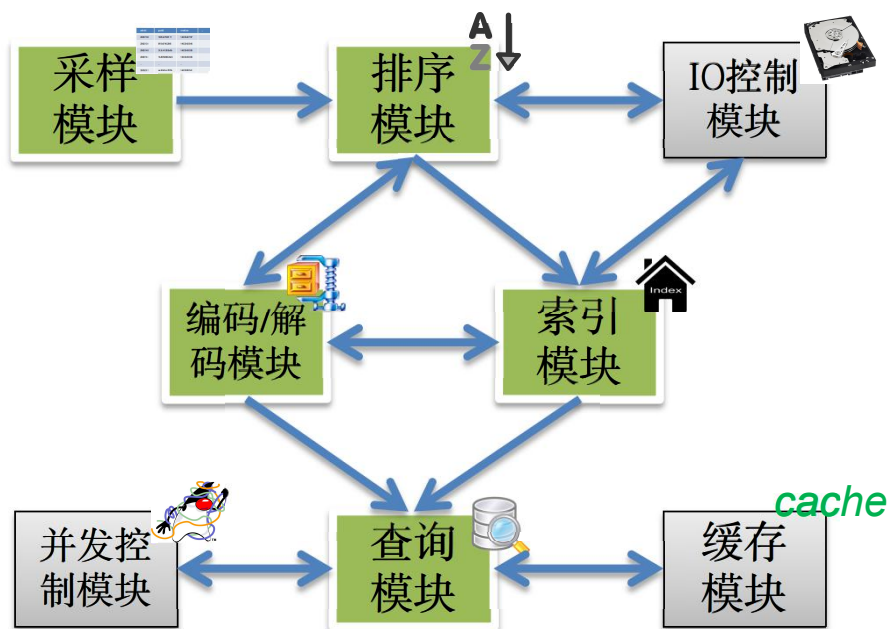
读取：如何组织数据？

- 非压缩方式，**解析速度快**，文件体积大；
- 压缩方式，解析速度略慢，**文件体积小**。

系统架构

总体结构

- 采样模块
- 排序模块
- 索引模块
- 编码/解码模块
- 查询模块
- IO控制模块
- 并发控制模块
- 缓存模块



图：系统整体架构图

系统架构——排序

类TeraSort排序

采样

获取数据分布、推断类型

分类

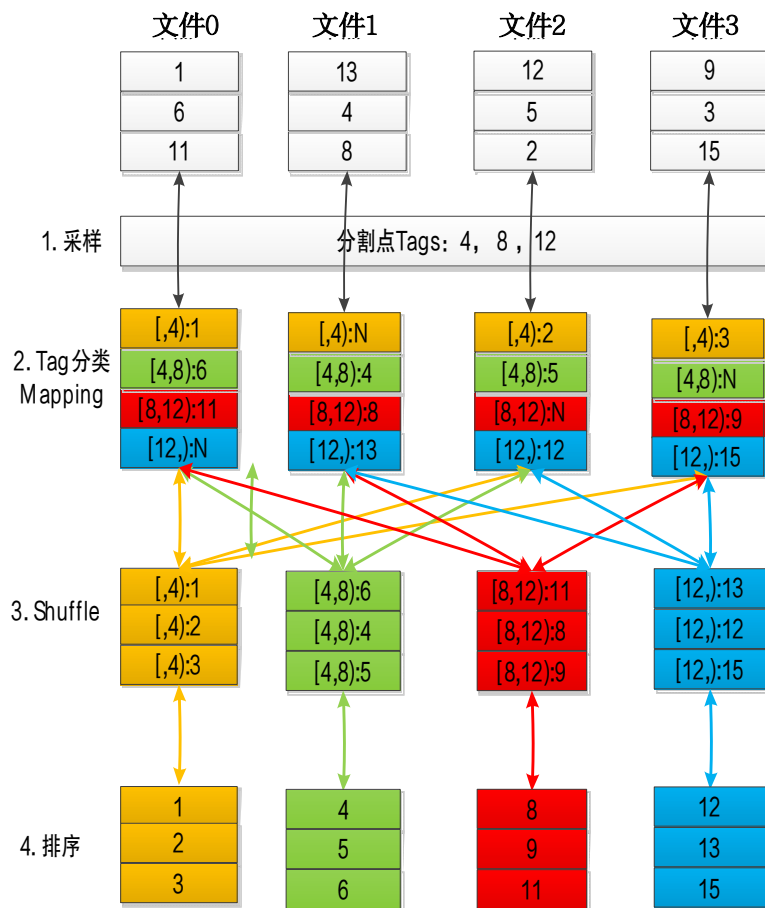
数据均匀划分到桶

桶间有序，桶内无序

桶内排序

桶足够小，内部排序

全局有序



图：分布式排序过程示意图

<http://sortbenchmark.org/>

系统架构——索引

改进的B+Tree索引

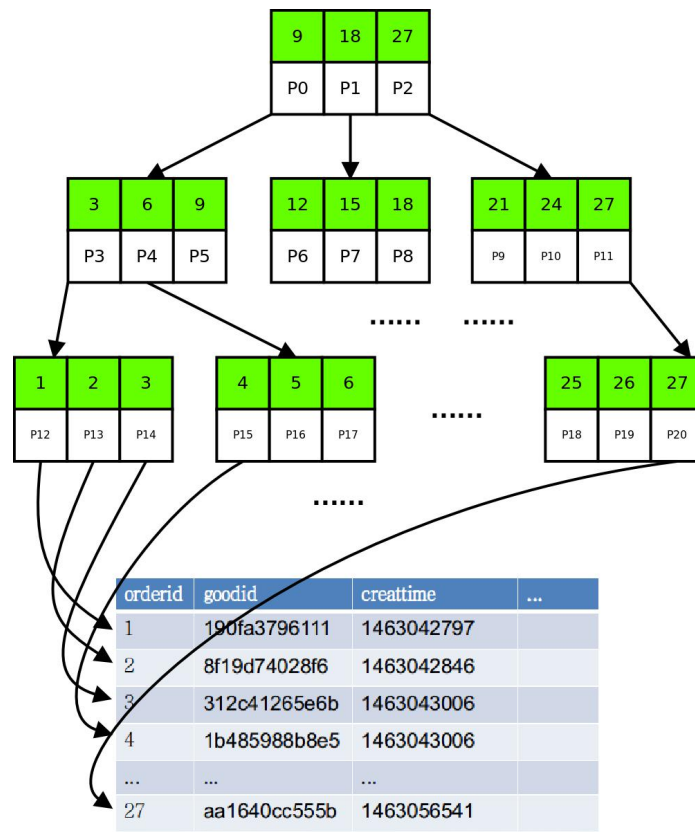
树更加平衡

- 数据已排序，构建速度更快
- 数据不再变化

关键查询路径常驻内存

- 减少读索引的I/O次数
- 针对该项目，缓存前两层

| | 不同值个数 | B+Tree中间层 节点个数 | 中间层占用空间 |
|---------|-------------|-------------------|---------|
| orderid | 400,000,000 | 737 | 9.2MB |
| goodid | 4,000,000 | 159 | 0.8MB |
| buyerid | 8,000,000 | 200 | 1.2MB |



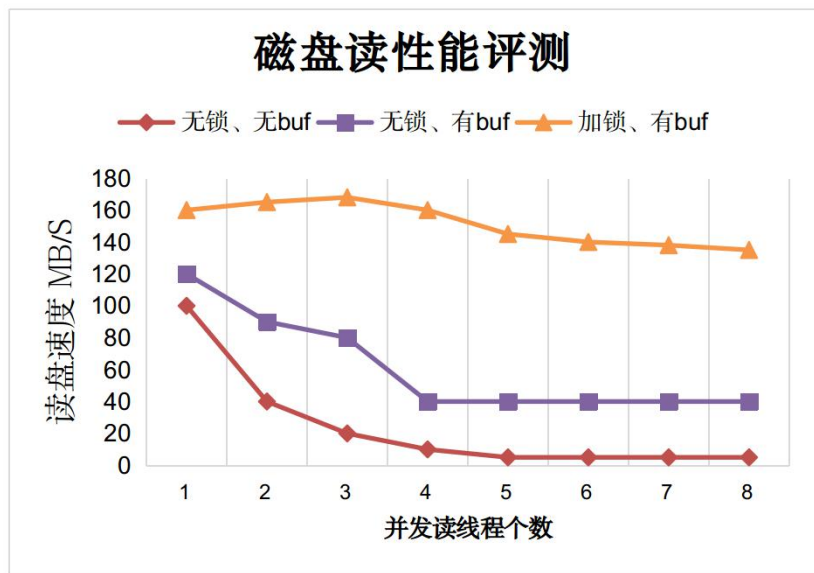
图：索引示意图

关键问题——磁盘读写控制

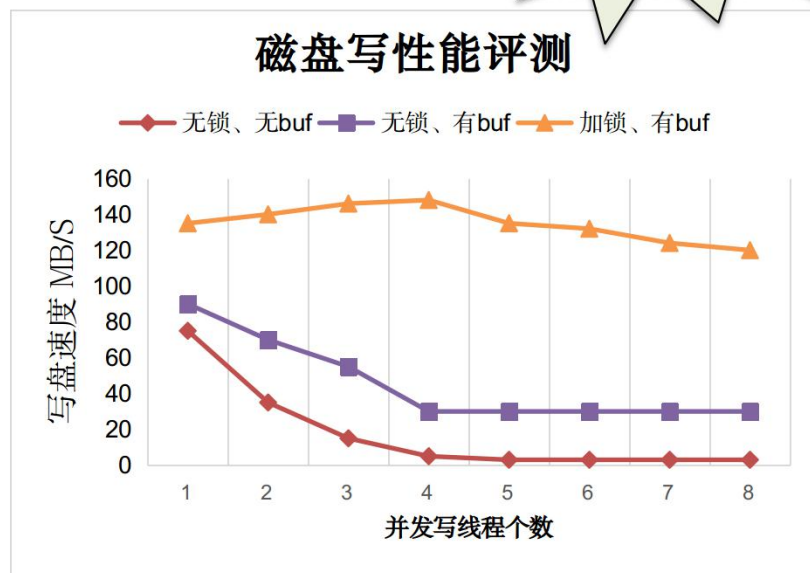
顺序大粒度磁盘IO

使用锁避免由竞争引起的频繁磁盘寻道

无控制的多
线程读写性
能非常低!



图：磁盘读性能评测

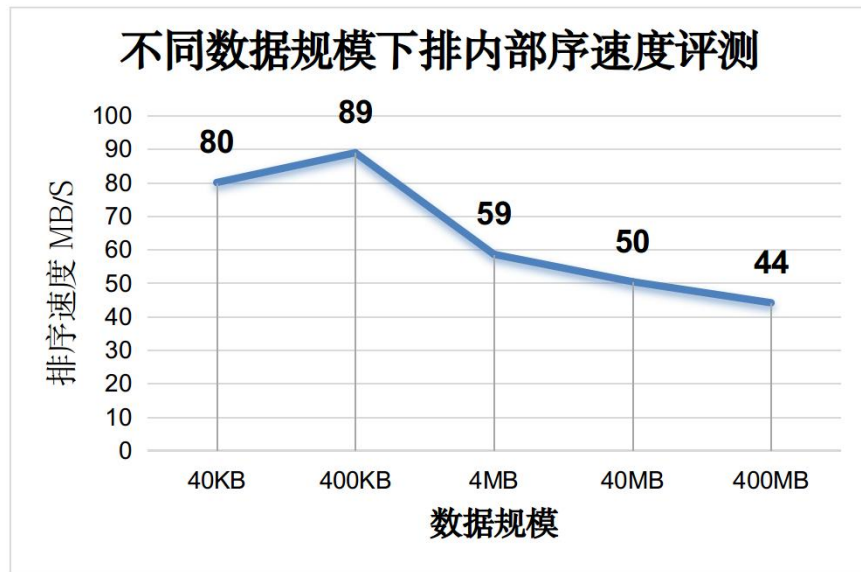


图：磁盘写性能评测

关键问题——磁盘读写控制

磁盘读速度与排序速度的匹配

- 无读写控制时，三块盘聚合带宽
 $35\text{MB/s} \times 3 = 105\text{MB/s}$
- 控制良好时，三块盘聚合带宽
 $150\text{MB/s} \times 3 = 450\text{MB/s}$
- 桶大小为40MB，8核排序速度为
 $50\text{MB/s} \times 8 = 400\text{MB/s}$



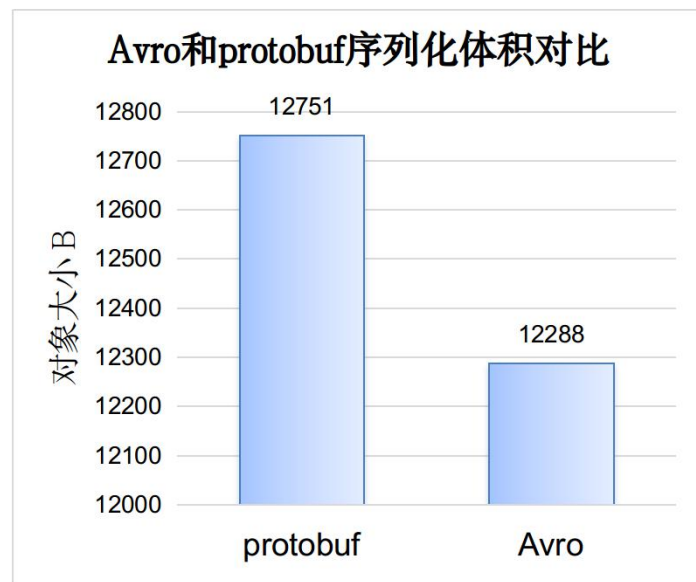
排序过程中，IO控制是关键！

关键问题——数据压缩

数据压缩体现在编码过程中

| 序列化方法 | 跨语言 | 支持修改scheme | 只序列化数据 | 速度 | 结果大小 |
|-----------------|-----|------------|--------|-----|------|
| Java原生序列化 | × | × | × | 非常慢 | 很大 |
| Kryo | × | × | × | 较快 | 较小 |
| Avro | ✓ | × | ✓ | 快 | 小 |
| Protocol Buffer | ✓ | ✓ | ✓ | 快 | 小 |
| Json | ✓ | × | ✓ | 很慢 | 较大 |
| XML | ✓ | × | ✓ | 很慢 | 较大 |

表：常见序列化方法对比



图：Avro和protobuf序列化体积对比图

<https://github.com/thekvs/cpp-serializers>

<http://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking>

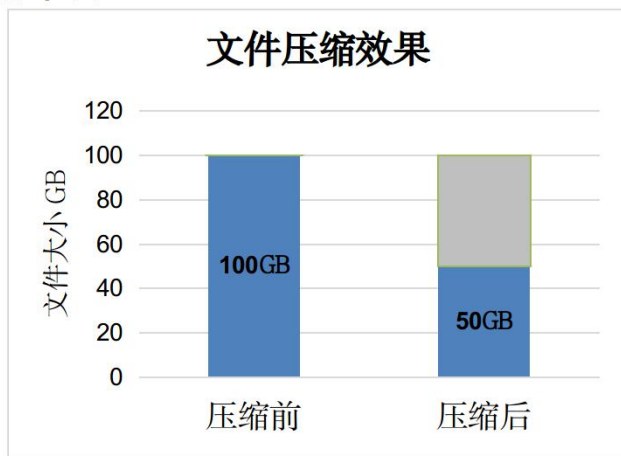
关键问题——数据压缩

压缩方式

- 对Key编号，只存储编号
- 基于类型推断的压缩方式，使用Avro编码

压缩效果

- 压缩率在40% ~ 70%之间
- 决赛数据压缩率为**50%**



图：文件压缩效果图

| 推断类型 | 存储格式 |
|--------------|--------------|
| BOOLEAN | 单字节的0或1 |
| LONG | 变长整数 |
| DOUBLE | 字符串 |
| STR_UUID | 字符串+byte[16] |
| STR_HALFUUID | 字符串+byte[8] |
| UUID | byte[16] |
| STRING | 字符串 |

表：不同数据类型的编码方式

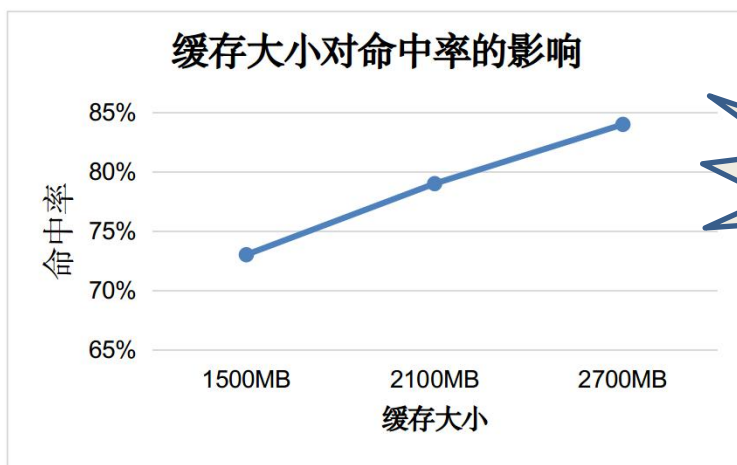
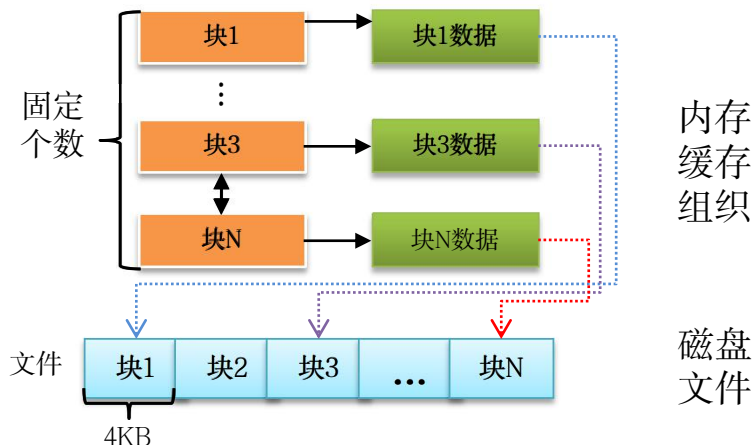
关键问题——数据缓存

缓存方式

- 基于块的缓存策略
- 块由<文件名, 块号>二元组标识

缓存内容

- B+Tree索引第三层节点
- 订单相关数据



关键问题——数据缓存

读取块1



读取块1

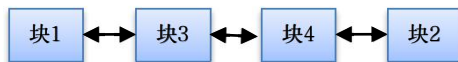


FIFO策略



Disk 1

...



FIFO策略



Disk 3

LRU策略

- 查找需要修改链表结构
- 可能的加锁策略
 - synchronized
 - ReentrantLock

FIFO策略

- 查找不需要修改链表结构
- cache miss才会修改结构
- 采用读写锁
 - 查找仅需加读锁
 - cache miss时加写锁

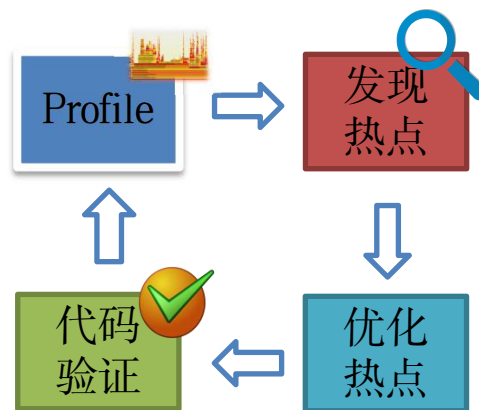
进一步拆锁

- 针对不同磁盘做缓存
- 进一步提高并发度

关键问题——优化热点代码

采用迭代方式优化热点代码

1. 使用Profiler评测代码
2. 发现热点
3. 优化热点
4. 验证优化结果
5. 下一轮优化



<https://github.com/brendangregg/FlameGraph>

<https://github.com/dcapwell/lightweight-java-profiler>

关键问题——优化热点代码

优化字符串拆分

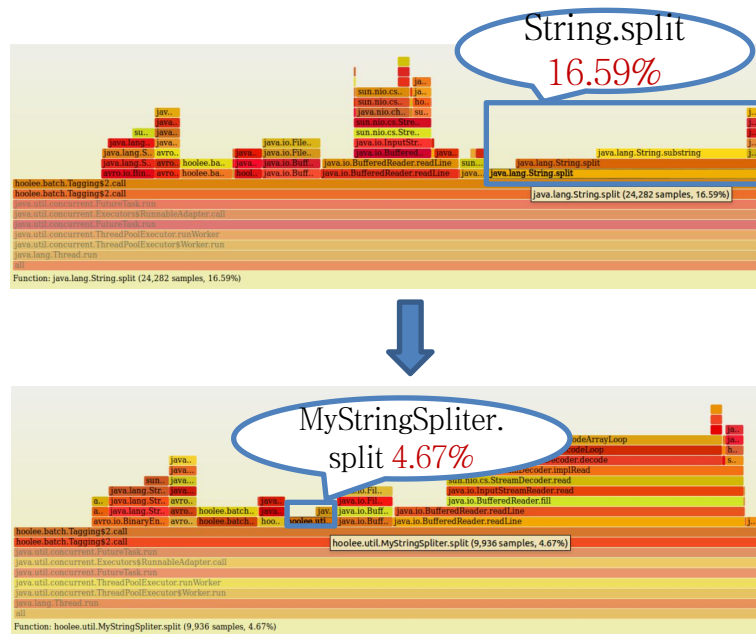
- 使用substring()替代split()
- split耗时从17%降到5%

优化Avro解码过程

- 去掉安全性拷贝
- 减少无用逻辑
- 解码耗时从60%降到20%

优化采样过程

采样过程从4分钟降到20秒



图：字符串拆分优化效果图

我们本可以做的更好

堆外内存

堆外内存导致内存泄漏

- Direct Memory不像新生代、老年代那样，有GC直接管理
- 只能等到Full GC时，"顺便地"清理掉

堆外内存效申请、释放率不高

- 手动调用Cleaner.clean()方法释放堆外内存
- 申请和释放代价太高，使用堆外内存效果不理想

我们本可以做的更好

减小锁粒度

- 通过减小锁粒度增大缓存并发量
- 类似于ConcurrentHashMap中的锁策略

GC友好

- 过高的查询并发增加新生代内存占用
- 内存担保失败，进而频繁触发老年代CMS GC
- 调整新生代与老年代比例
- 控制并发查询速度

技术展望

SSD下的大数据应用

- SSD成本降低、容量变大（TB）、IOPS高
- 特别适用于的查询场景（**读多写少**）

存储计算融合

- 存储设备加入弱计算能力
- 缩短数据通路，减轻CPU压力

用分布式方案解决更大规模并发查询

- 分布式join

Thanks & QA

