



# Apache Pulsar 的设计与实现

- [xiaorui.cc](http://xiaorui.cc)

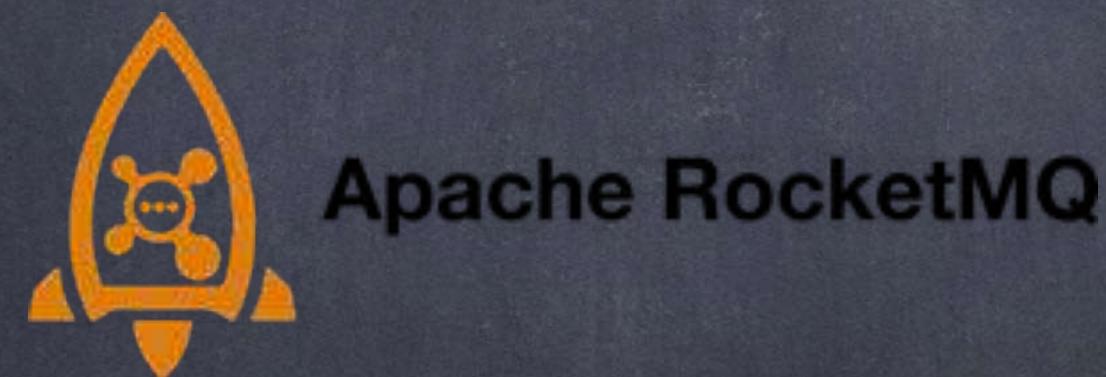
- [github.com/rfyiamcool](https://github.com/rfyiamcool)



pulsar intro



# 对比



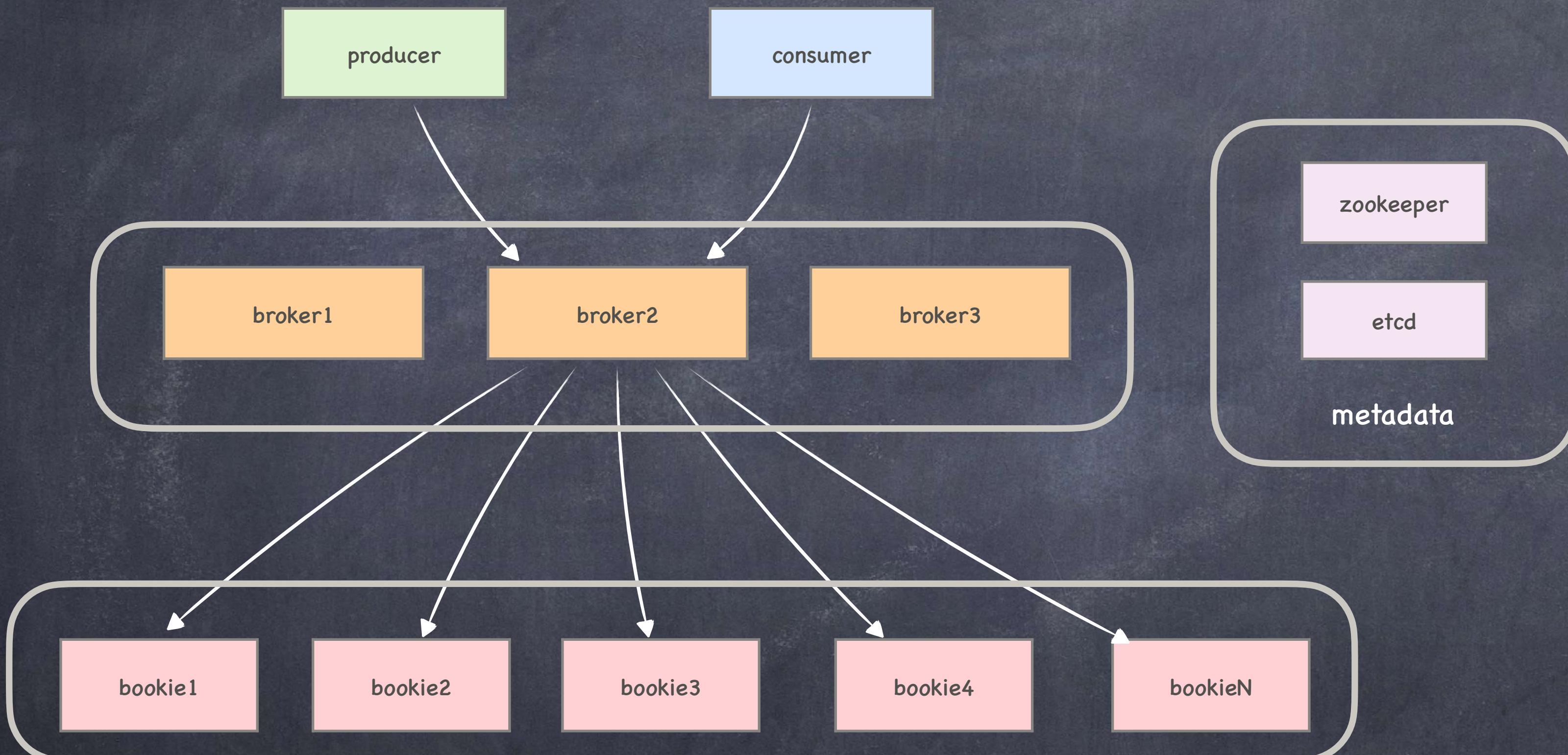
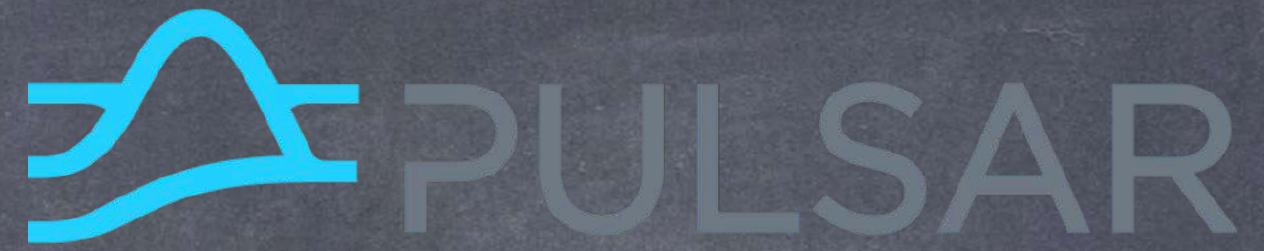
\* rabbitmq

\* kafka

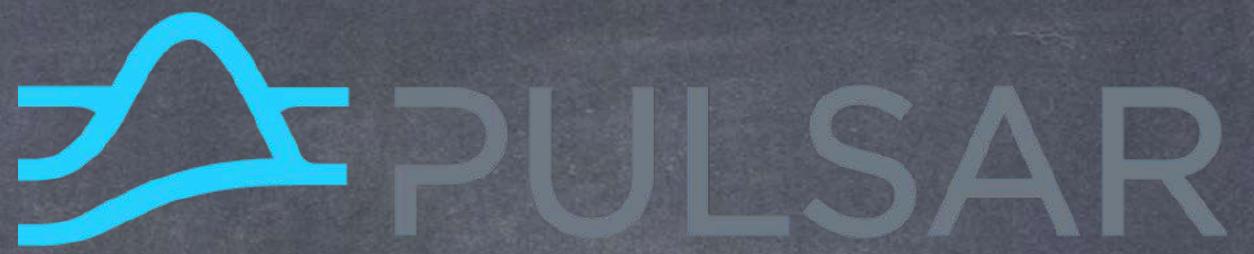
\* rocketmq

\* pulsar

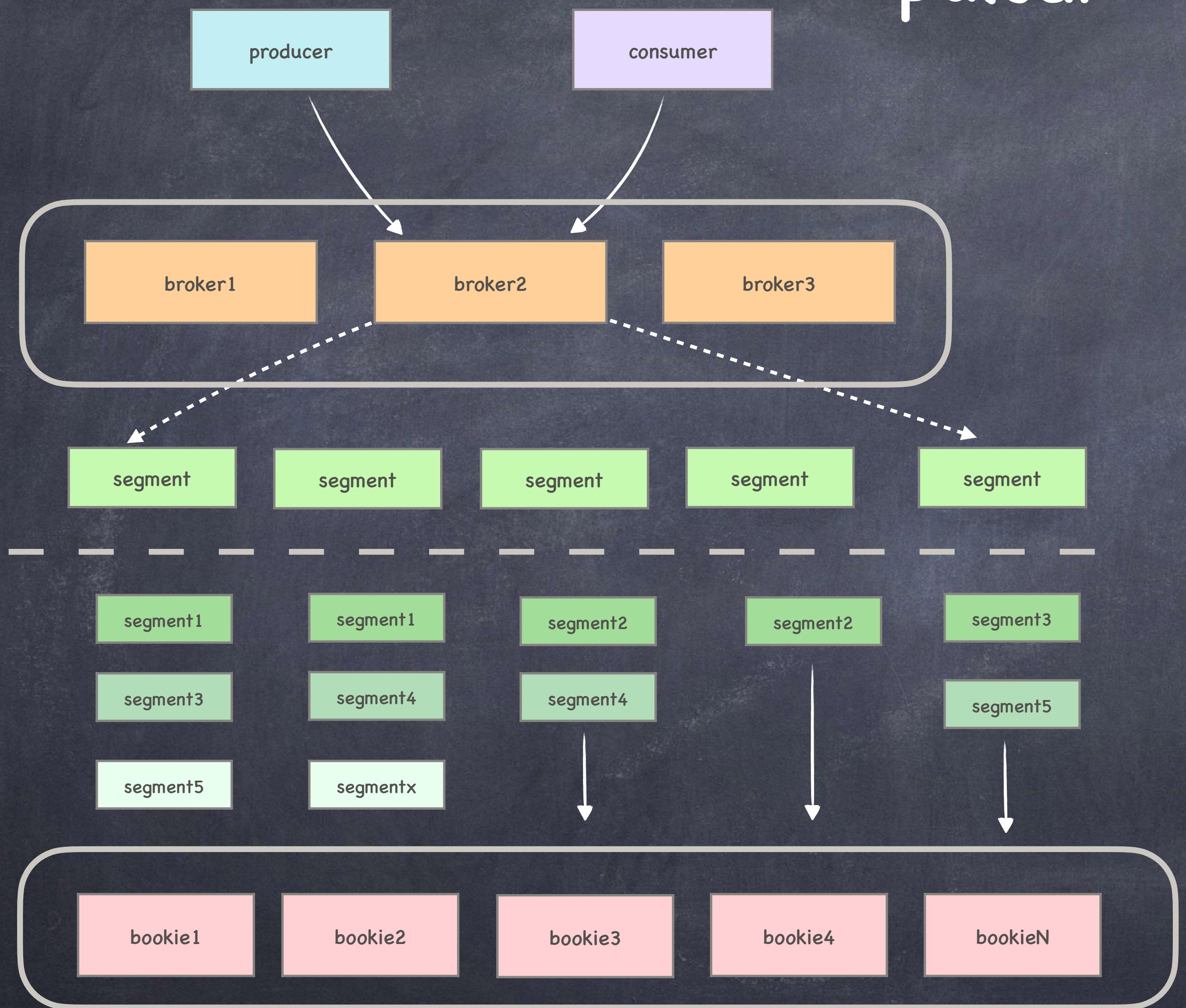
# pulsar



- 采用算存分离的架构
- 支持多租户
- 支持多数据中心
- 适配云原生
- 动态负载均衡
- 优化 kafka 的数据倾斜及水平扩展问题
- 解决海量 topic/partiton 的性能问题
- 挂点支持 kafka rabbit rocketmq 协议
- ...



# pulsar



## ● broker 计算层

- broker 无状态, 状态存于 metadata

- topic 被 load manager 均衡到不同的 broker

- 客户端对扩缩容和故障切换无感知

## ● bookie 存储层

- 有状态, 存储消息

- 条带化写入不同的 bookie 上

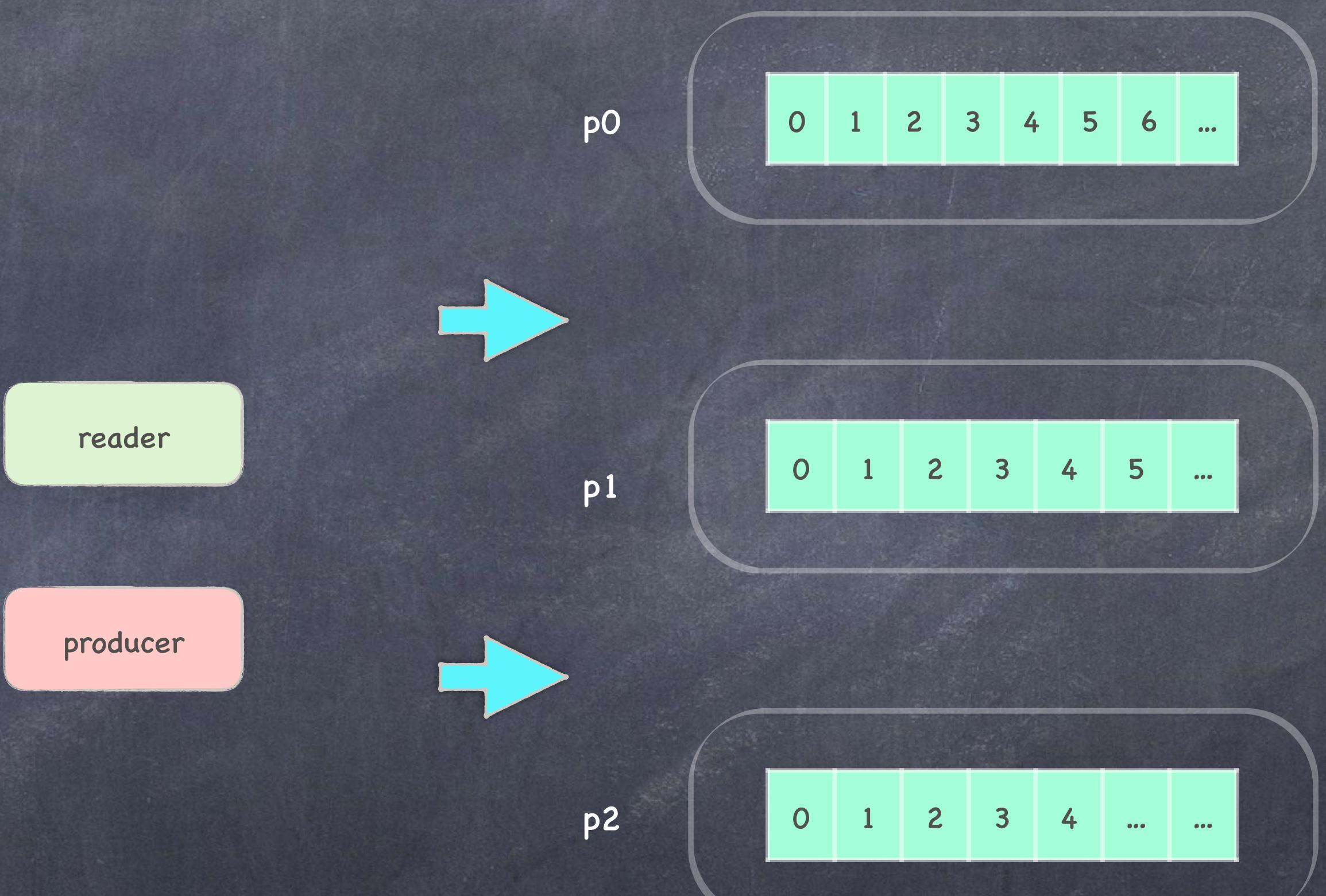
- 多副本写入不同的 bookie 上

- 采用 IO 隔离提高追尾读性能

kafka broker 分区跟 broker 绑定 !

# partition

- \* 一个 topic 下的 partition 概率上分布到多个 broker 上
- \* 提高客户端的并行度，提高效率
- \* 提高吞吐率，减轻单机负载压力
- \* 提高水平扩展性，可扩容分区数
- \* ...

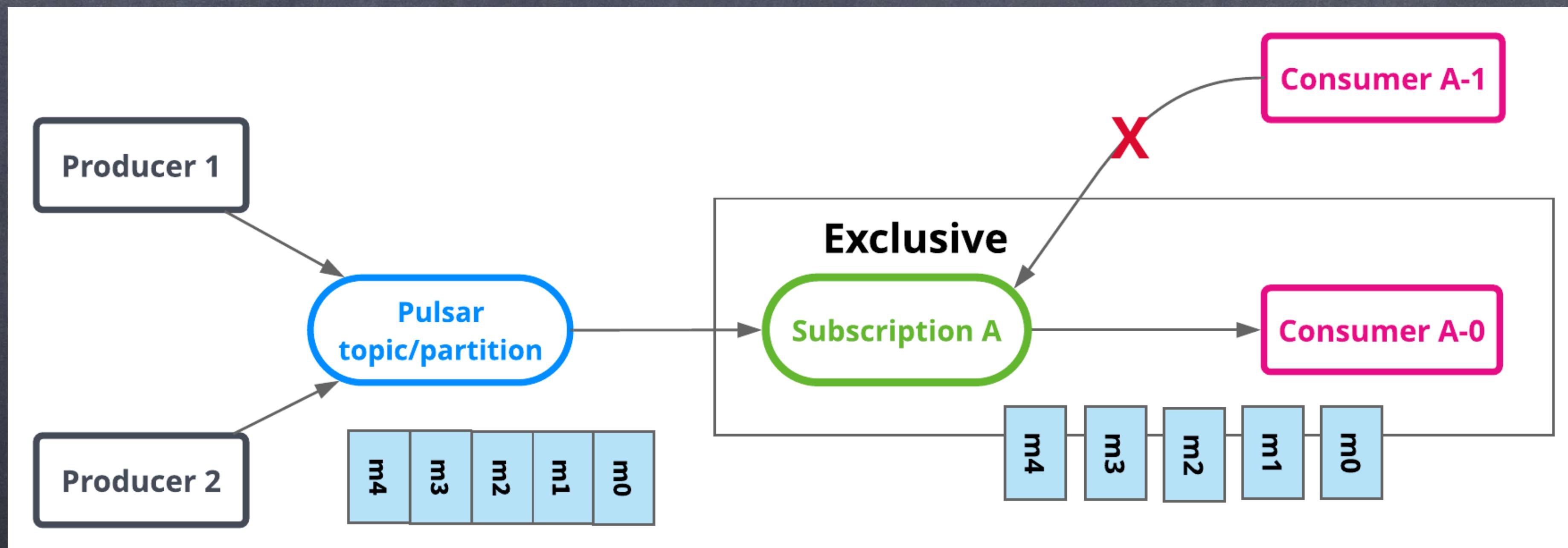


一个 partition 为一个并发维度 !!!

# subscription

## exclusive mode

只能与一个 **consumer** 关联, 只有这个 **consumer** 可以接收到消息. 其他消费者绑定会报错, 如果绑定的 **consumer** 出现故障了就会停止消费.

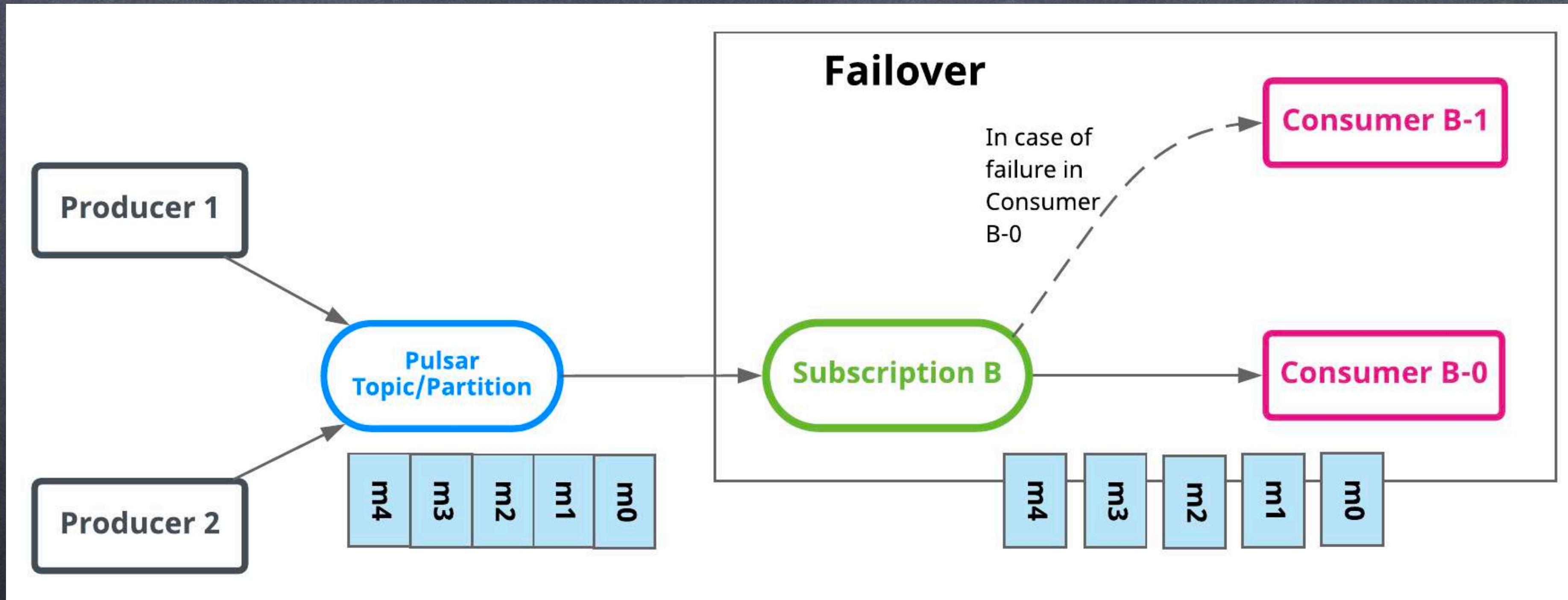


# subscription

## failover mode

当存在多个 **consumer** 时, 将会按字典顺序排序, 第一个 **consumer** 被初始化为唯一接受消息的消费者.

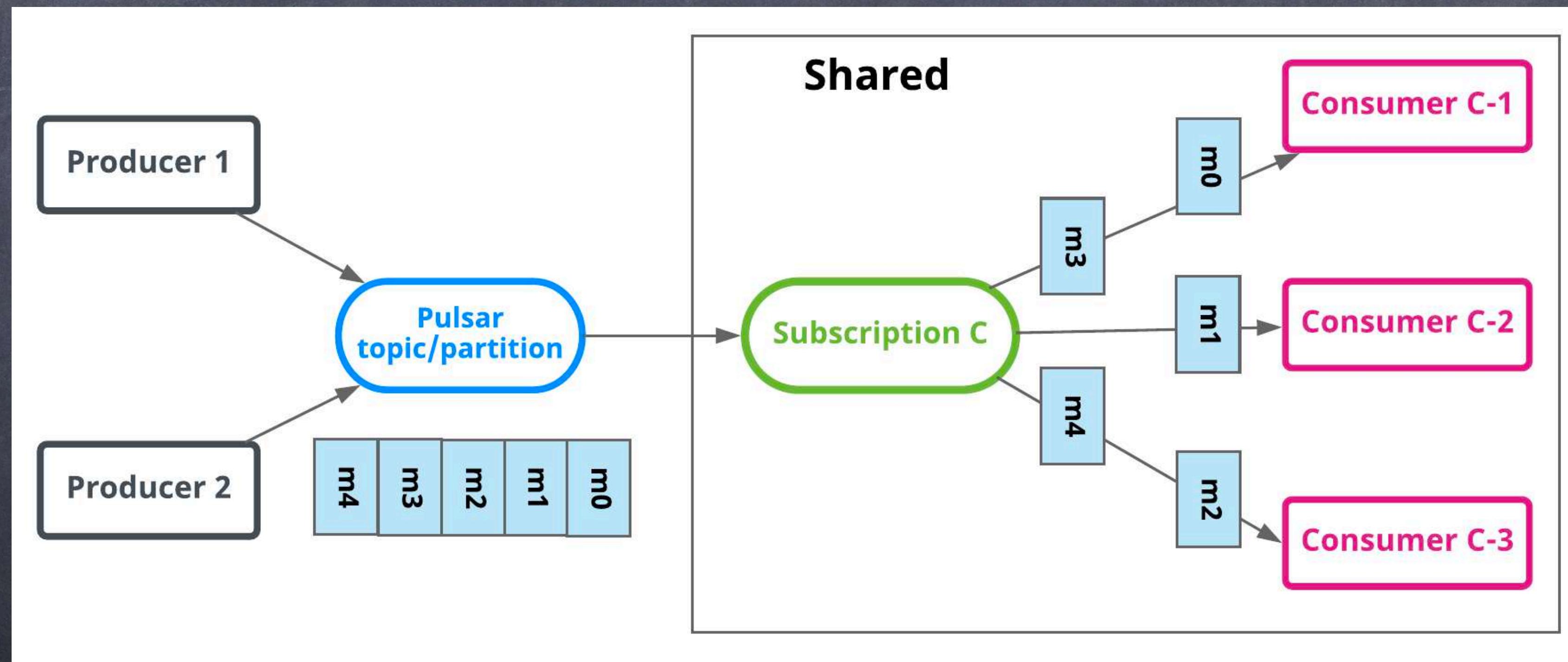
当 **consumer** 断开时, 所有的消息 (未被确认和后续进入的) 将会被分发给队列中的下一个 **consumer**.



# subscription

shared mode

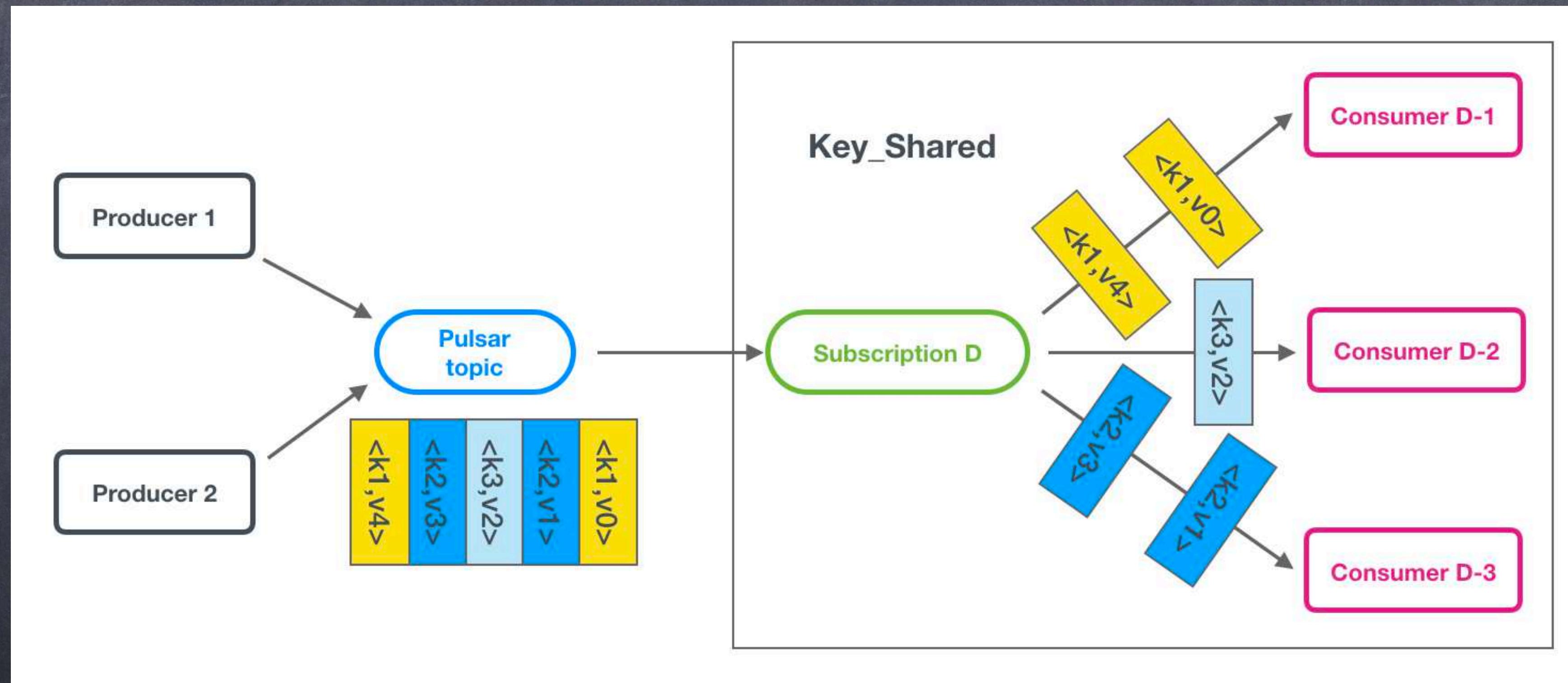
消息通过 **round robin** 轮询机制, 分发给不同的消费者,  
并且每个消息仅会被分发给一个消费者.



# subscription

## key\_shared mode

当存在多个 *Consumer* 时, 将根据消息的 *Key* 进行分发, *Key* 相同的消息只会被分发到同一个消费者.

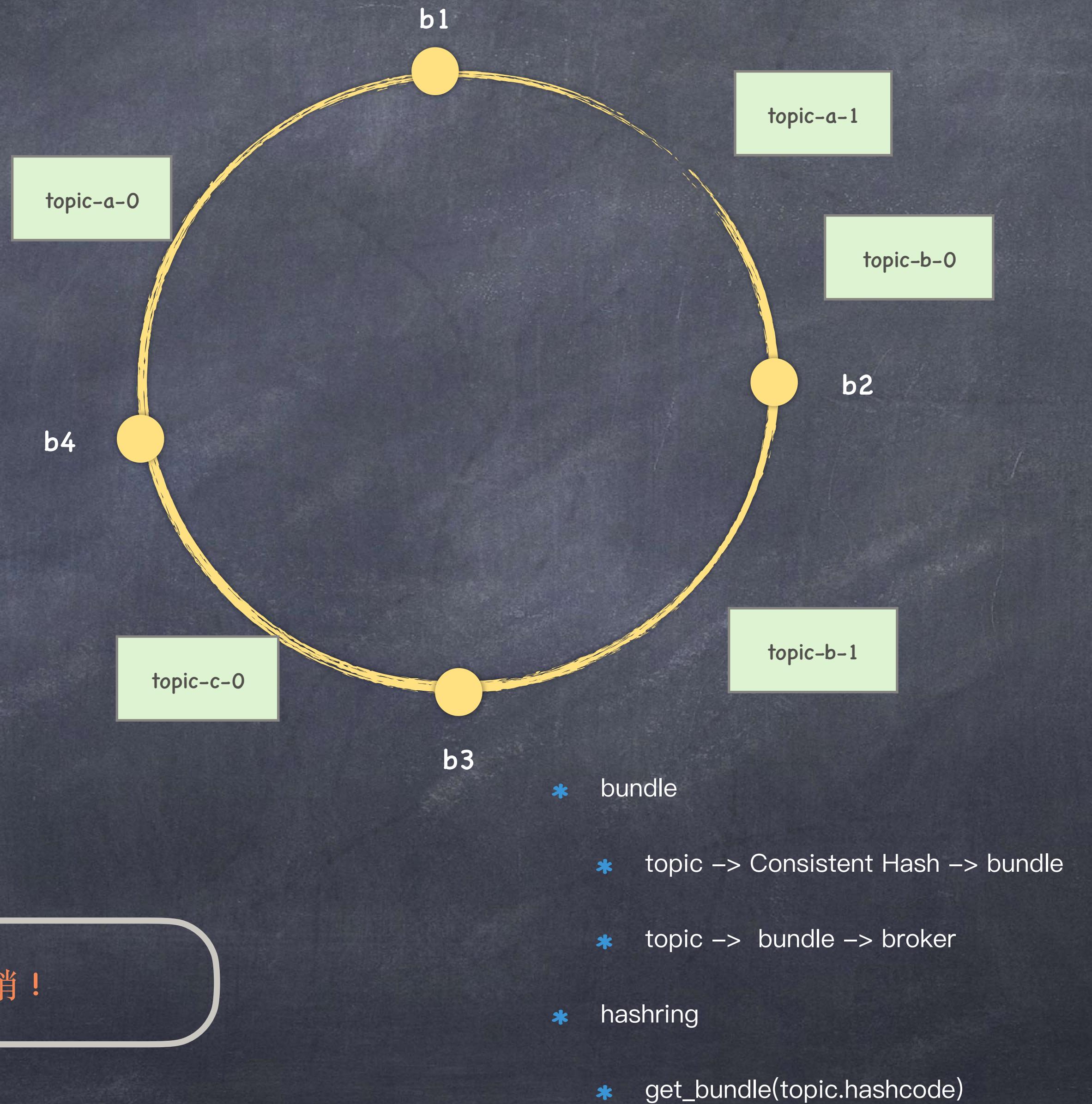
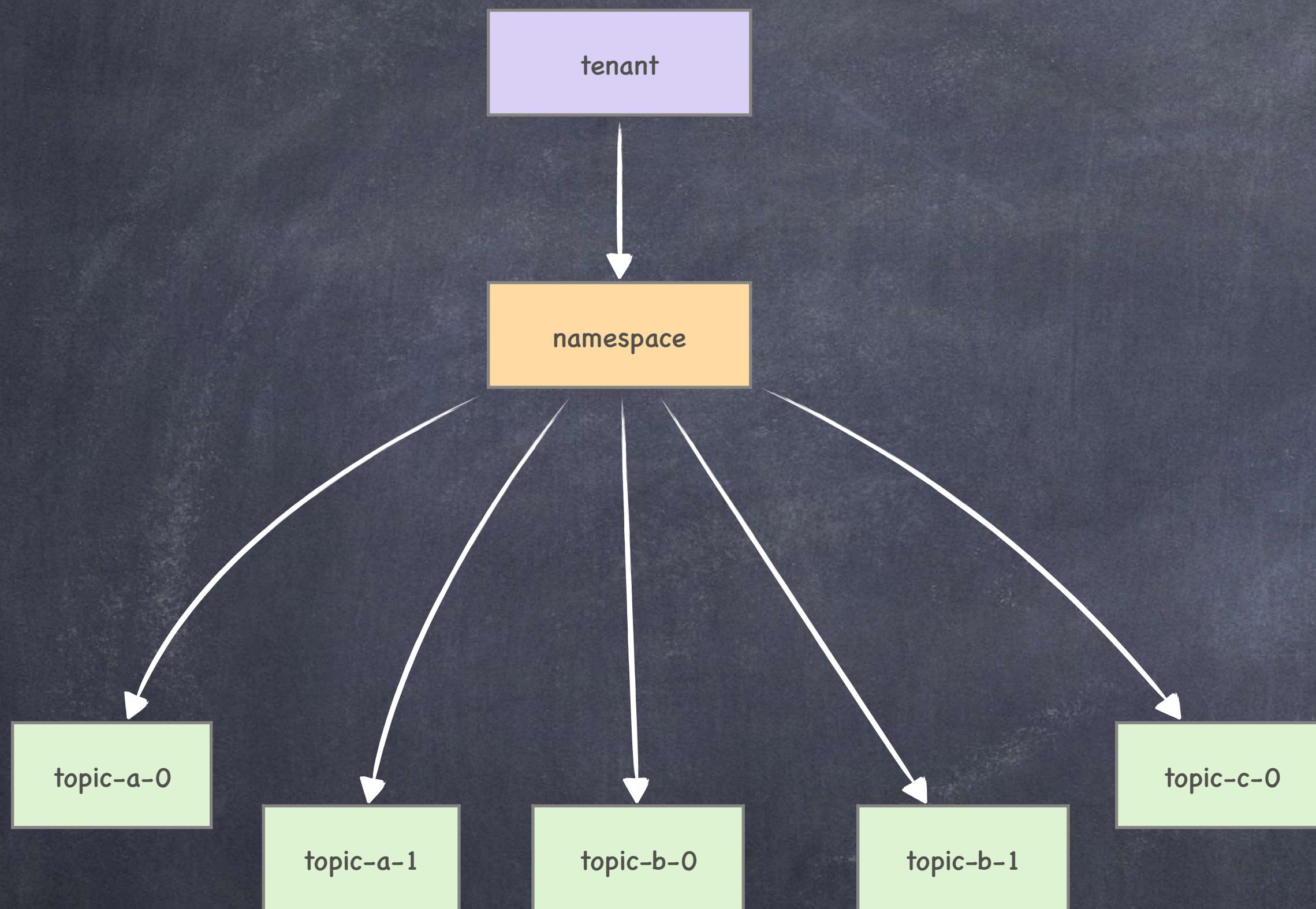




pulsar broker

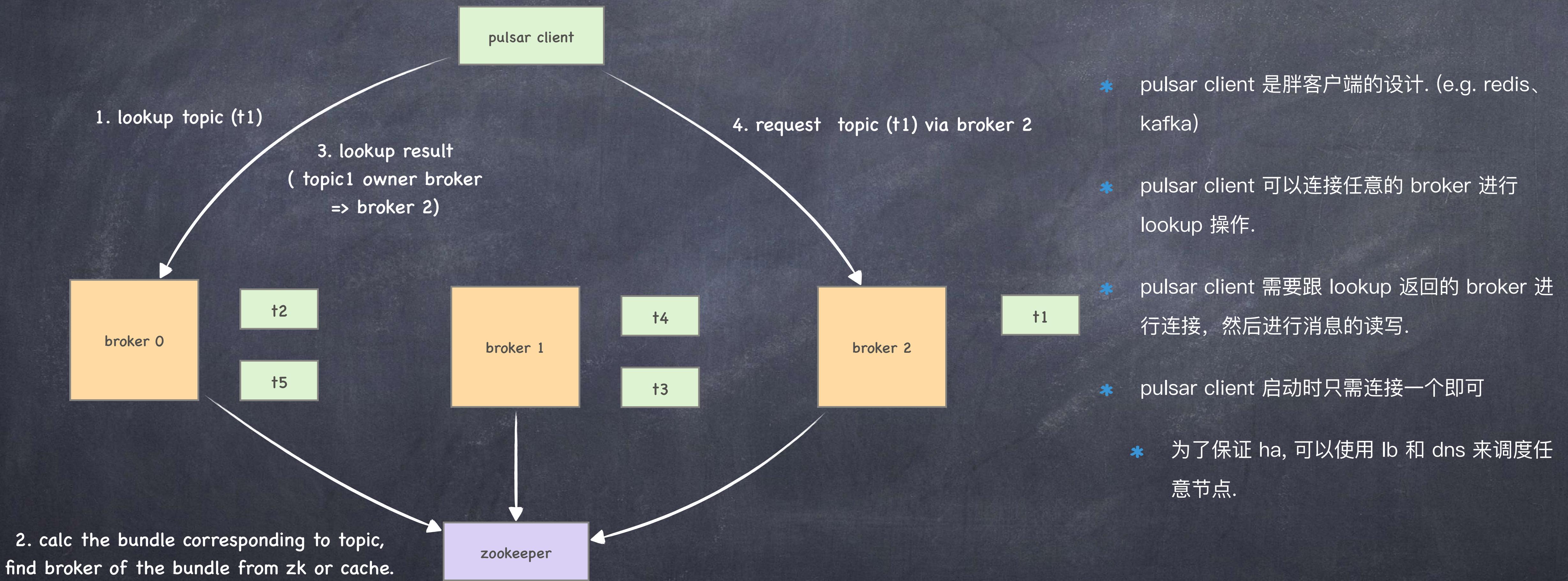


# namespace bundles



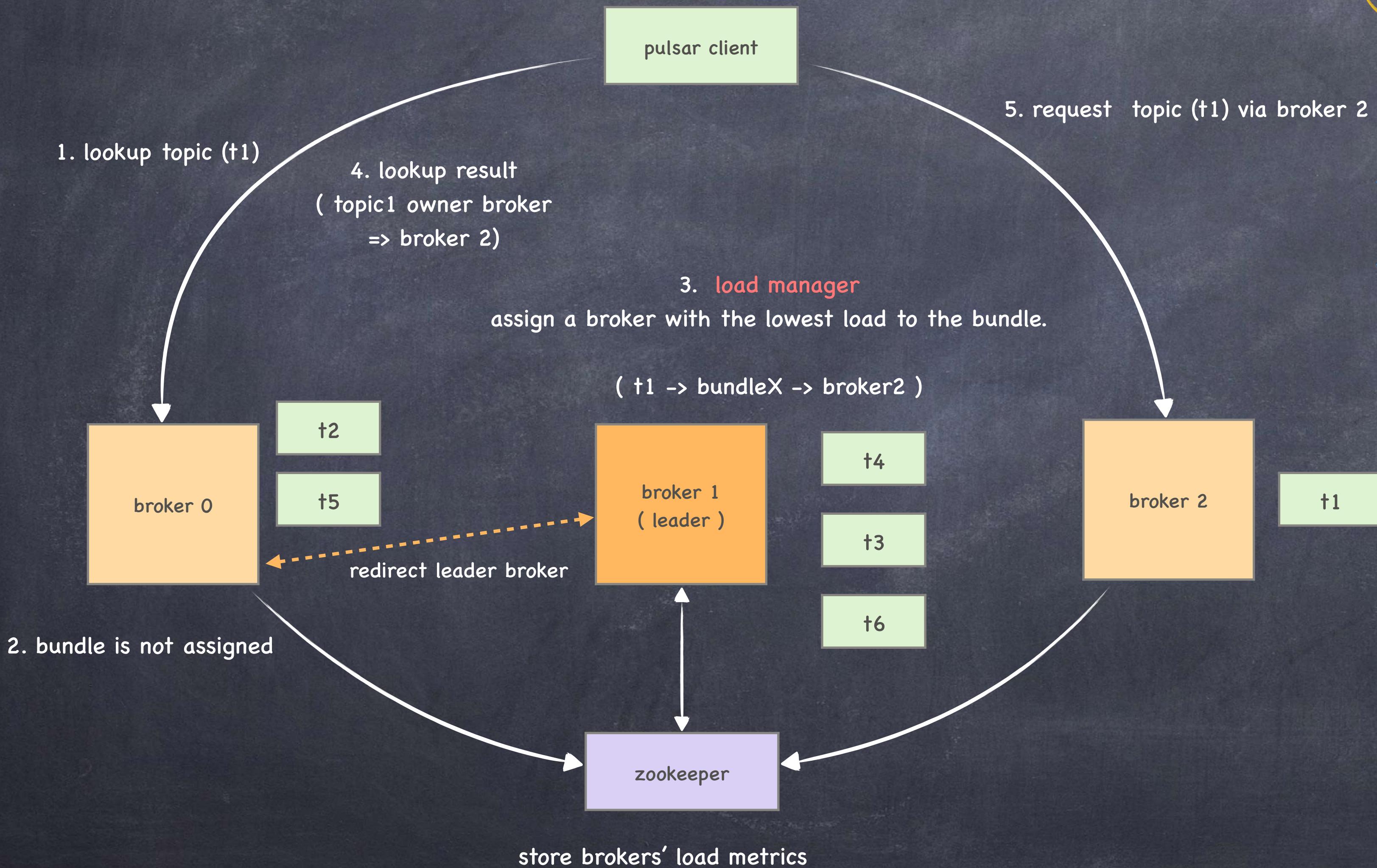
通过 **bundle** 进行负载均衡，减少对 **metadata** 的读写开销！

# topic lookup



# assign bundles to brokers

lazy assign policy



- \* **load manager** 选择低负载的 broker 来分配绑定 bundle.
- \* 分配方式
  - \* 中心化分配，统一由 leader ( load manager ) 来分配；
  - \* 简单，不易出问题
  - \* 按照统一负载视图进行分配 bundle -> broker.
- \* 非中心化分配，每个 broker 都可以分配，通过分布式锁避免重复分配.
- \* 无需转跳，当前 broker 的 load manager 即可分配
- \* 概率上存在负载的误差

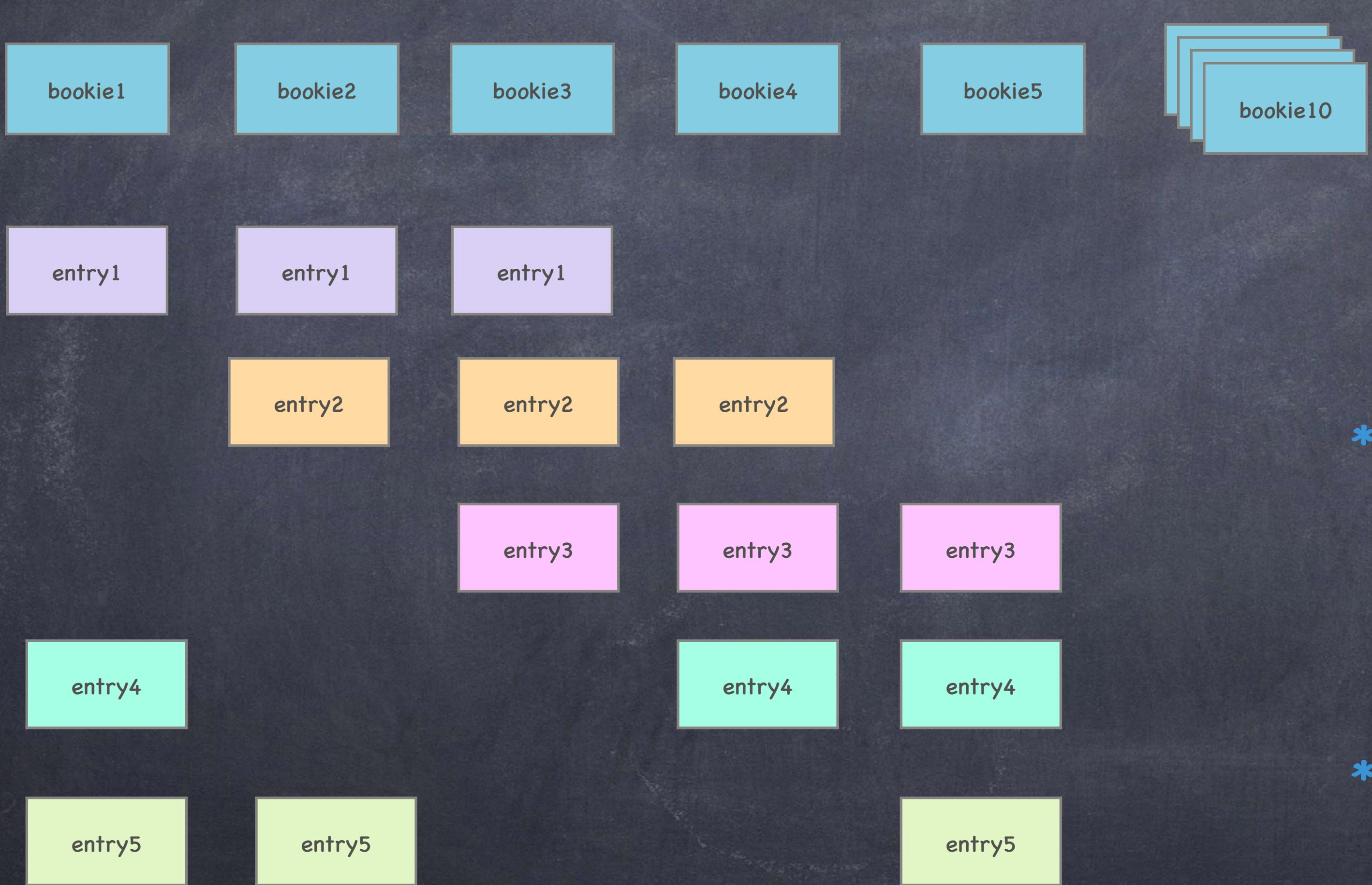
# topic layout



- \* topic
- \* partition topic
- \* ledger
- \* entry
- \* message

# bookie 节点对等架构

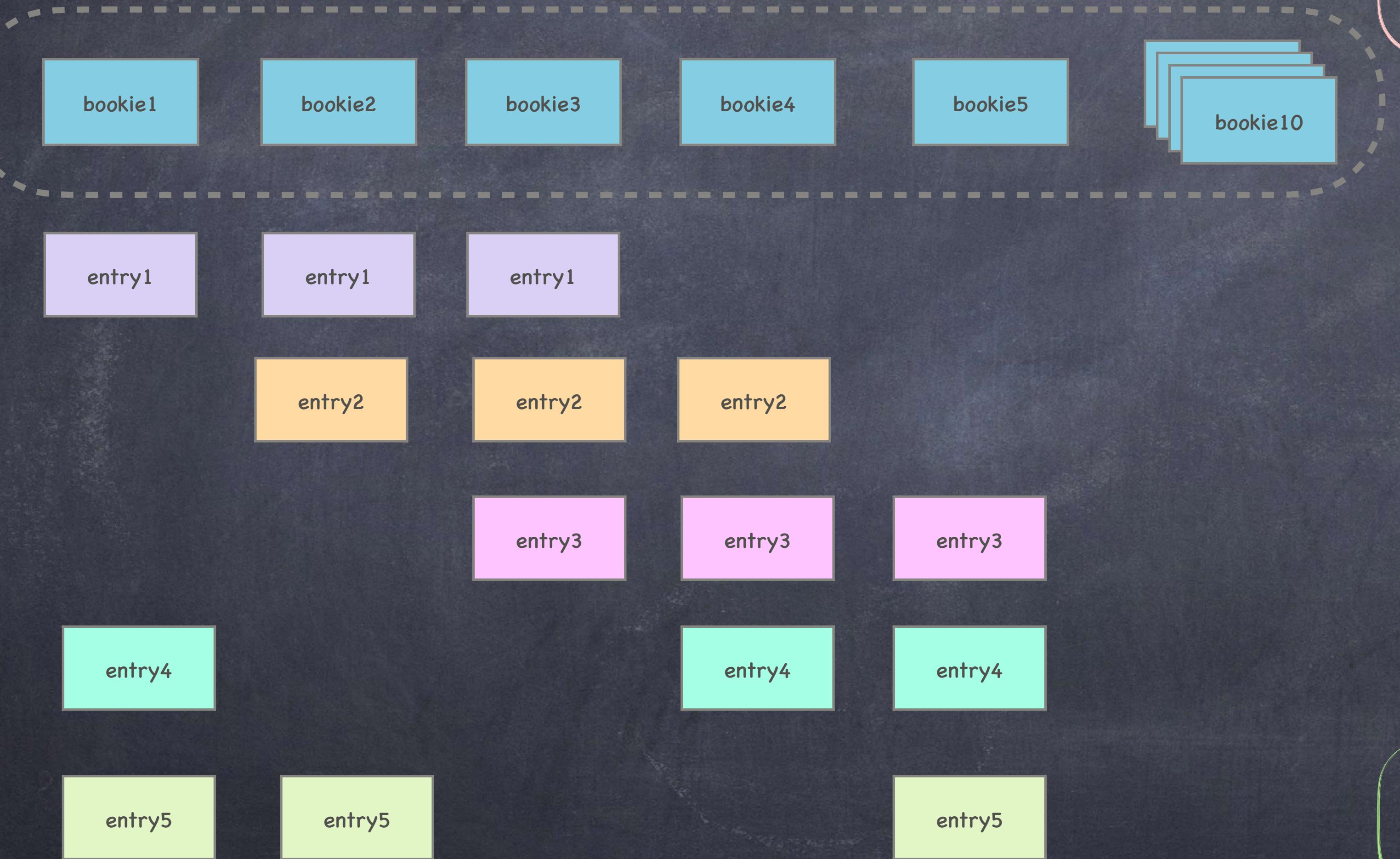
ledger quorum (  $e=5$ ,  $qw=3$ ,  $qa=2$  )



- \* openLedger (  $e$ ,  $qw$ ,  $qa$  )
- \* ensemble size 组内的节点数量
- \* write quorum 并发写入的副本数
- \* ack quorum 等待的确认数
- \* 节点对等，无主从之分
- \* 简化一致性协议
- \* 并发写入减少 leader  $\rightarrow$  follower 同步时延
- \* 多副本条带化并发写入
- \* 数据存储更加的均衡，避免倾斜

# 如何进行条带式读写

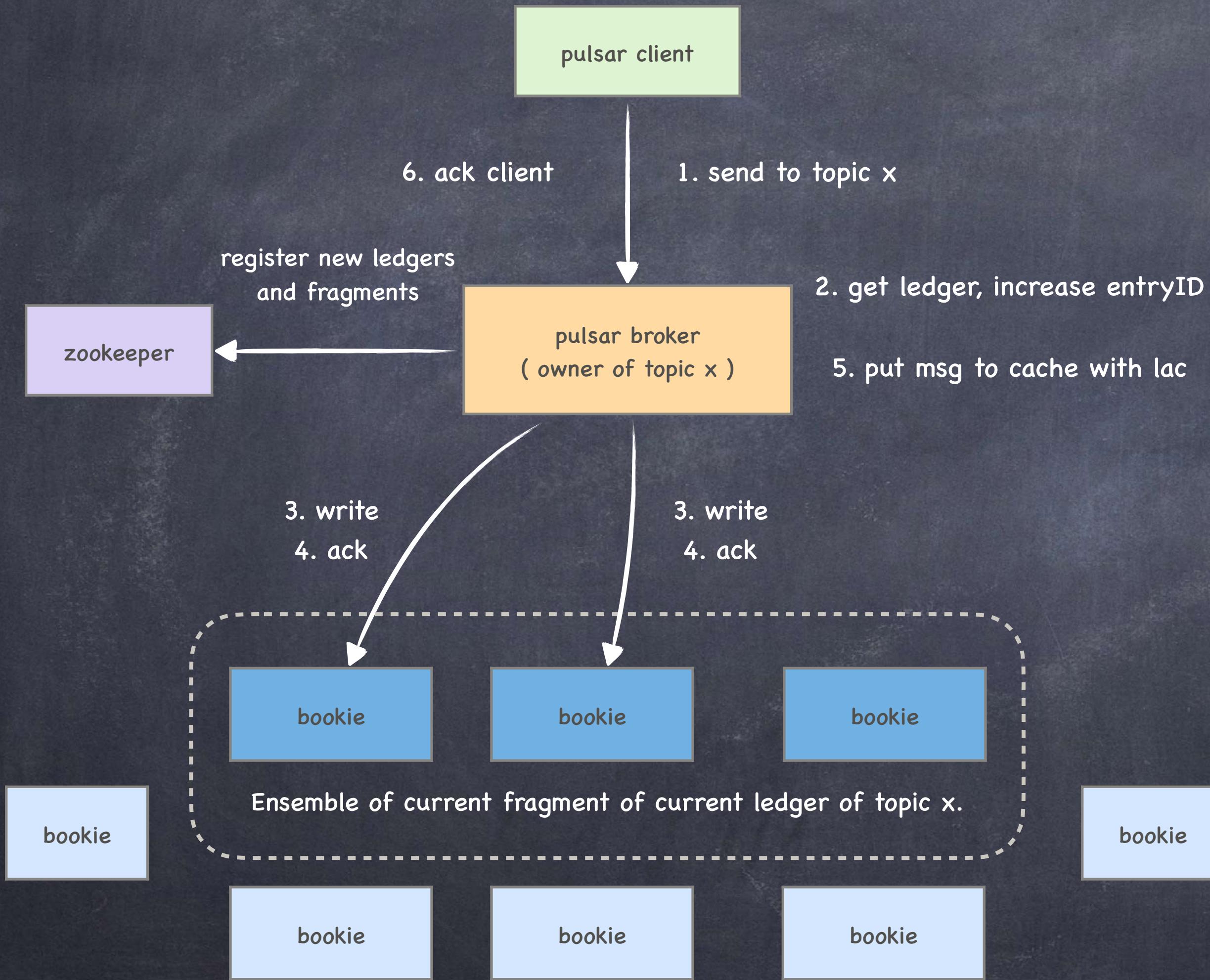
ledger quorum ( e=5, qw=3, qa=2 )



```
index = entry_id % ensemble_size  
bookie = members[index%ensemble_size]  
bookie_list = {}  
  
for i = 0; i < qw ; i++ {  
    node = members[(index + 1)%ensemble_size]  
    append(bookie_list, node)  
}  
  
return bookie_list
```

根据 entryID 计算可写入和读取的 bookie 节点

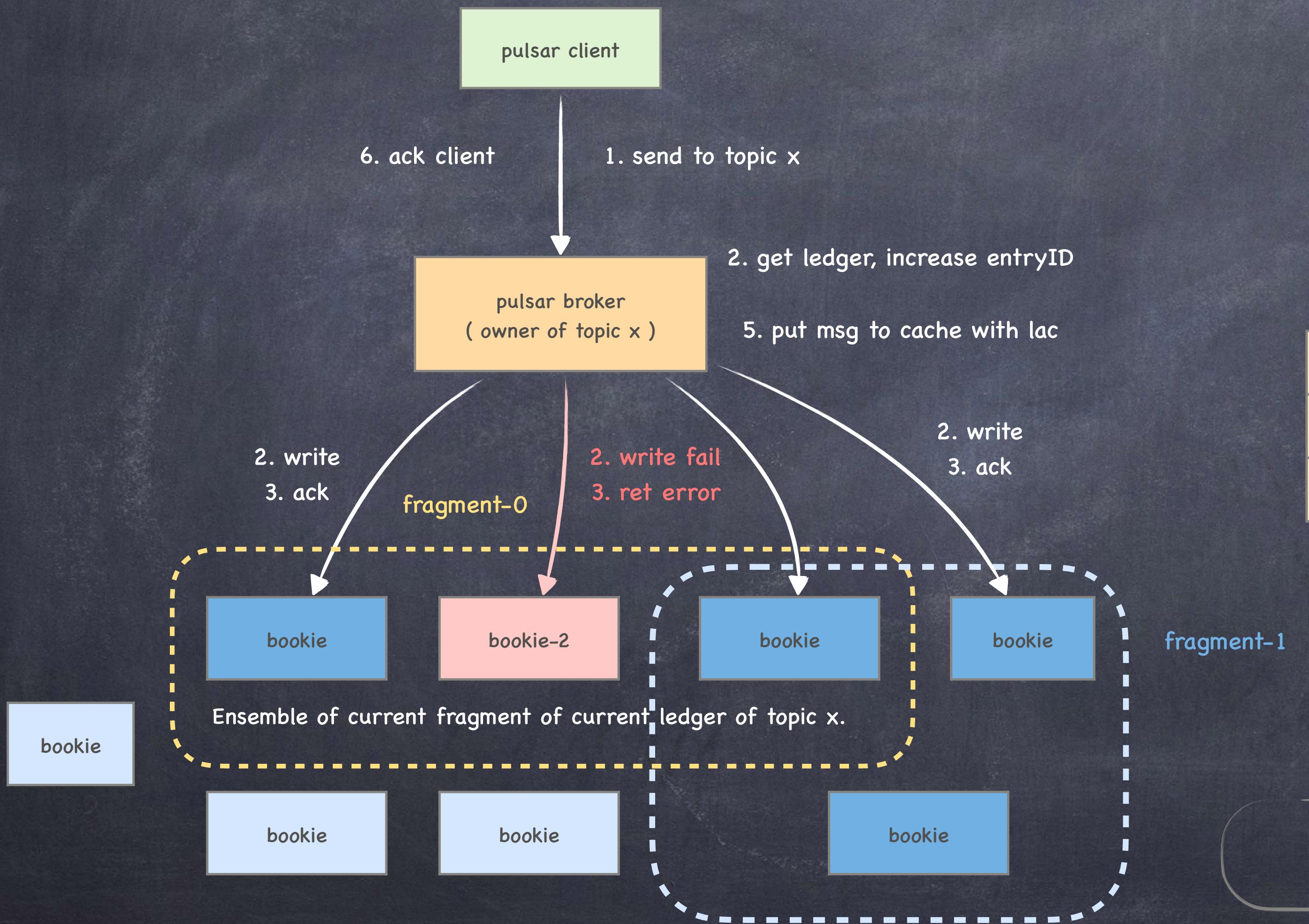
# write path dataflow



- \* 客户端发起写数据
- \* 通过任意 broker 的 lookup service 找到 topic 的 owner broker;
- \* 跳转到该 owner broker 后，进行消息的读写；
- \* 由 owner borker 使用 bookkeeper client 把消息并发写到 bookie 节点上。
- \* 当收到 ack quorum 确认后则认为写入成功
- \* 返回通知 pulsar client 写入成功

# write path dataflow

ledger metadata

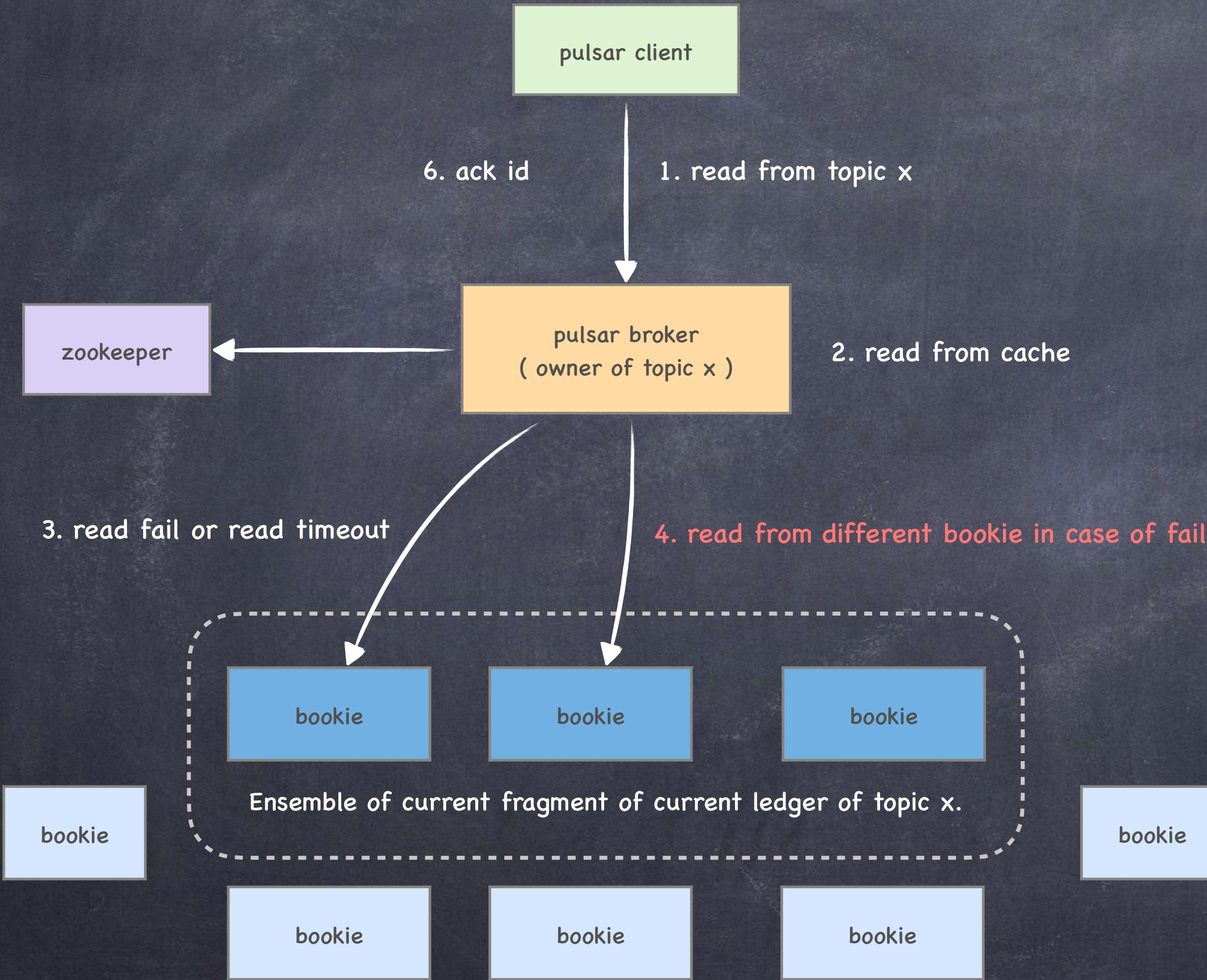


quorumSize	ensembleSize	lastEntryID	fragments	...
------------	--------------	-------------	-----------	-----

fragment	fragment	fragment
firstEntryID 0	ensemble Member b1,b2,b3	fragment0
firstEntryID 501	ensemble Member b3, b4, b5	fragment1

entryID 是严格单调递增的

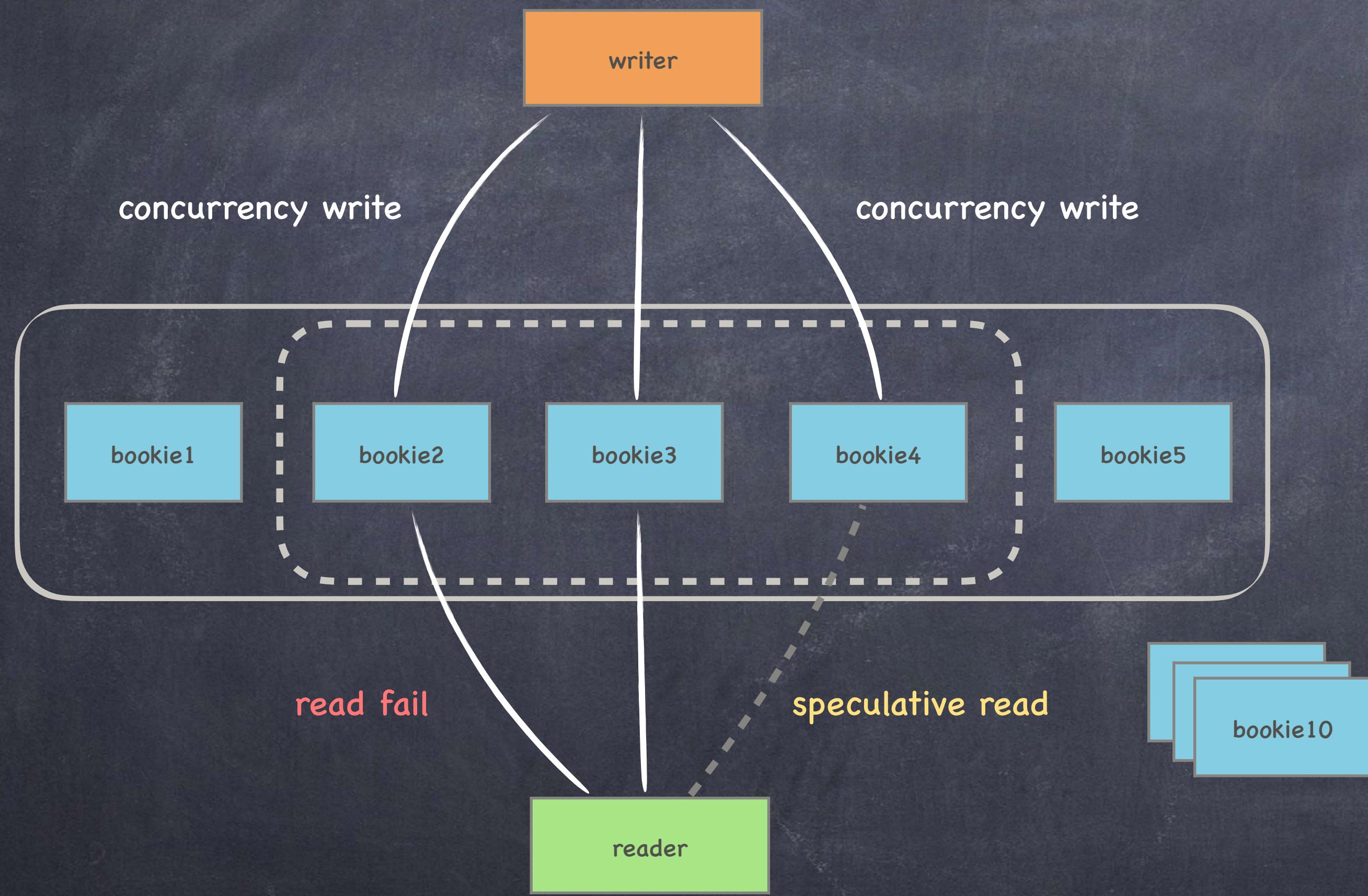
# read path dataflow



- \* 客户端发起数据读请求
- \* 从 owner broker 的缓存中查找数据
- \* 如果找不到缓存，则从 bookie 去拿数据
- \* 如果某个 bookie 异常，则从另一个 bookie 获取；
- \* 如果在一个时间范围未返回，则发起 (speculative read) 请求。

# bookie read & write ha

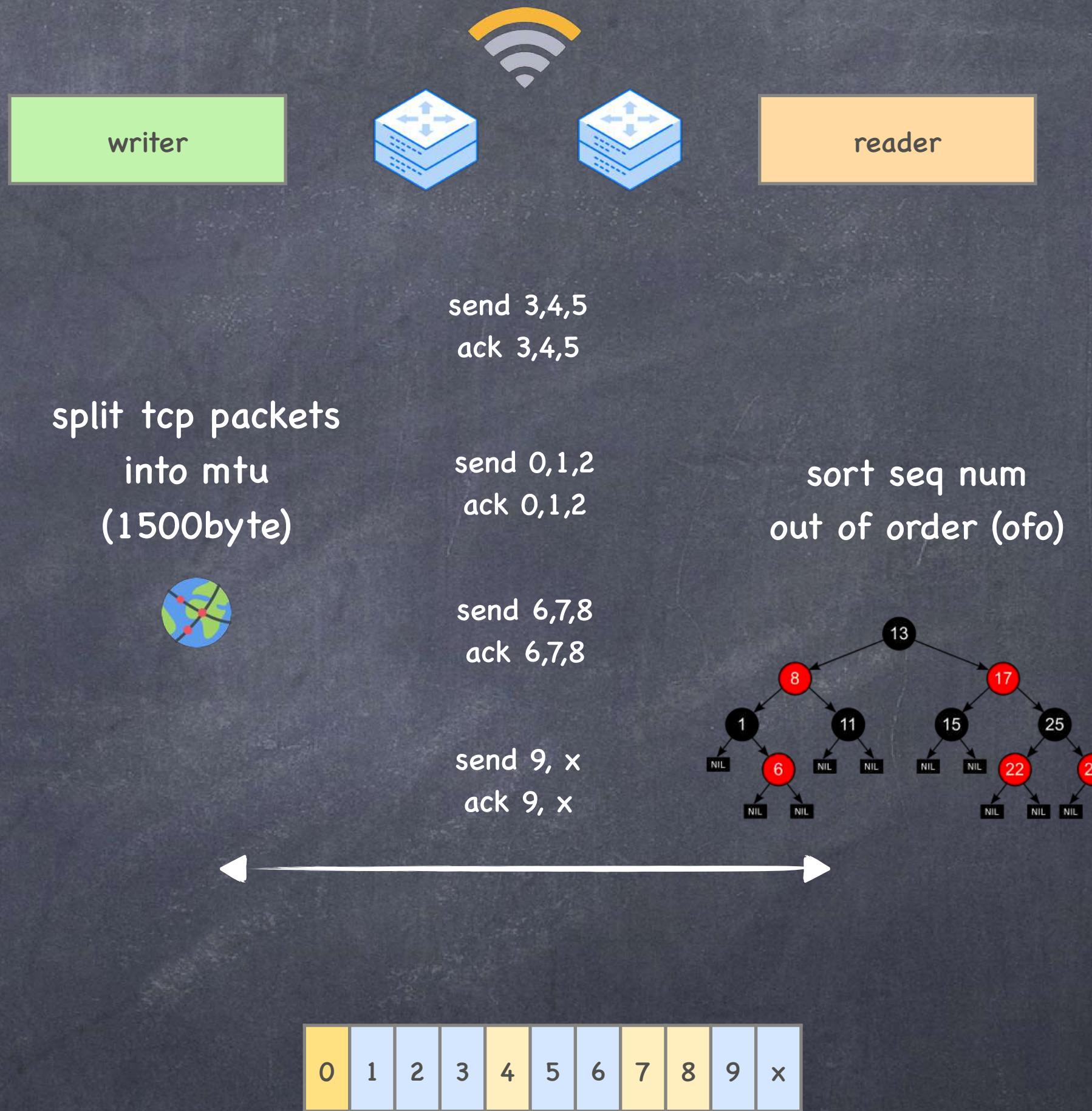
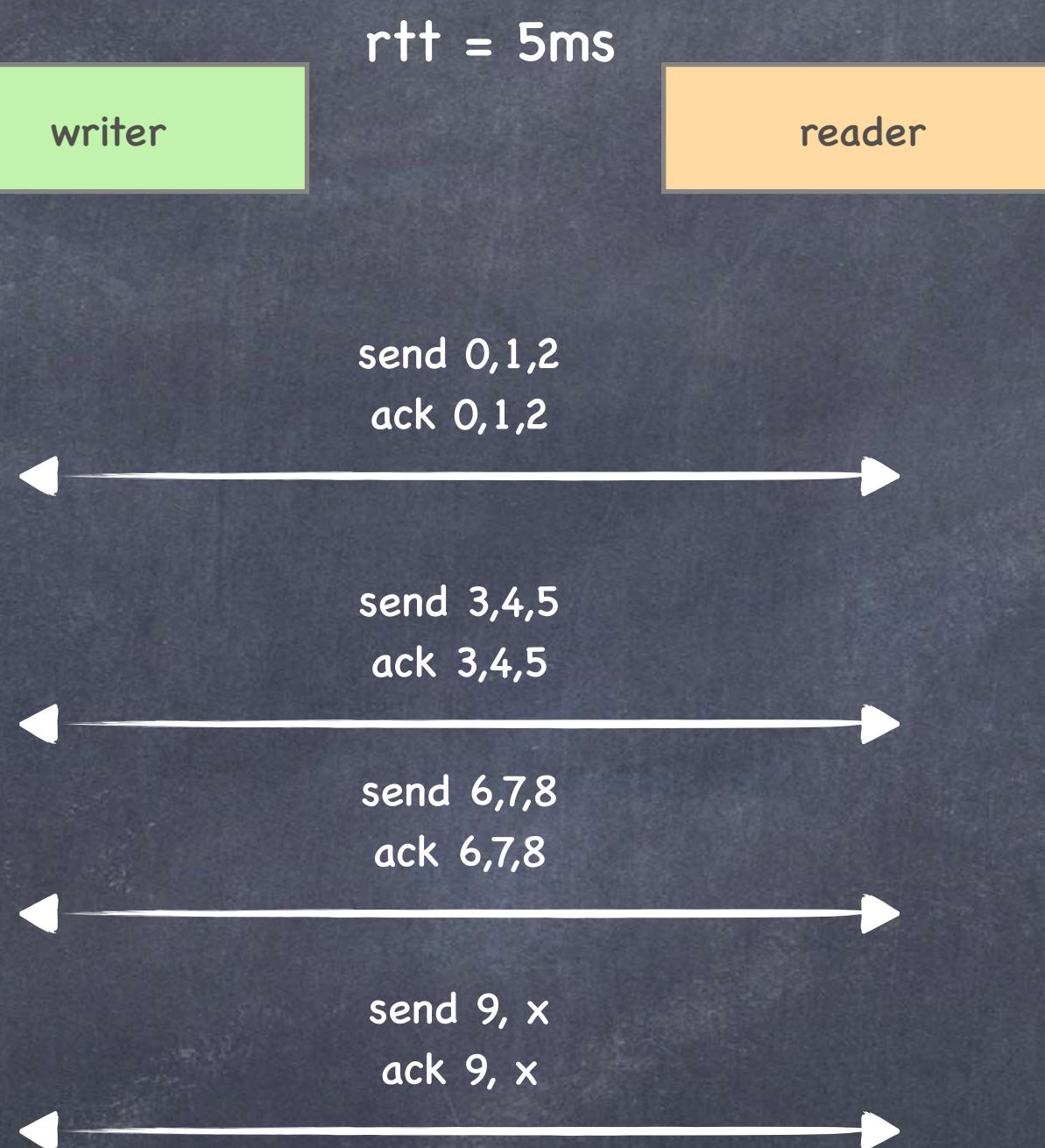
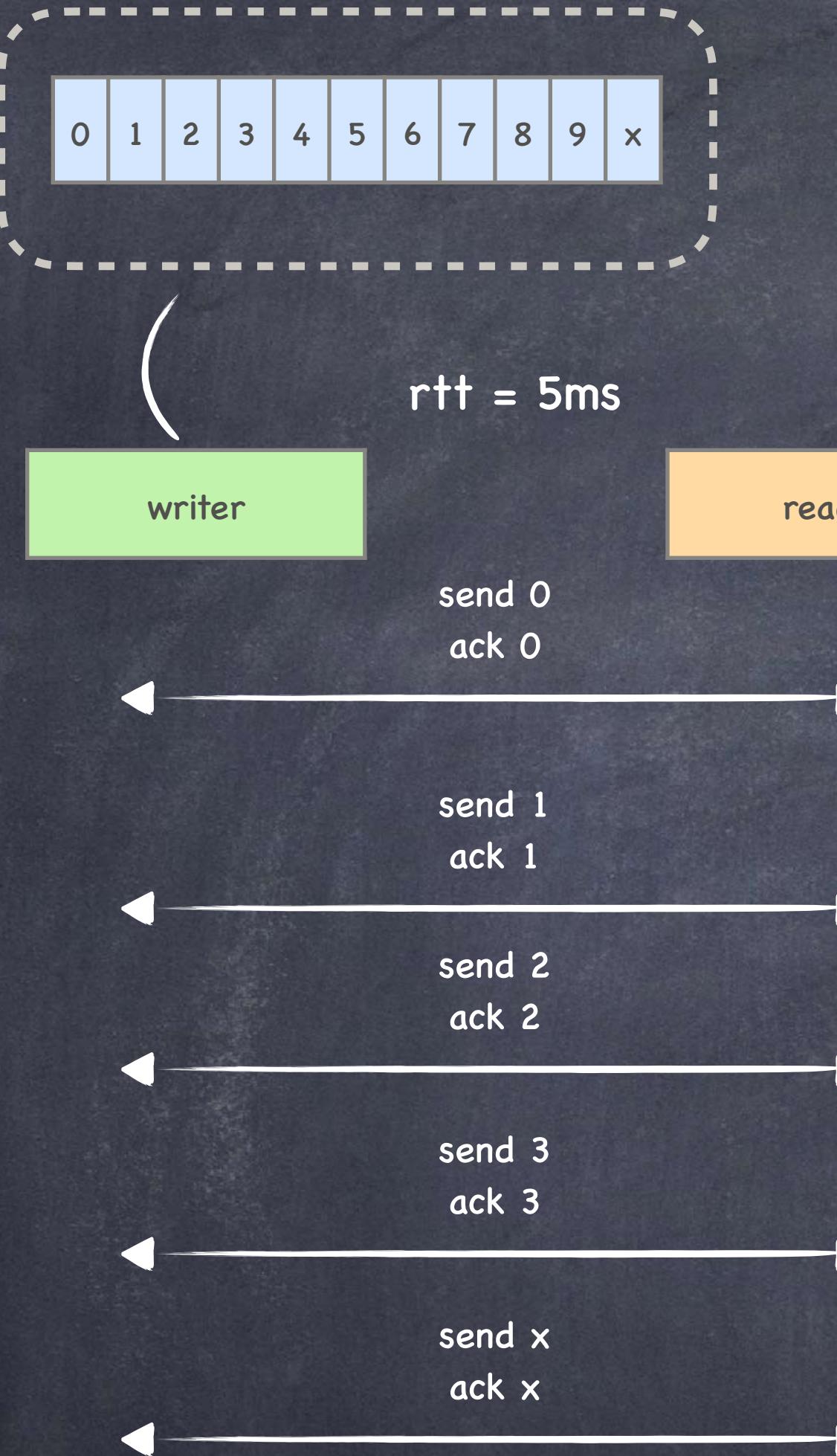
ledger quorum ( e=5, qw=3, qa=2 )



```
ledgerMetadata {  
    ensembleSize = 3,  
    writeQuorumSize = 2,  
    ackQuorumSize = 2,  
    state=CLOSED,  
    length=87679012,  
    lastEntryId=43008,  
    ensembles = {  
        0=[bookie1, bookie2, bookie3],  
        2011=[bookie2, bookie3, bookie4]  
    }  
}
```

- \* 写高可用性 – Ensemble Change
- \* 写失败不会切新的 ledger, 而是在 ledger 添加 segment members 分段
- \* 读高可用 – Speculative Read
- \* 对等副本都可以提供读
- \* 通过推测读机制 (speculative read) 来减少长尾时延

# lac & lap



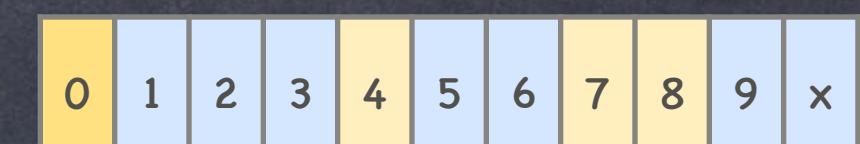
- \* 串行写入
- \* 批量串行写入
- \* 乱序写入, 顺序读取

如何高速写又顺序读 ?

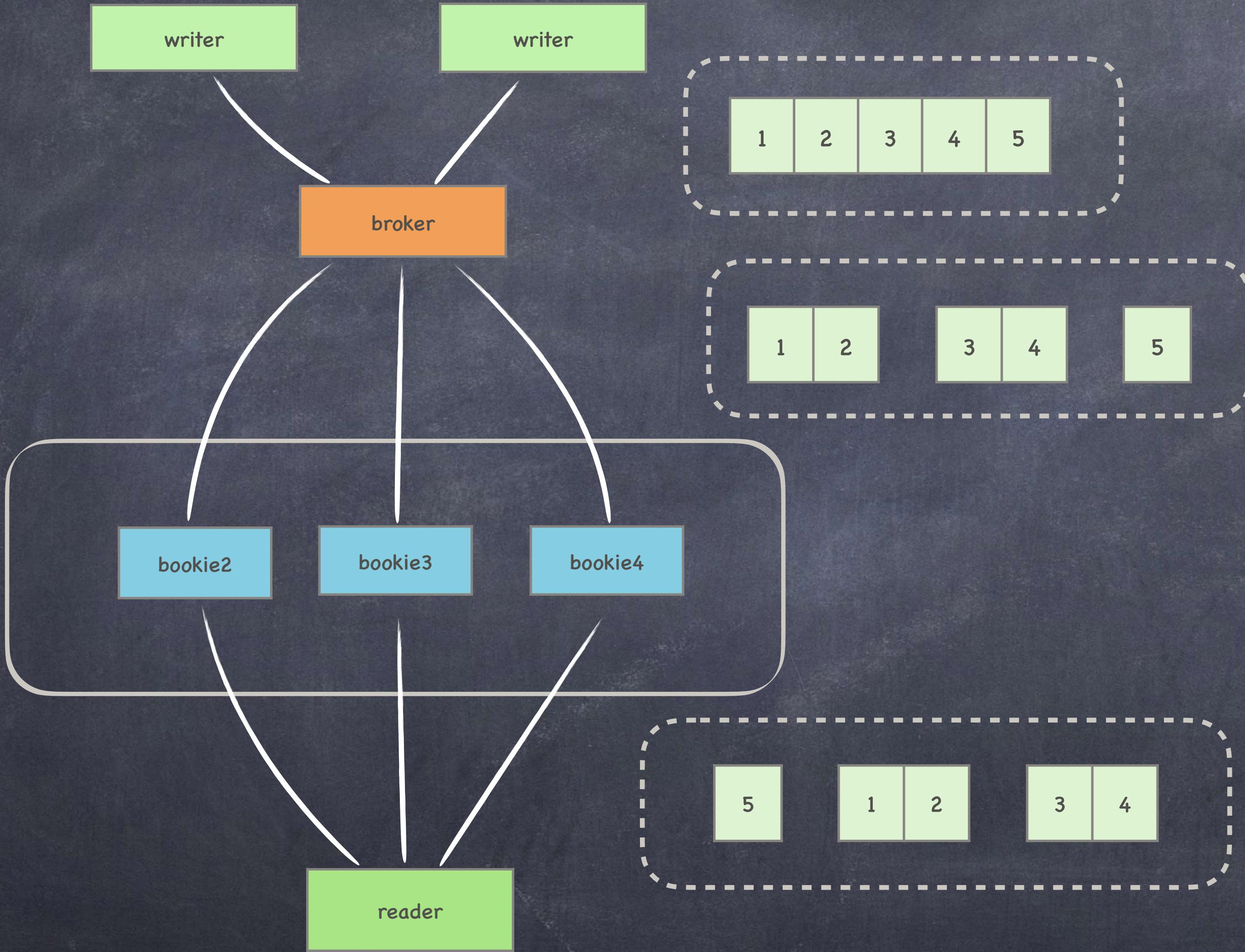
when ack 0, post 0,1,2,3 !

when ack 4, post 4,5,6 !

when ack 7 and 8, post 7,8,9,x !

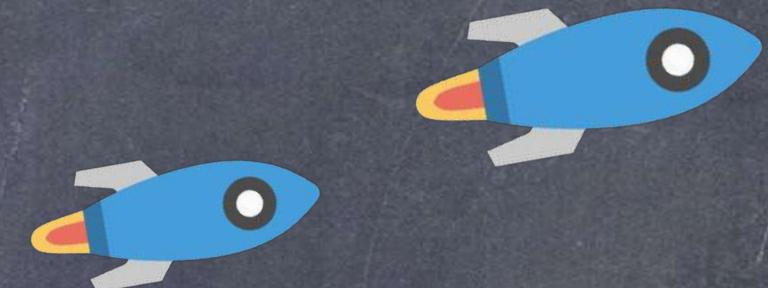


# lac & lap

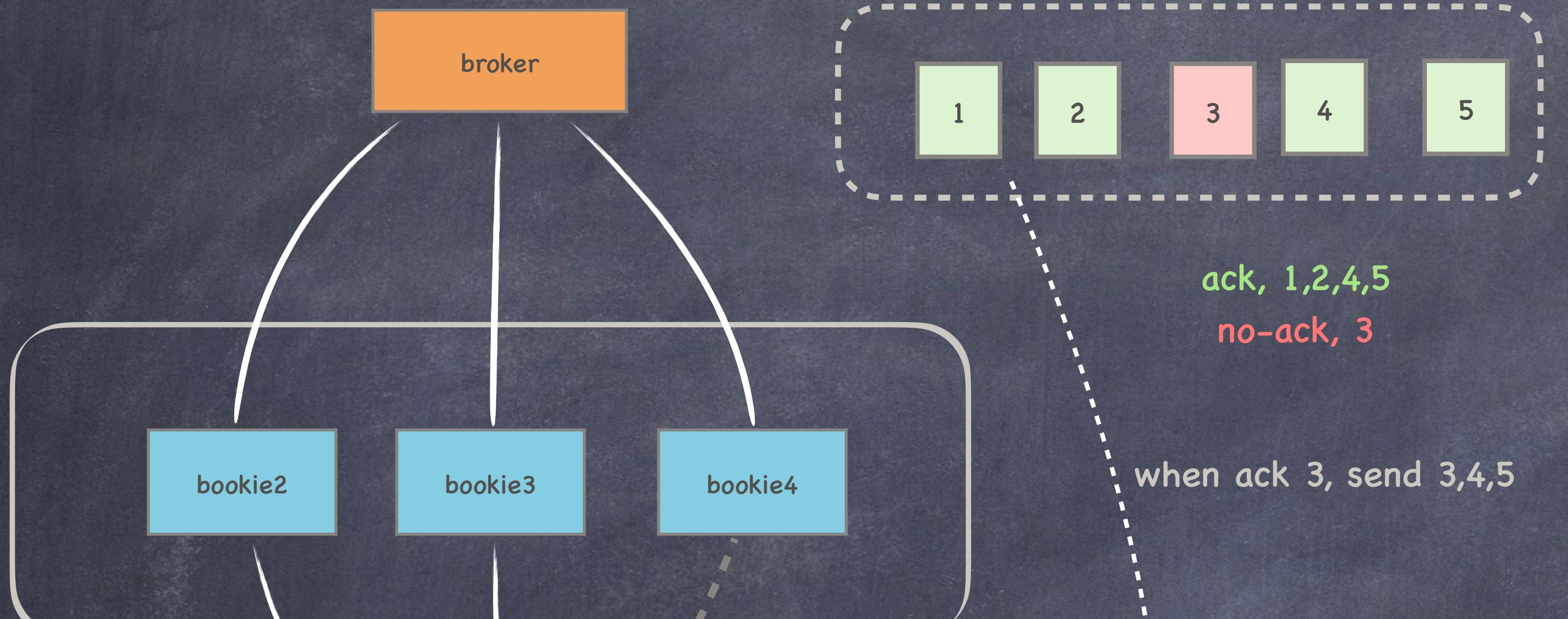


- \* broker 为了实现高吞吐, 允许并发乱序写入
- \* 但 consumer 必然无法忍受乱序读取

Pulsar  
如何实现顺序读取 ?

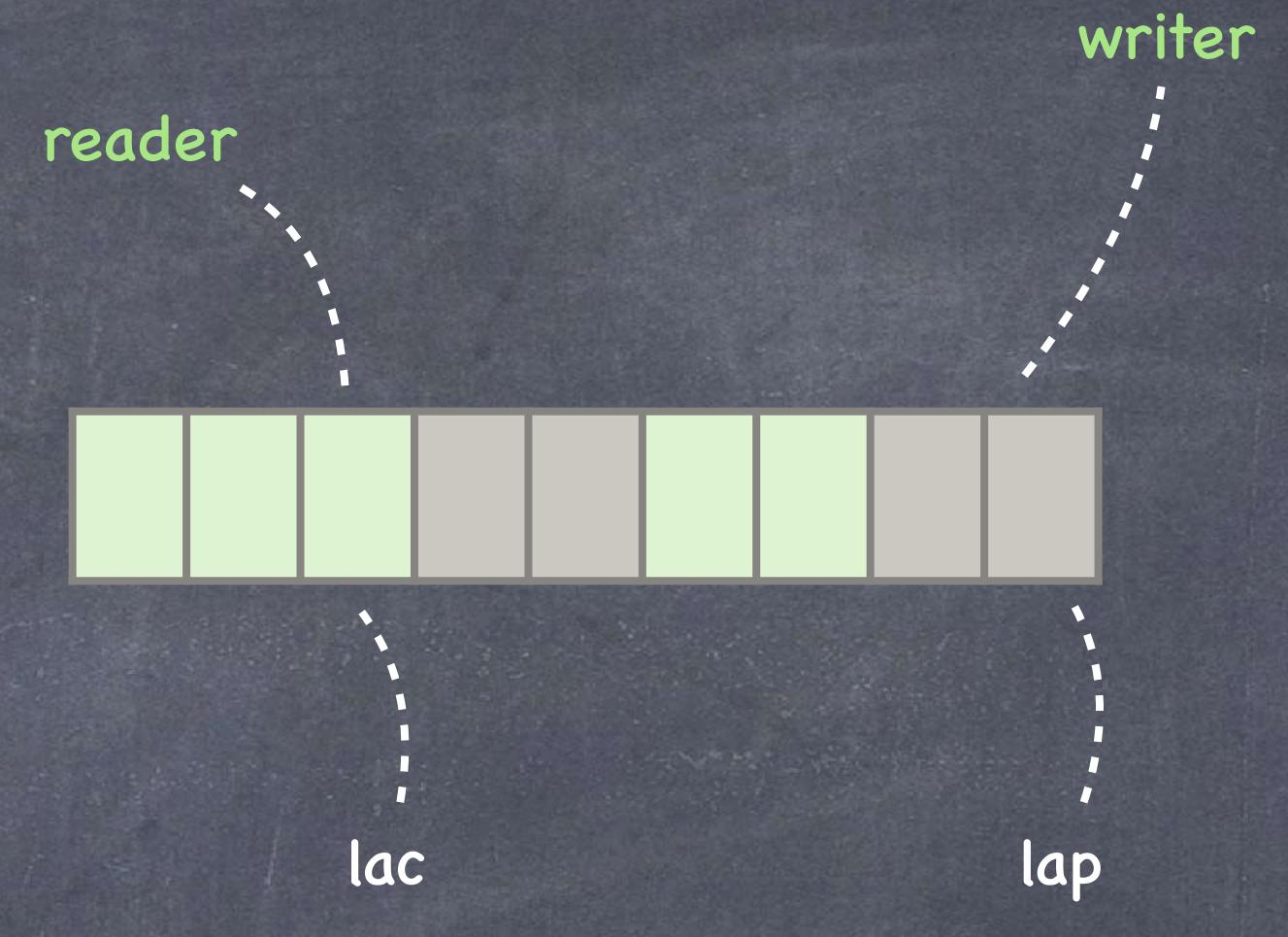


# lac & lap



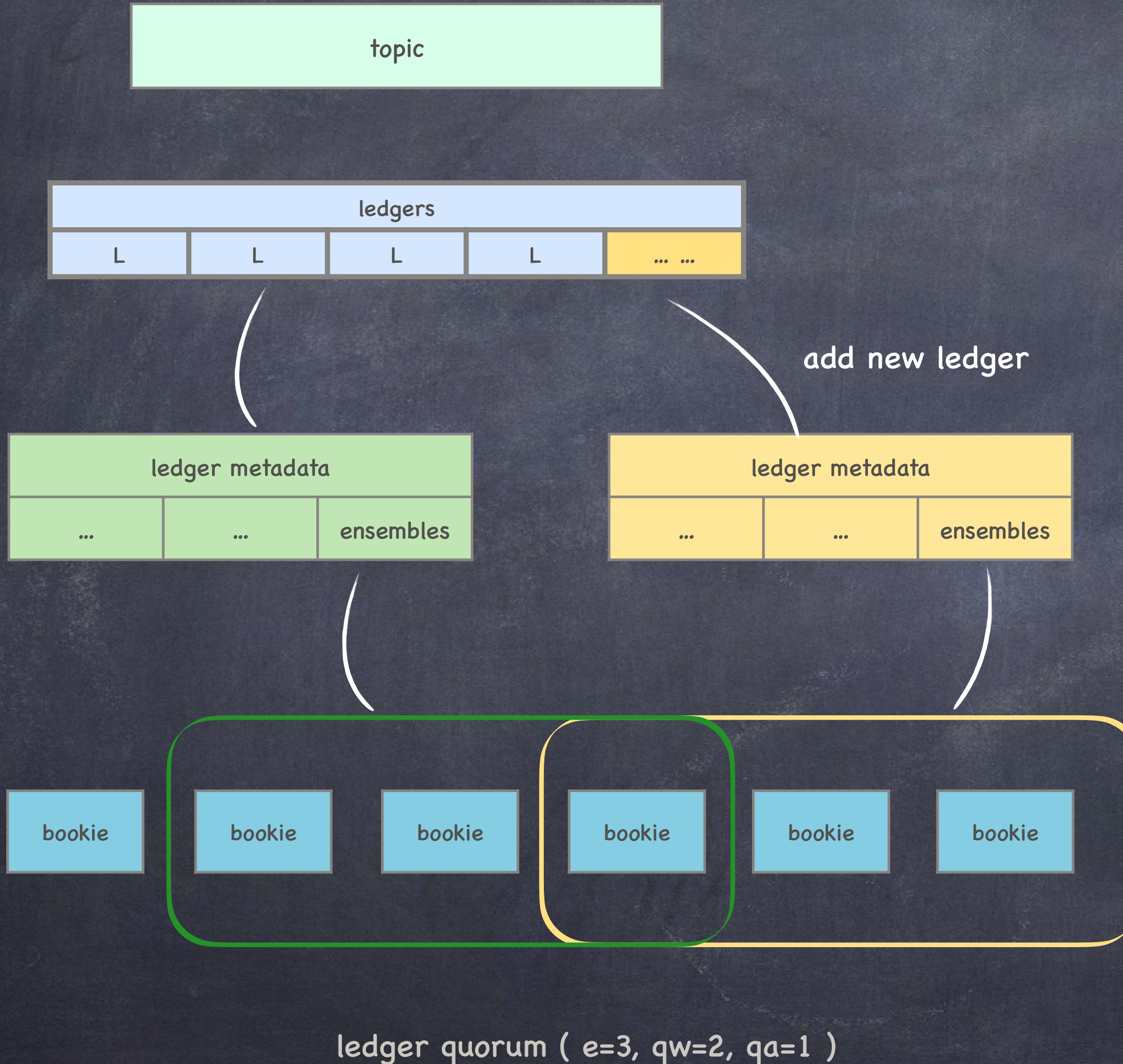
lac 是随着 entry 存到 bookie 里 !!!

Entry struct
ledgerid
entryid
lac
crc
[]data



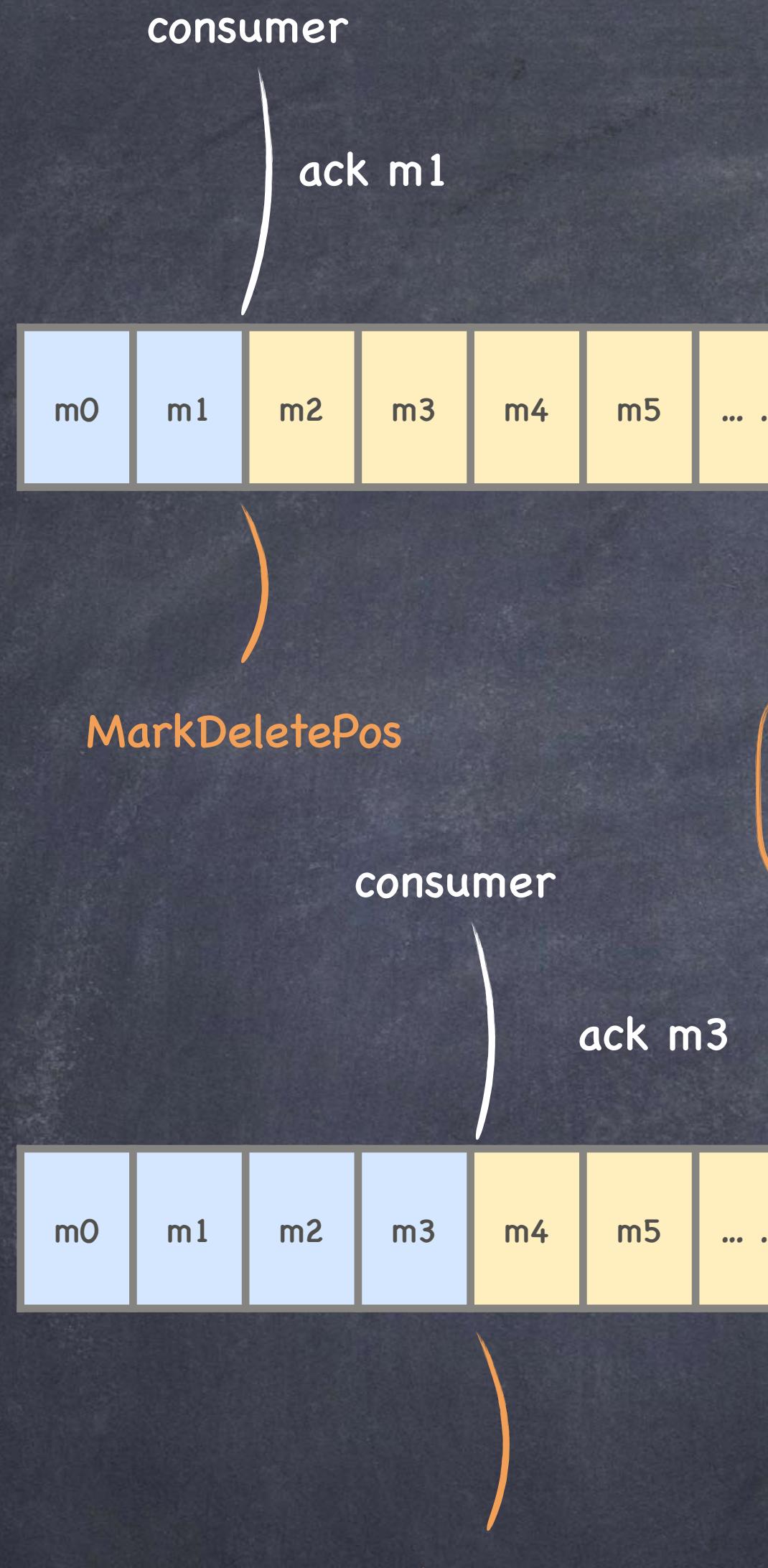
- \* lac
- \* last add confirmed
- \* 最近的连续的 message-id
- \* lac 后面的数据不可见 !!!
- \* lap
- \* last add pushed
- \* 最大的 push message-id
- \* lap – lac 为飞行中的消息量

# ledger rollover

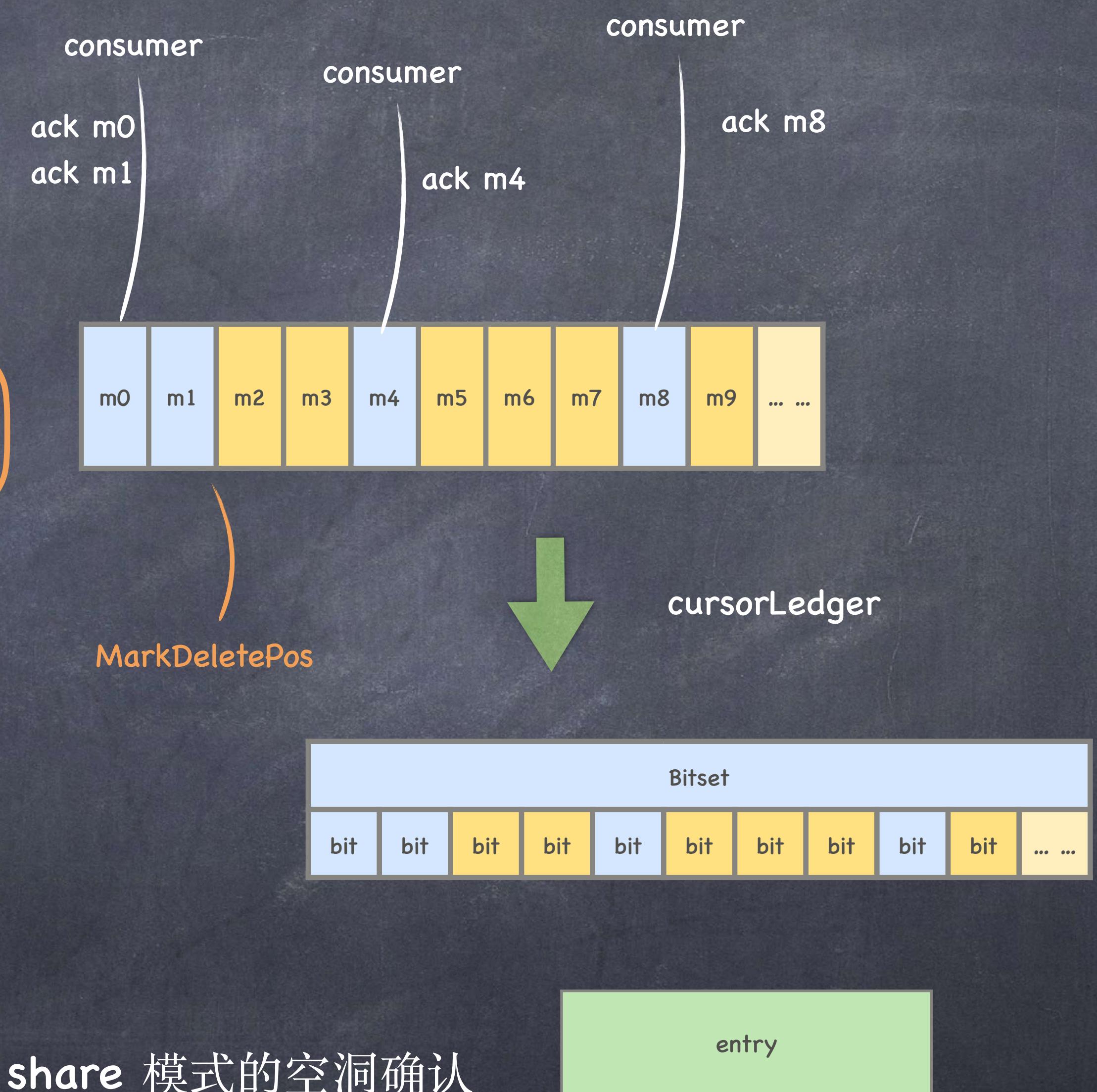


- \* 已达到 ledger 最大翻转时间
  - \* maximumRolloverTimeMs = 4hour
- \* 已达到 ledger 最大的 entry 数量
  - \* maxEntriesPerLedger = 50000
- \* 已达到 ledger 的最大容量 ( mb )
  - \* maxSizePerLedgerMb = 100

# CURSOR



当前面 entry 被确认,  
则可以移动 markDeletePos 指针





为什么没有使用 raft、paxos 等一致性协议？





# disk io view



# 机械硬盘

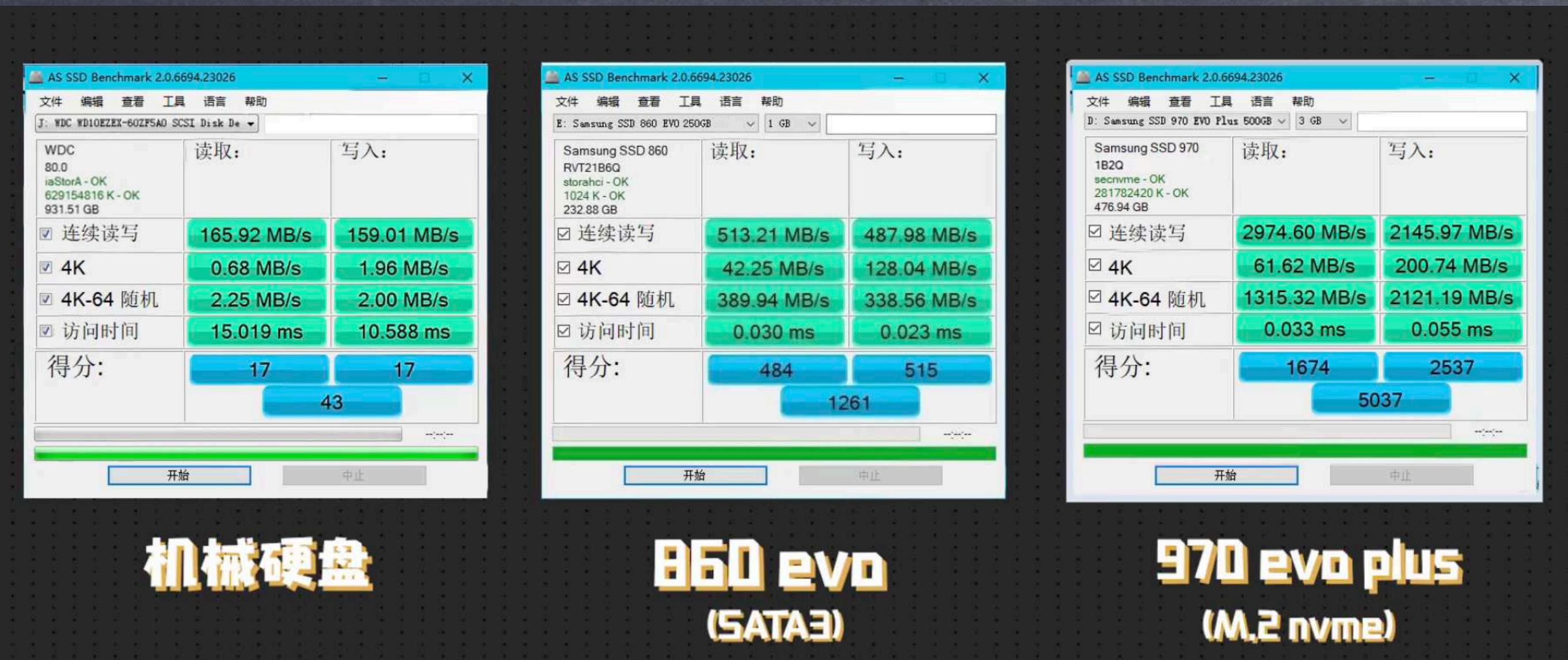


每一圈为一个磁道，每个框为一个扇区 !!!



如何访问在第五磁道的第3扇区数据 ???

- 机械硬盘通常为每分钟 7200, 10000 转
- IOPS =  $1000\text{ms} / (\text{寻道时间} + \text{旋转延迟} + \text{数据传输时间})$
- 寻道时间
  - 磁头移动到指定磁道需要的时间
  - 耗时在 3 – 15 ms (通常选定 3 ms)
- 旋转延迟
  - 磁头定位到某一磁道的扇区所需要的时间
  - $7200 \text{ rpm} = 4.17 \text{ ms}$
  - $10000 \text{ rpm} = 3 \text{ ms}$
  - 旋转简单计算 (分钟)
    - 最坏耗时 (等一圈)
$$60 \times 1000\text{ms} \div 7200 = 8.33 \text{ ms}$$
    - 平均耗时 (等半圈)
$$60 \times 1000\text{ms} \div 7200/2 = 4.17 \text{ ms}$$
- 传输时间
  - 从磁盘读出或者写入经历的时间 (通常在计算时省略)



### 寻道

- 7200转/分平均物理寻道时间是10.5ms

- 10000转/分平均物理寻道时间是7ms

- 15000转/分平均物理寻道时间是5ms

### 旋转时延

- $60 * 1000 / 7200 / 2 = 4.17\text{ms}$

- $60 * 1000 / 10000 / 2 = 3\text{ms}$

- $60 * 1000 / 15000 / 2 = 2\text{ms}$

### IOPS

- 7200 rpm = 140

- 10000 rpm = 167

- 150000 rpm = 200



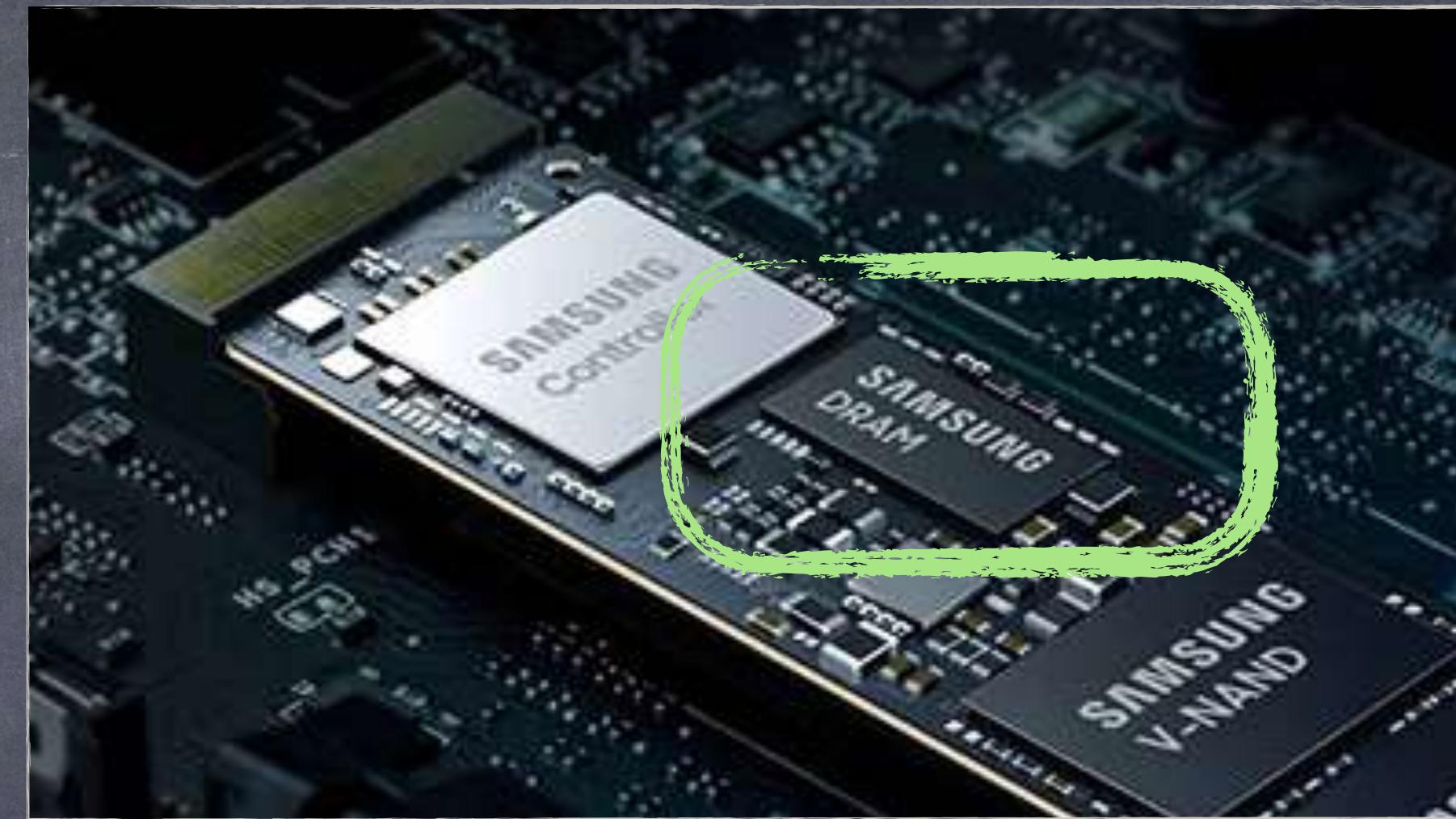
大多数硬盘有缓存机制 !!!

# ssd



消费级 SSD

- iops
  - 随机读 150w + iops
  - 随机写 150w + iops
- bd
  - 顺序读 7000 mb/s
  - 顺序写 6500 mb/s



为啥消费级 ssd 这么猛 ?



消费级 ssd vs 企业级 ssd

企业级 SSD

- iops
  - 顺序读 45w iops
  - 顺序写 40w iops
- bd
  - 顺序读 2900 mb/s
  - 顺序写 1400 mb/s

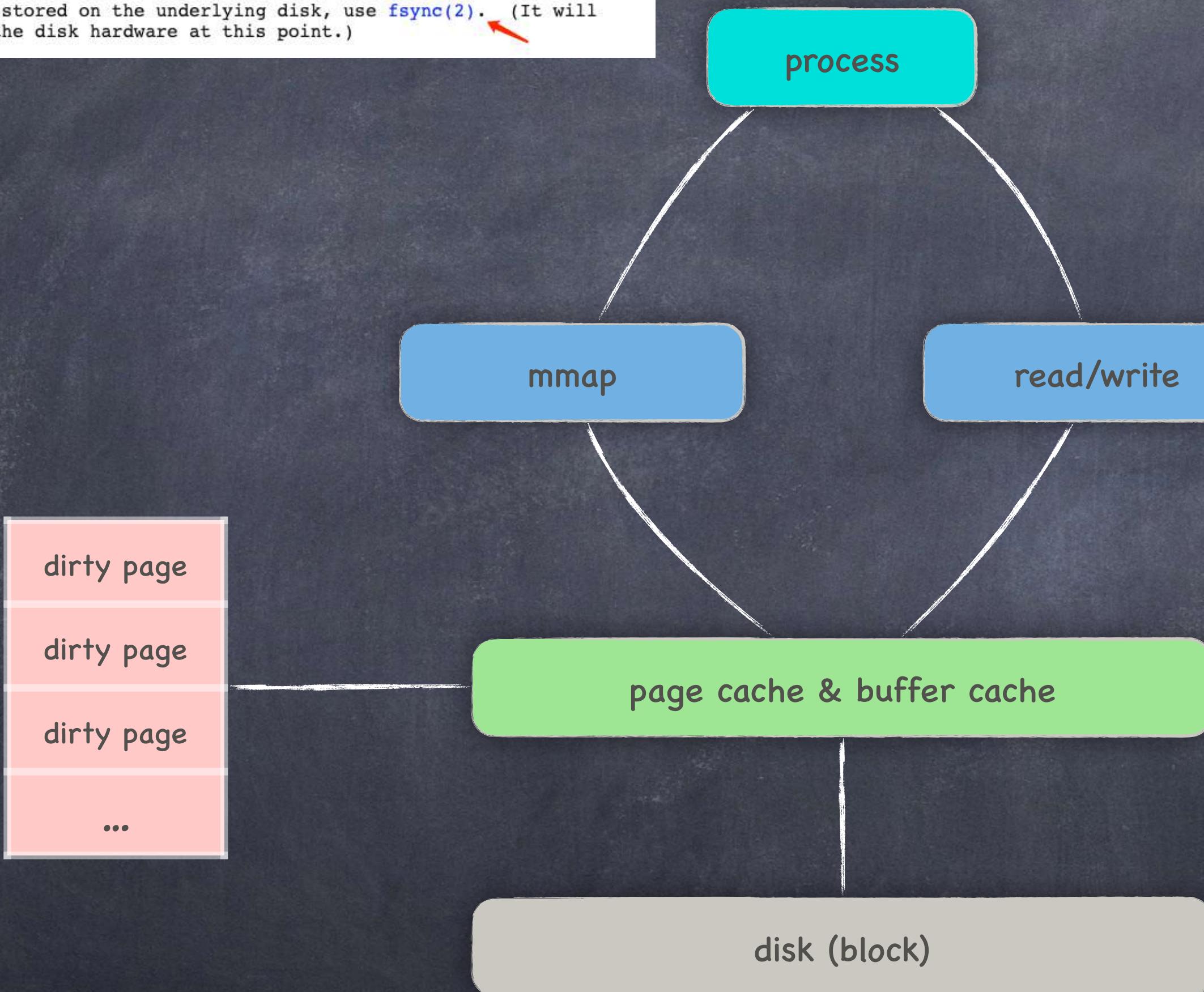
sync, fsync, fdatasync

# page cache

NOTES

A successful close does not guarantee that the data has been successfully saved to disk, as the kernel uses the buffer cache to defer writes. Typically, filesystems do not flush buffers when a file is closed. If you need to be sure that the data is physically stored on the underlying disk, use `fsync(2)`. (It will depend on the disk hardware at this point.)

[top](#)



缓冲写，缓存读

- ① `dirty_background_ratio = 10 %`
- ② 脏页率超过指标开始 **Flush Dirty PageCache**
- ③ `dirty_ratio = 20 %`
- ④ 阻塞所有的写操作来进行**Flush**
- ⑤ `dirty_writeback_centisecs = 500 // 5s`
- ⑥ 检查是否需要 **flush**
- ⑦ `dirty_expire_centisecs = 3000 // 30s`
- ⑧ 把超过 **30s** 的脏页写到磁盘里

# page cache



# summary



高性能要点

减少 IO , 顺序 IO , 批量读写

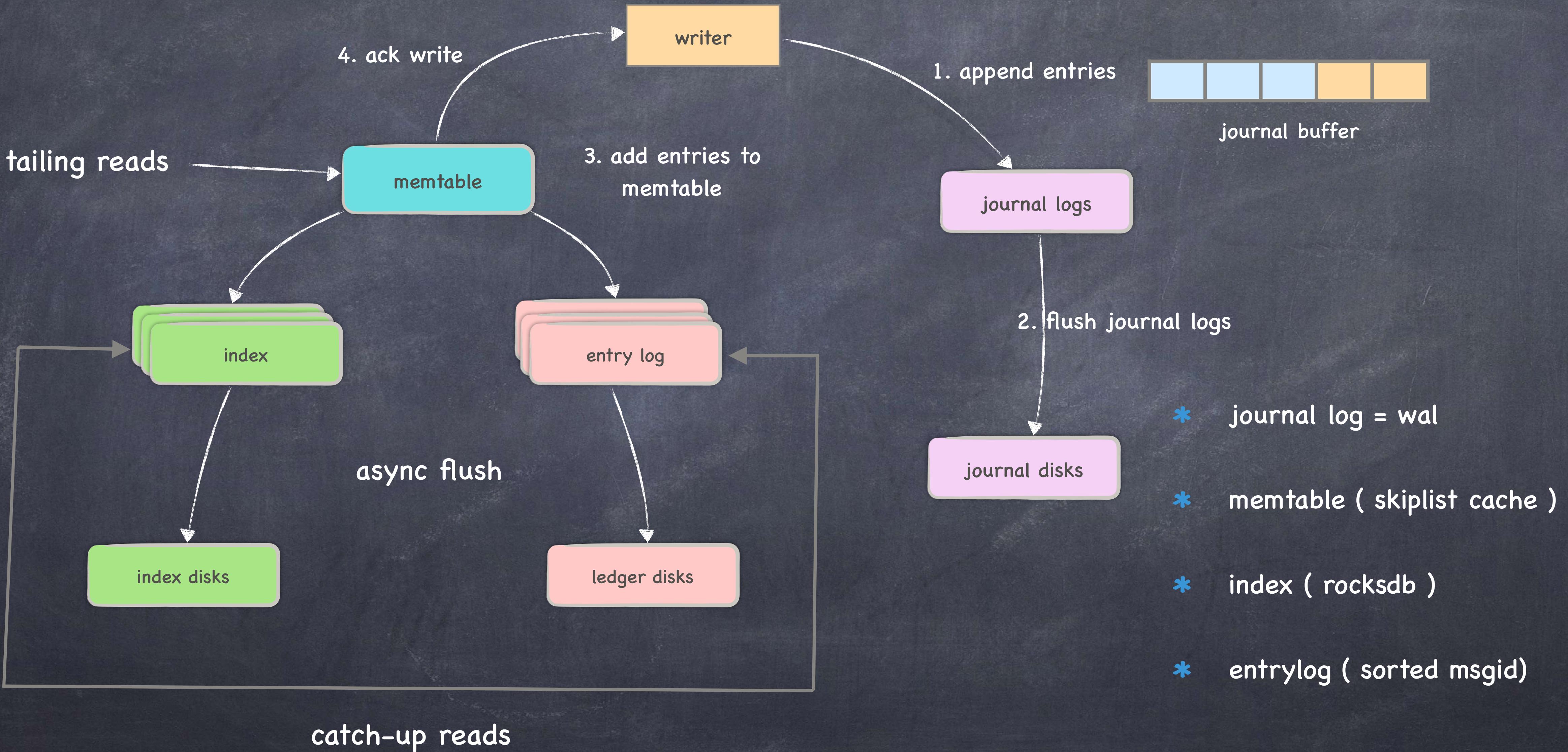




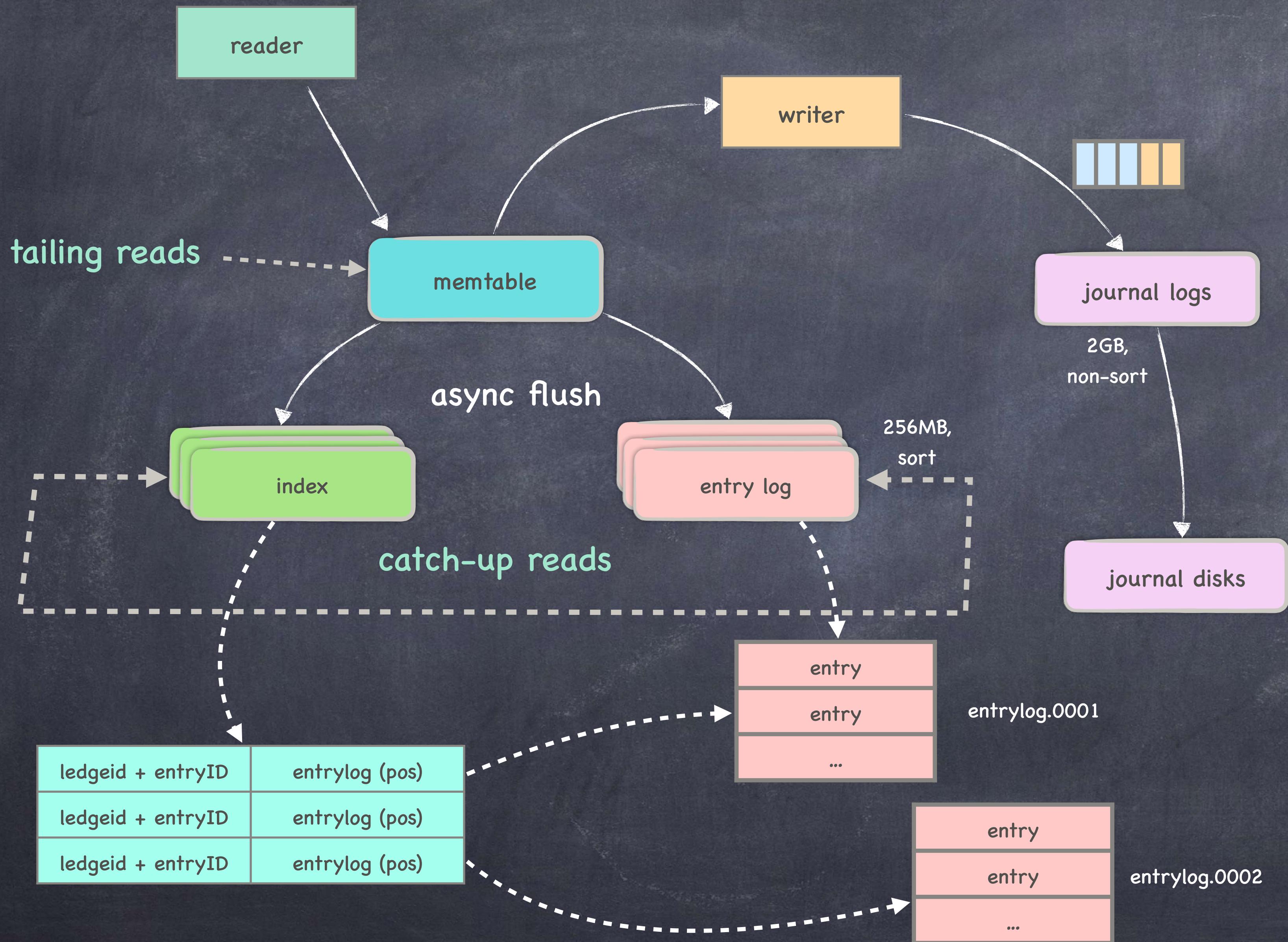
pulsar bookkeeper



# bookkeeper design



# read & write dataflow



高速写入, 高效读取

- \* IO 读写分离设计
- \* journal 高速的写
- \* entry log 实现高效的读
- \* memtable 减少追读缓存污染问题

mysql 的 redo.log ,  
innodb\_buf\_pool, ibd ?

# IO 读写隔离

journal log

ledgerid	entryid	lac	data
3	1000	x	x
1	500	x	x
2	30	x	x
3	1001	x	x
2	31	x	x
1	501	x	x



ledgerid	entryid	pos
2	31	entry.0001, offset
...	...	...
3	1001	entry.0002, offset

index (rocksdb)

entry logger

ledgerid	entryid	lac	data
1	500	x	x
1	501	x	x
2	30	x	x
2	31	x	x

entrylog.0001

entrylog.0002

- \* 如何快速地写数据 ?
- \* 利用磁盘的顺序 `io append` 数据
- \* 乱序写入
- \* 如何快速地读数据 ?
- \* 有序且攒一波 `entry` 排序后写入 `entrylog` 和 `index`
- \* 自定义缓存减少 IO , 借助预读和顺序 IO 读取

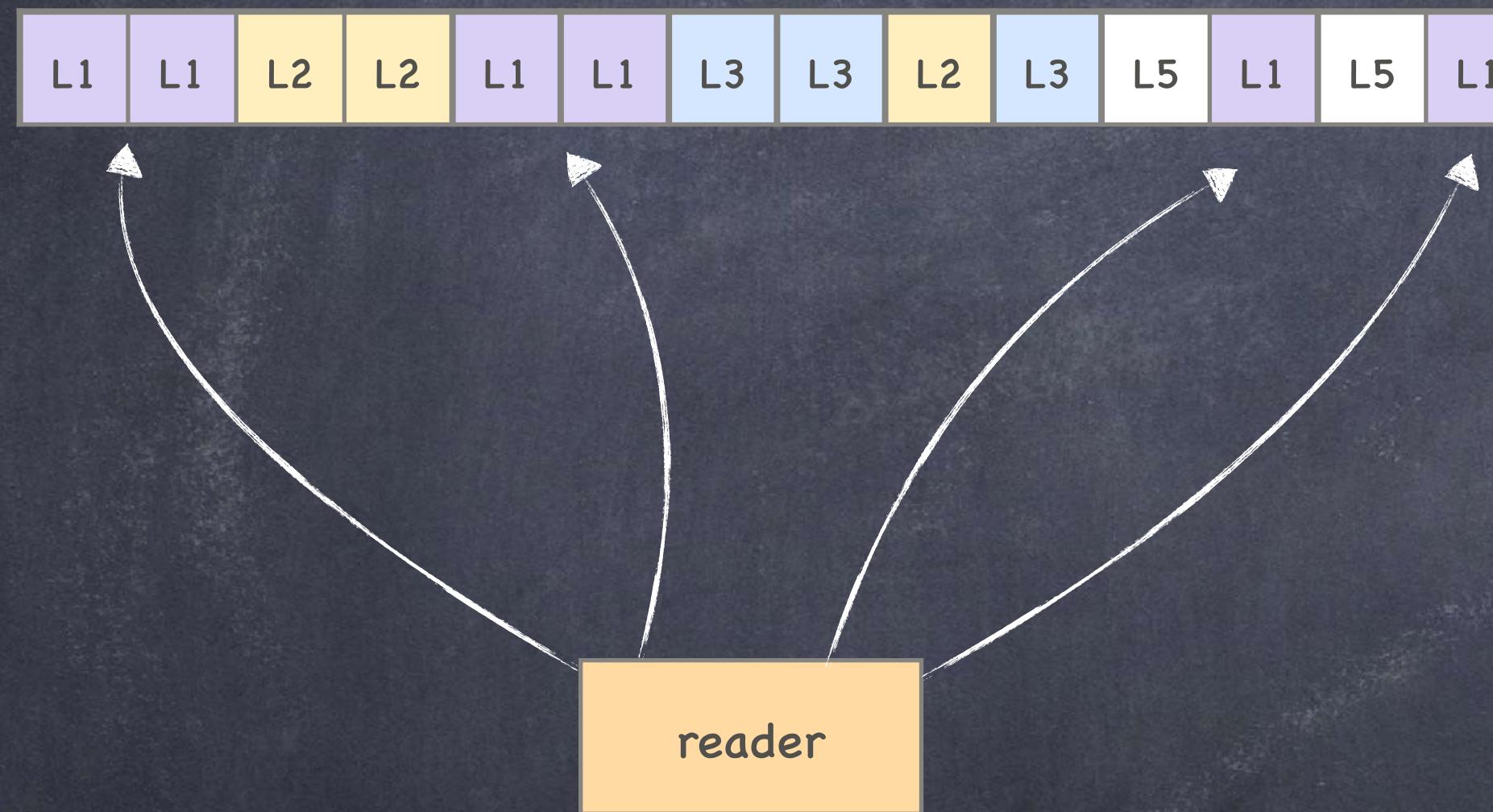


entrylog 为什么要有序？

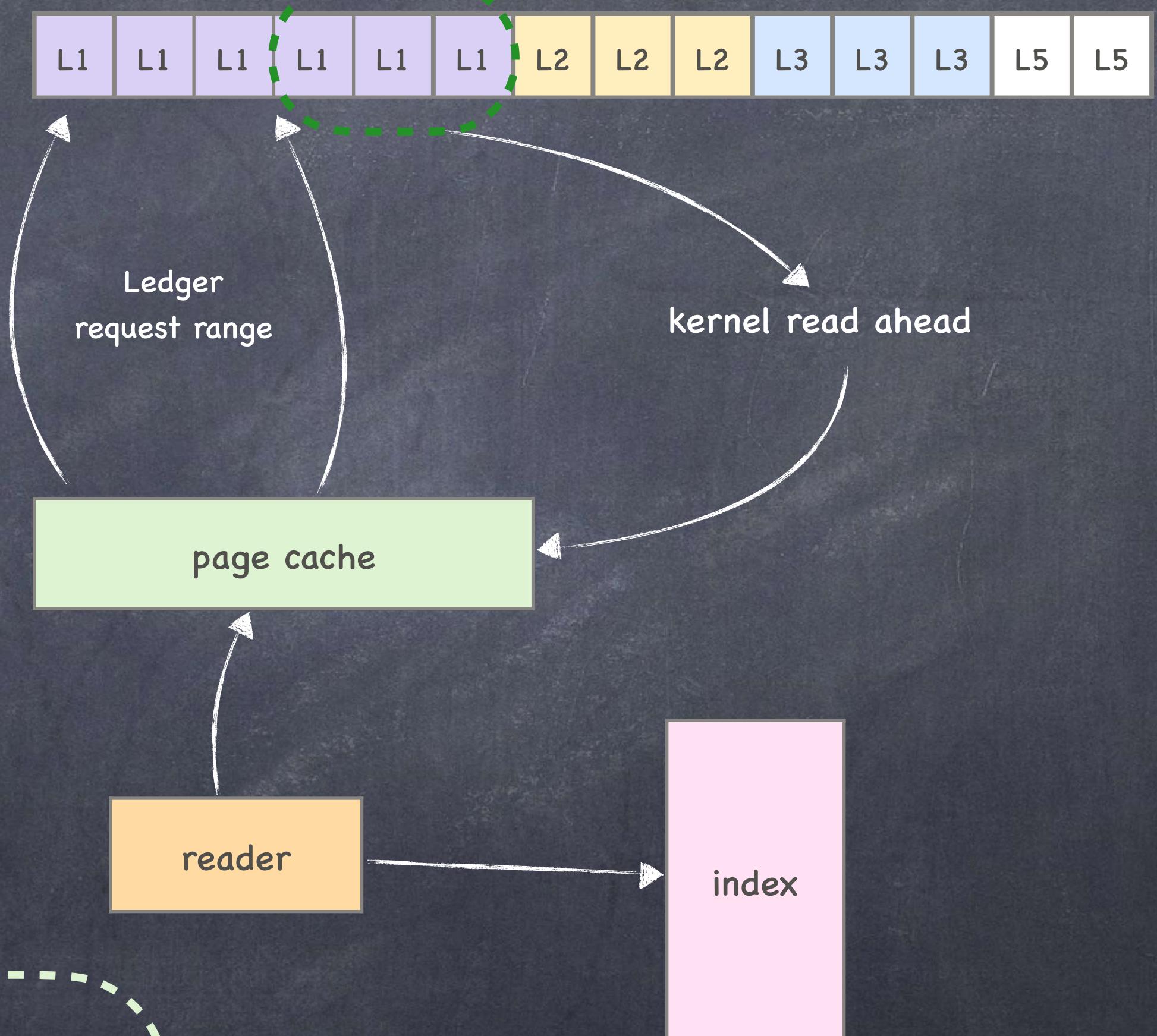
# entrylog io 优化

if entry logger is non-sorted

随机 IO 读取, 退化成点查

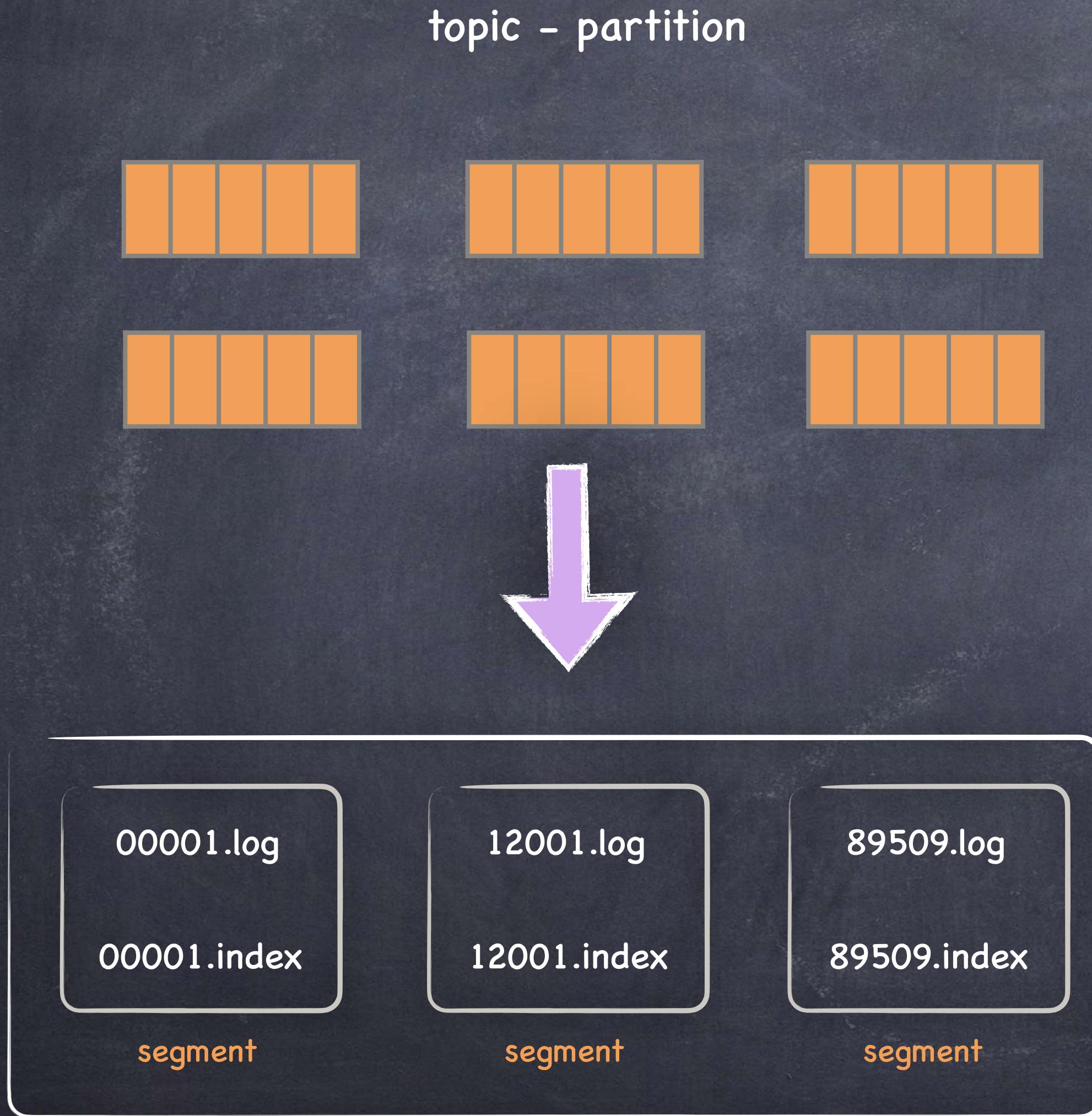


sorted entry logger



利用磁盘的顺序 IO 快速遍历读取,  
且大块的范围读取

# 退化随机 IO 问题



kafka 瓶颈

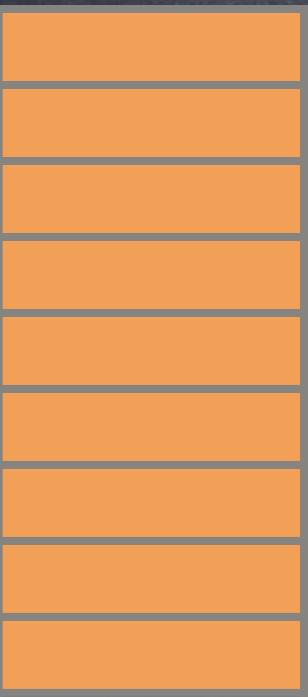
- \* 如果出现很多的 `topic/partition` 的场景，并发下每个分区都会写 `log` 文件。
- \* 单个文件按照 `append` 写是顺序 IO
- \* 文件太多，那么局部的顺序写会退化到随机 io .

$$\text{topic } 200 * \text{ part } 64 = 12800$$

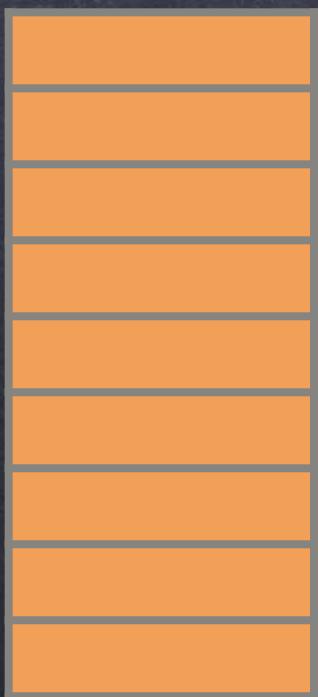
同一时间只写一个  
journal 文件！

journal log

memtable



02.log



01.log

# 退化随机 IO 问题

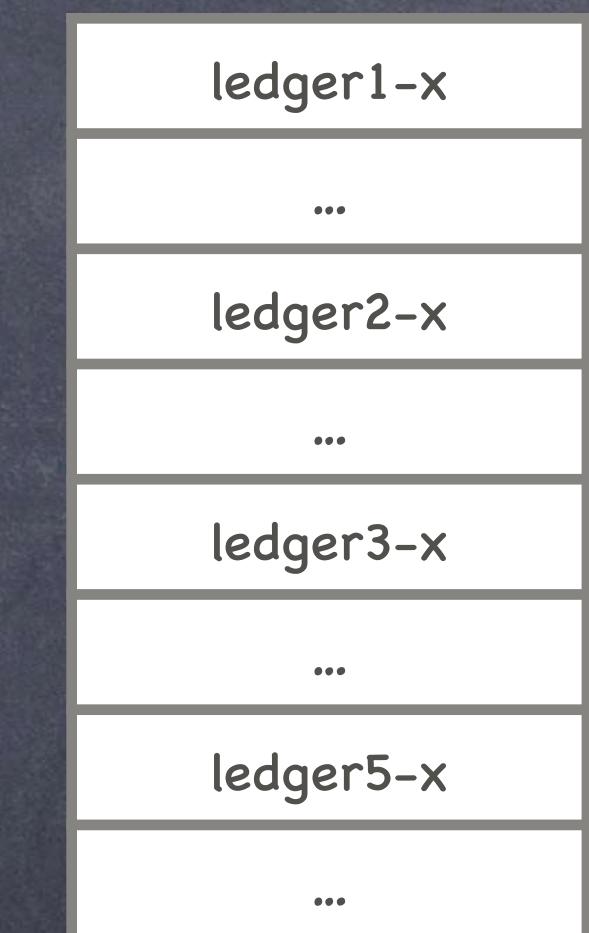
entry log



- \* journal 作为 wal 日志, 不排序直接 append 写入

- \* 当 memtable 超过阈值时, 把排序的消息持久化到 entry-log 文件, 并更新 index.

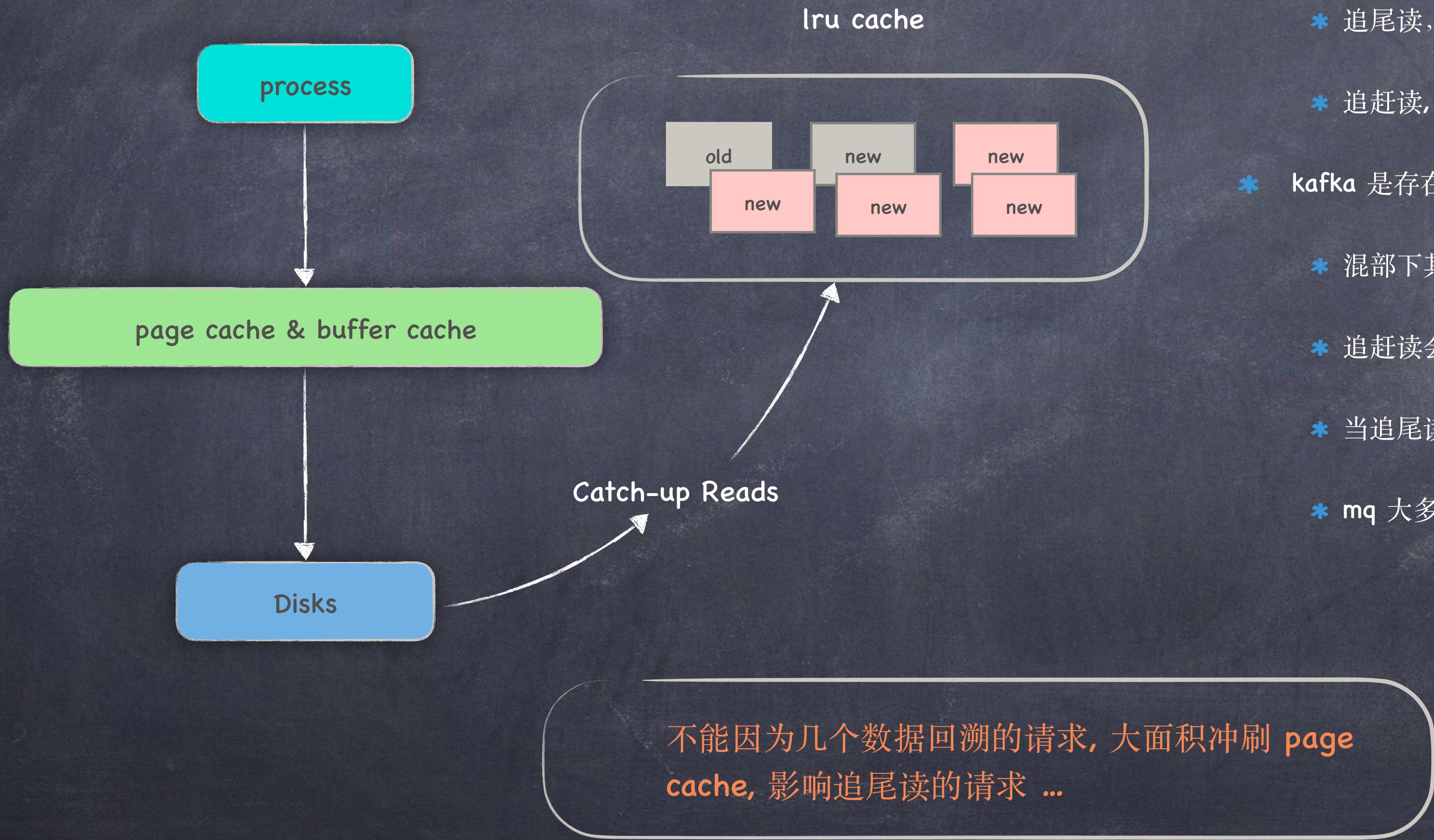
index (rocksdb)



Pulsar

如何解决海量文件下  
退化随机 IO 的问题 ?

# 缓存污染优化



## \* Tailing reads & Catch-up reads

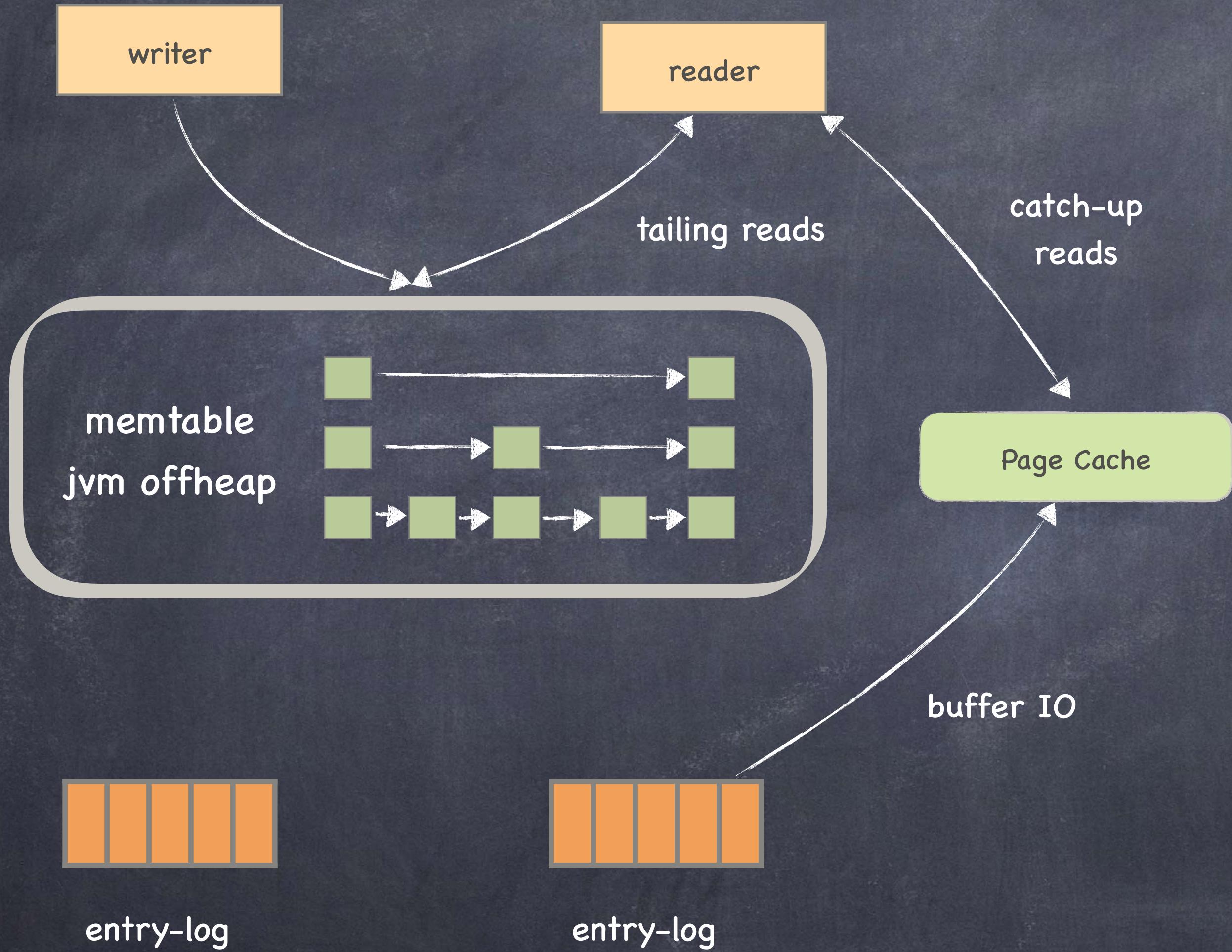
- \* 追尾读, 读最新的数据, 大概率在缓存中 ;
- \* 追赶读, 读老数据, 可能未在缓存里 .

## \* kafka 是存在缓存污染问题

- \* 混部下其他服务冲刷 page cache, 容器也无法管控 page cache
- \* 追赶读会把 page cache 中的较新的 page 页冲刷掉
- \* 当追尾读发生 cache-miss, 需要从磁盘中读取数据
- \* mq 大多数是追尾读, hh

# 缓存污染优化

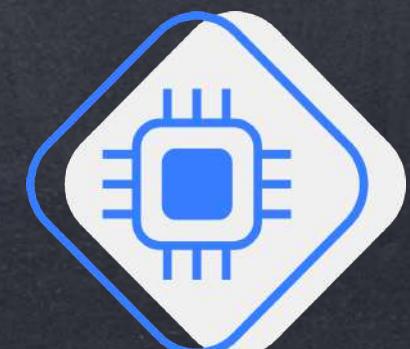
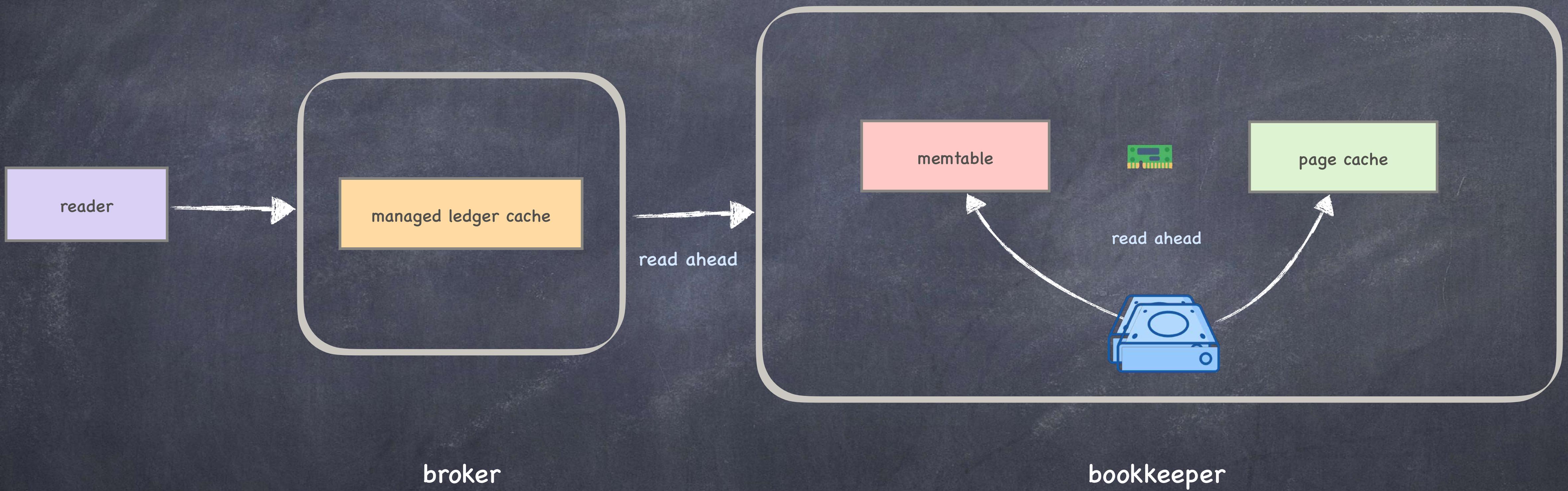
其他大厂二开 kafka 做了此优化.



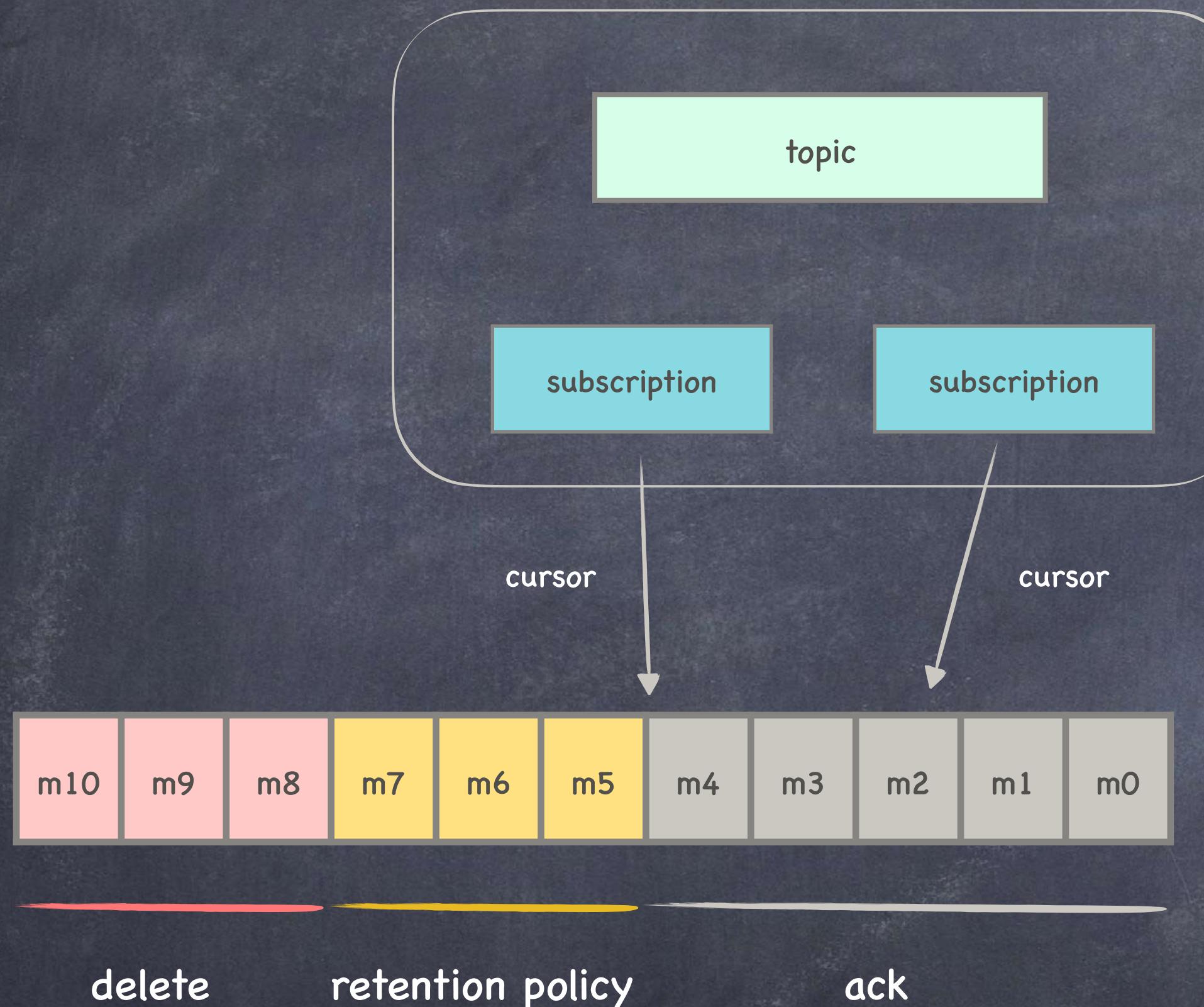
- \* 在 jvm 里构建 **memtable** 堆外缓存；
- \* jvm 的缓存可控，不像 **page cache** 受各种情况影响.
- \* **catch-up reads** 过程
  - \* 尝试从 **memtable** 寻找；
  - \* 使用 **buffer IO** 读取文件，尝试从 **page cache** 寻找；
  - \* 读到的数据会先写到 **page cache**，则返回给用户态；
  - \* 利用 **read ahead** 预读，命中预测，成倍预读.

pulsar 如何优化缓存污染的问题

# dataflow under multi cache layers



# retention



retention 判断点取决于 topic 最慢的那个 cursor.

## \* retention

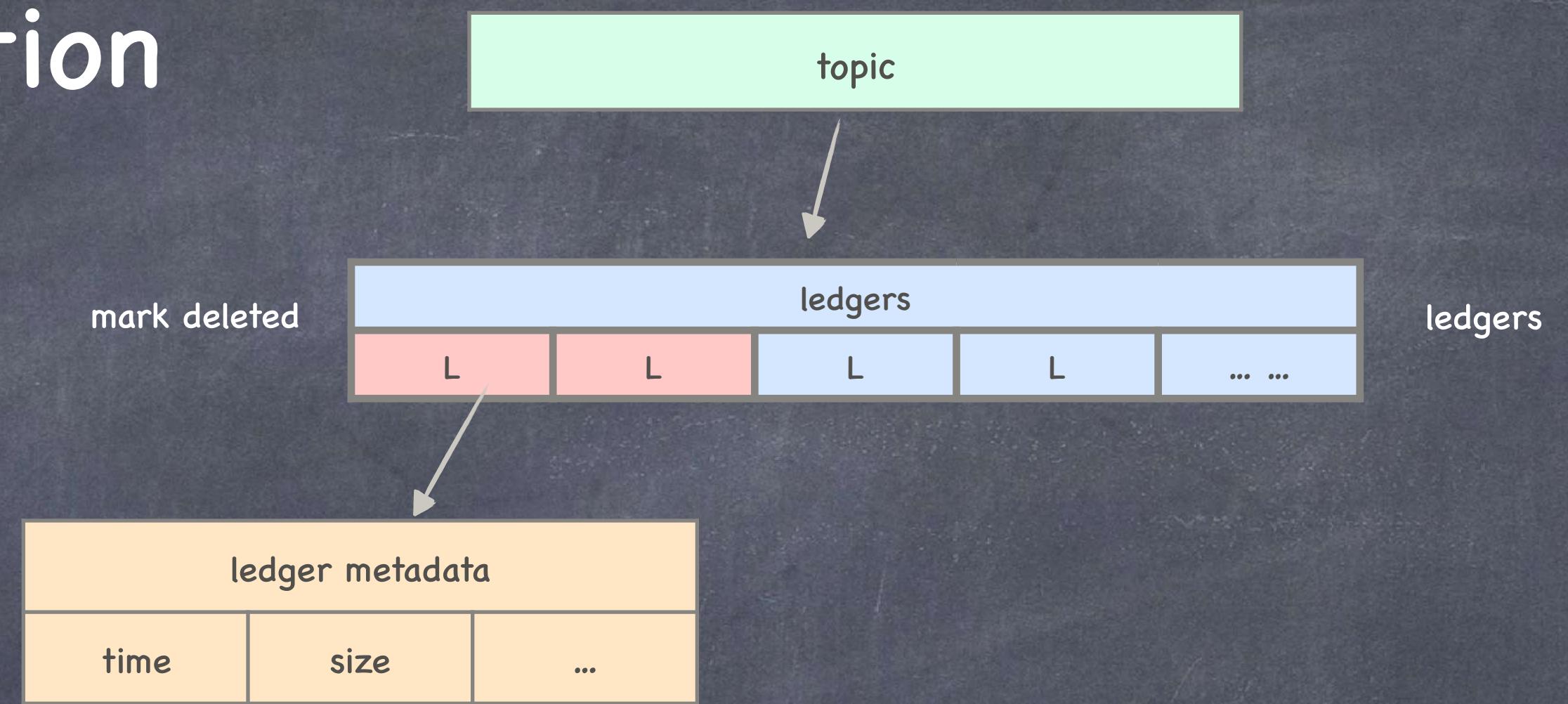
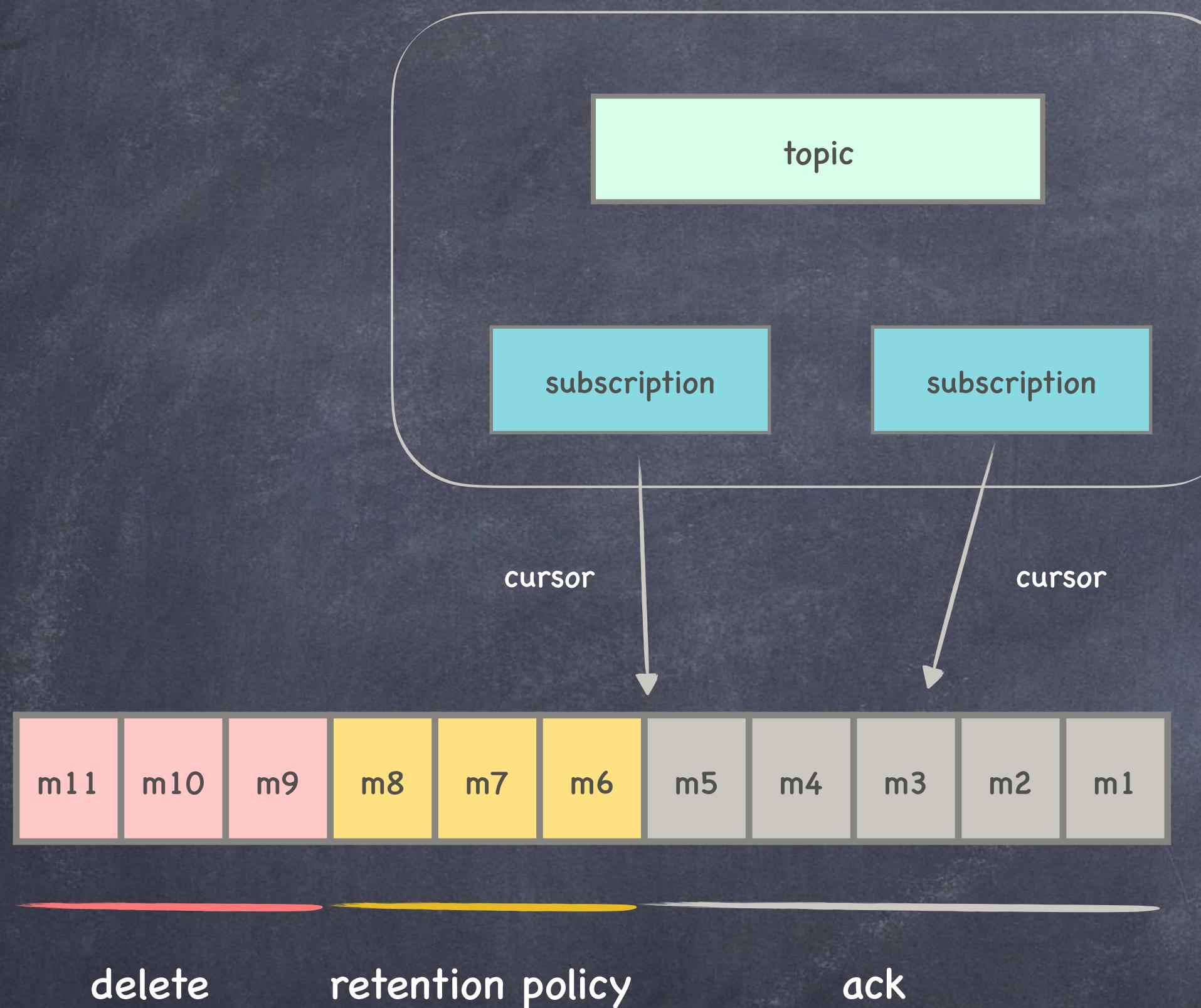
\* 当消息被 ack 之后, 在 bookie 保留多久的时间

## \* policy

\* 按照 bytes 存储大小保留数据

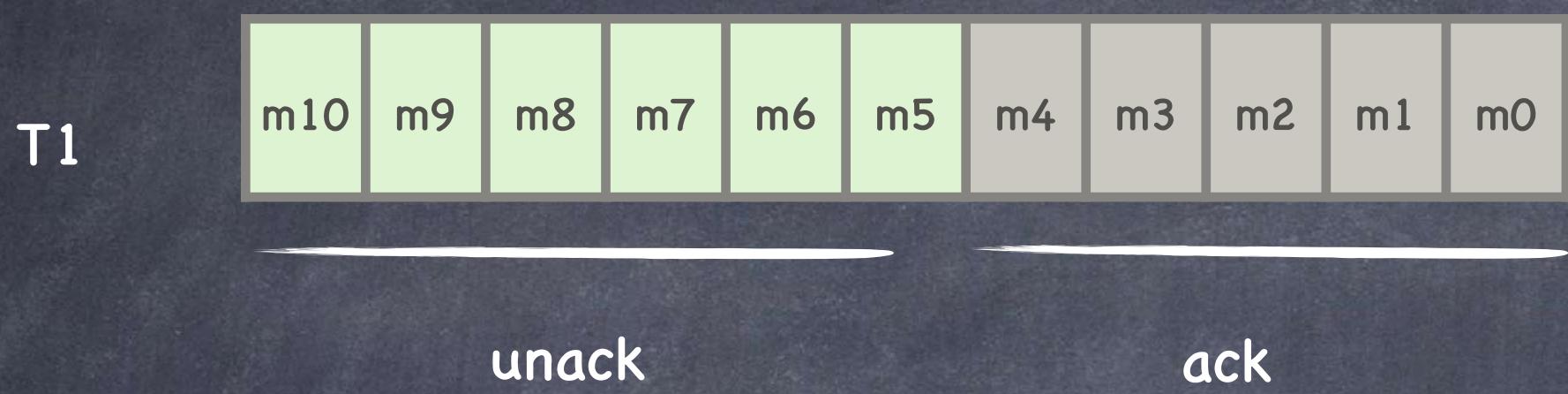
\* 按照 time 时间保留数据

# retention

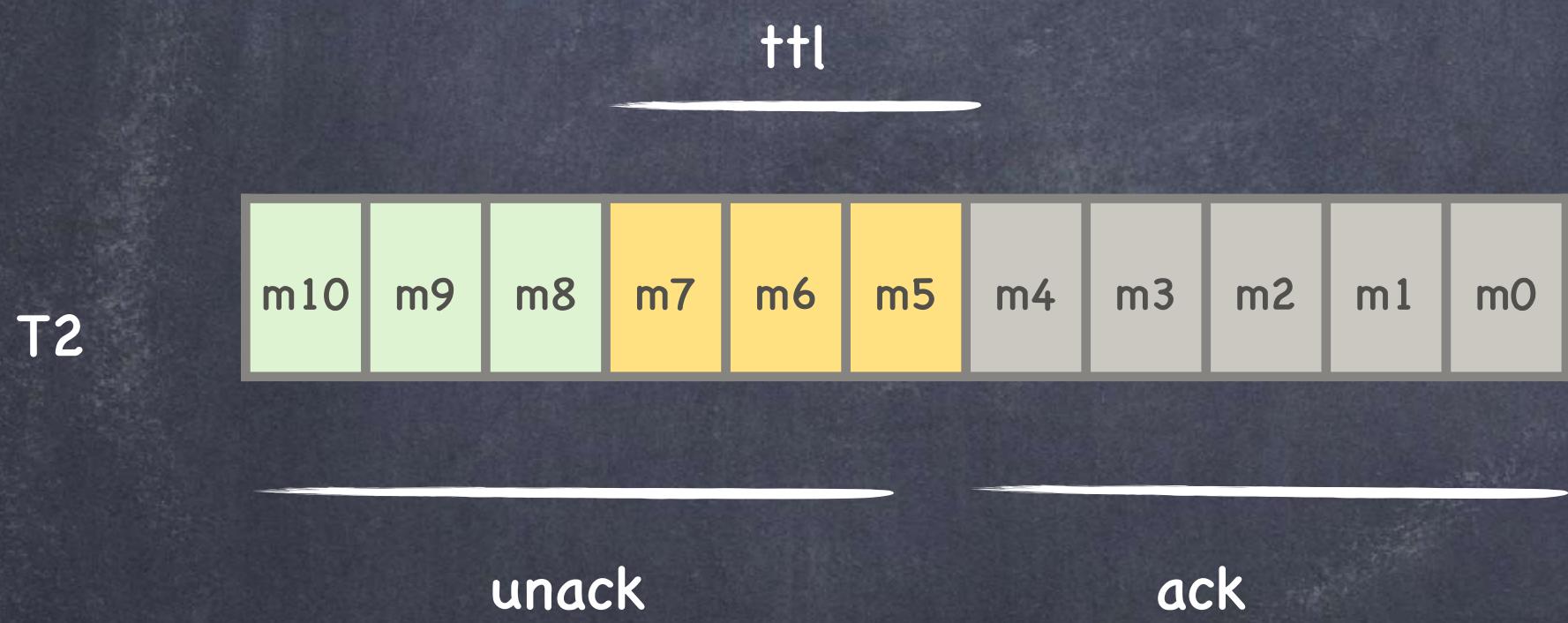


- \* 扫描其 **broker** 上所有关联的 **topic**；
- \* 通过时间序来遍历所有的 **ledger**；
- \* 根据 **ledger** 的活跃时间和数据大小, 判断是否满足 **retention** 条件；
- \* 在 **zookeeper** 中标记删除过期的 **ledger**；
- \* **bookKeeper** 通过 **gc** 机制来清理已过期的 **ledger**.

# TTL 消息过期

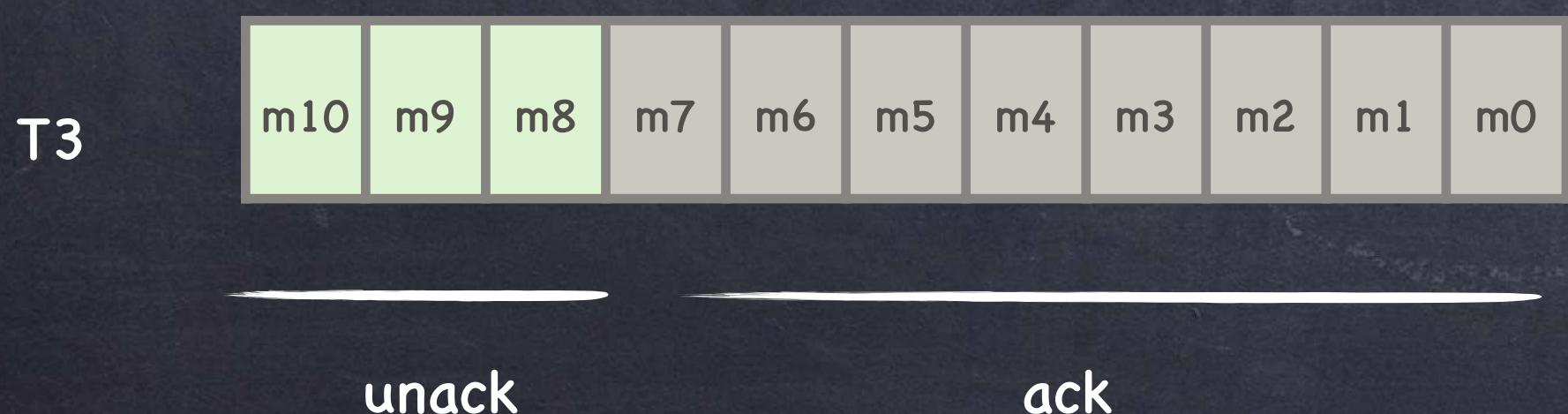


如果 **topic** 下的某个 **subscription**, 总是不消费, 不提交 ???

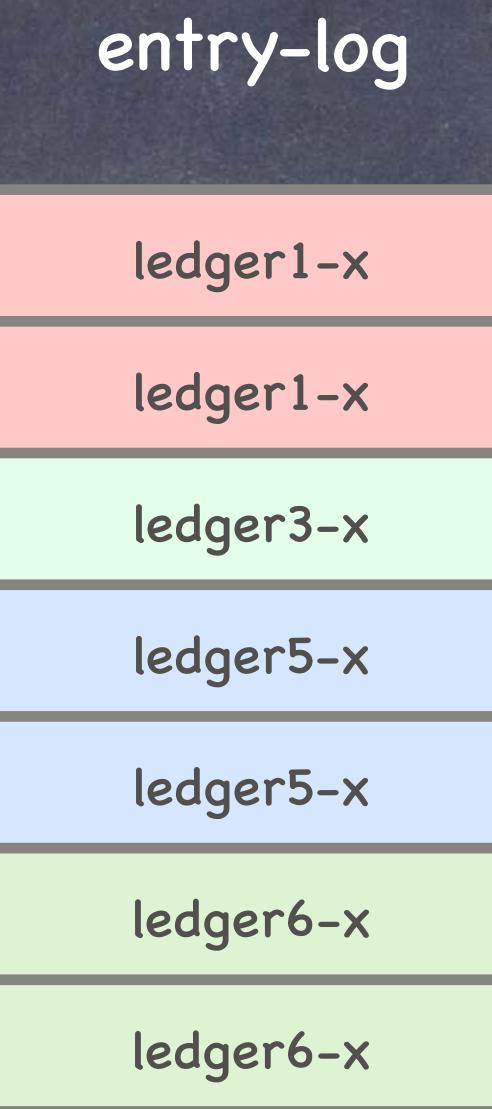


## • TTL

- 指定时间内没有被用户 **ack**, **broker** 主动帮你 **ack**.
- TTL 本质是把 **cursor** 往后移动 !!!
- 订阅者会读到移动 **cursor** 之后的新数据.



# bookie gc 基本原理

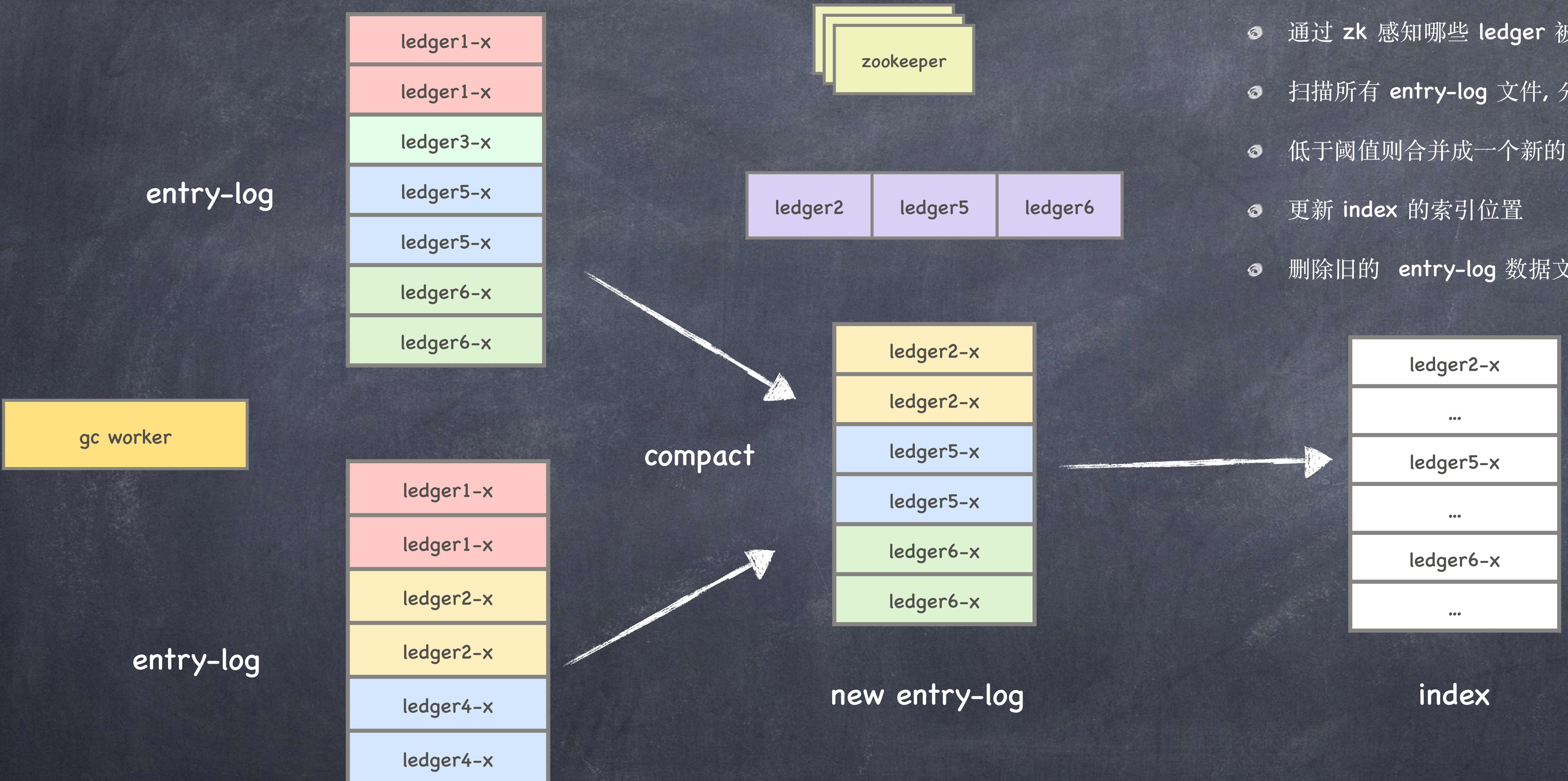


**entrylog** 中含有多个 **ledger** 的 **entry**.  
如何清理删除文件中无效 **ledger** 的数据 ?

**pulsar** 通过 **entry-log** 的方式解决了随机 **io** 的问题，那么如何进行数据删除？

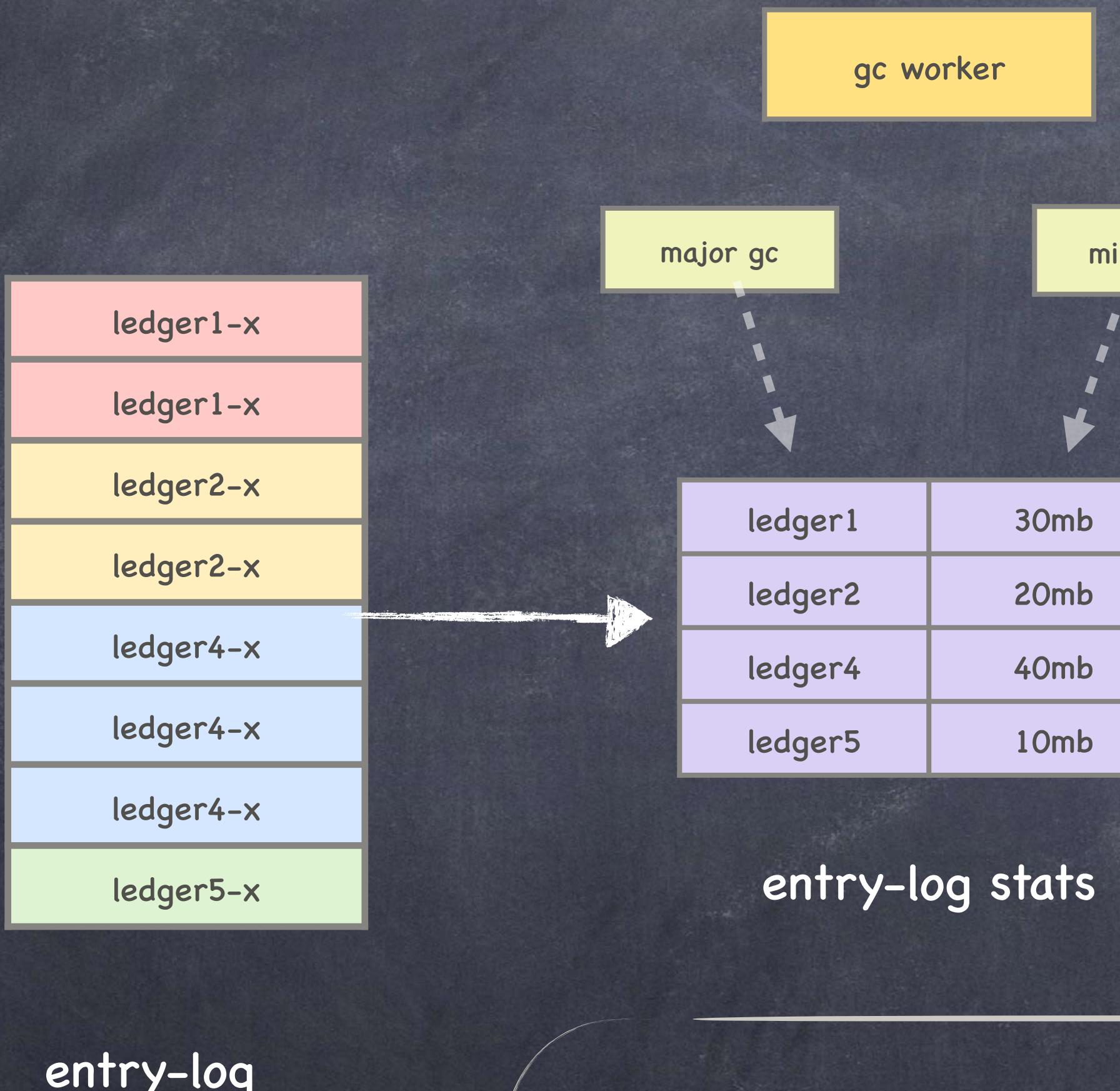
- ② **pulsar** 通过 **entry-log** 的方式解决了随机 **io** 的问题，但 **gc** 无疑是复杂的.
- ③ 对比其他的 **mq** 实现
  - ④ **kafka** 直接删除对应 **segments** 文件即可
  - ⑤ **rocketmq** 跟 **pulsar** 的实现类似

# bookie gc 基本原理



- 通过 zk 感知哪些 ledger 被标记删除
- 扫描所有 entry-log 文件, 分析其有效数据占比
- 低于阈值则合并成一个新的 entry-log 日志
- 更新 index 的索引位置
- 删除旧的 entry-log 数据文件

# bookie gc 基本原理



## ② Major GC

- ② 24h interval, valid data < 80% trigger

## ③ Minor GC

- ② 1h interval, valid data < 20% trigger

## ④ e.g.

- ② 如果 ledger 1,2,4 被标记删除, 有效数据占比是 10%, 符合 minor/major 阈值, 执行 gc 垃圾回收.
- ② 如果 ledger 5 被标记删除, 有效数据占比是 90%, 不符合任意的清理阈值, 不进行 gc 垃圾回收.

阈值的作用是牺牲一些空间开销来减少 cpu 开销 .

# bookie gc 基本原理

如何减少 gc 操作对线上服务的影响 ?

按照 bytes 进行限速



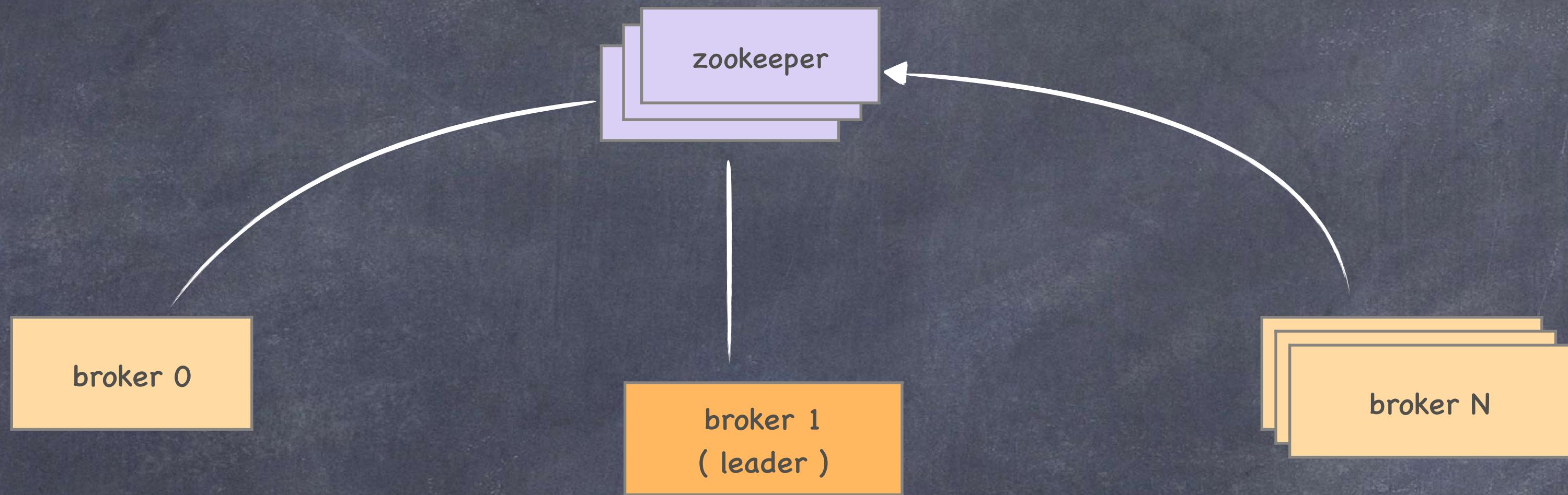
按照 entry 个数进行限速



## advanced features

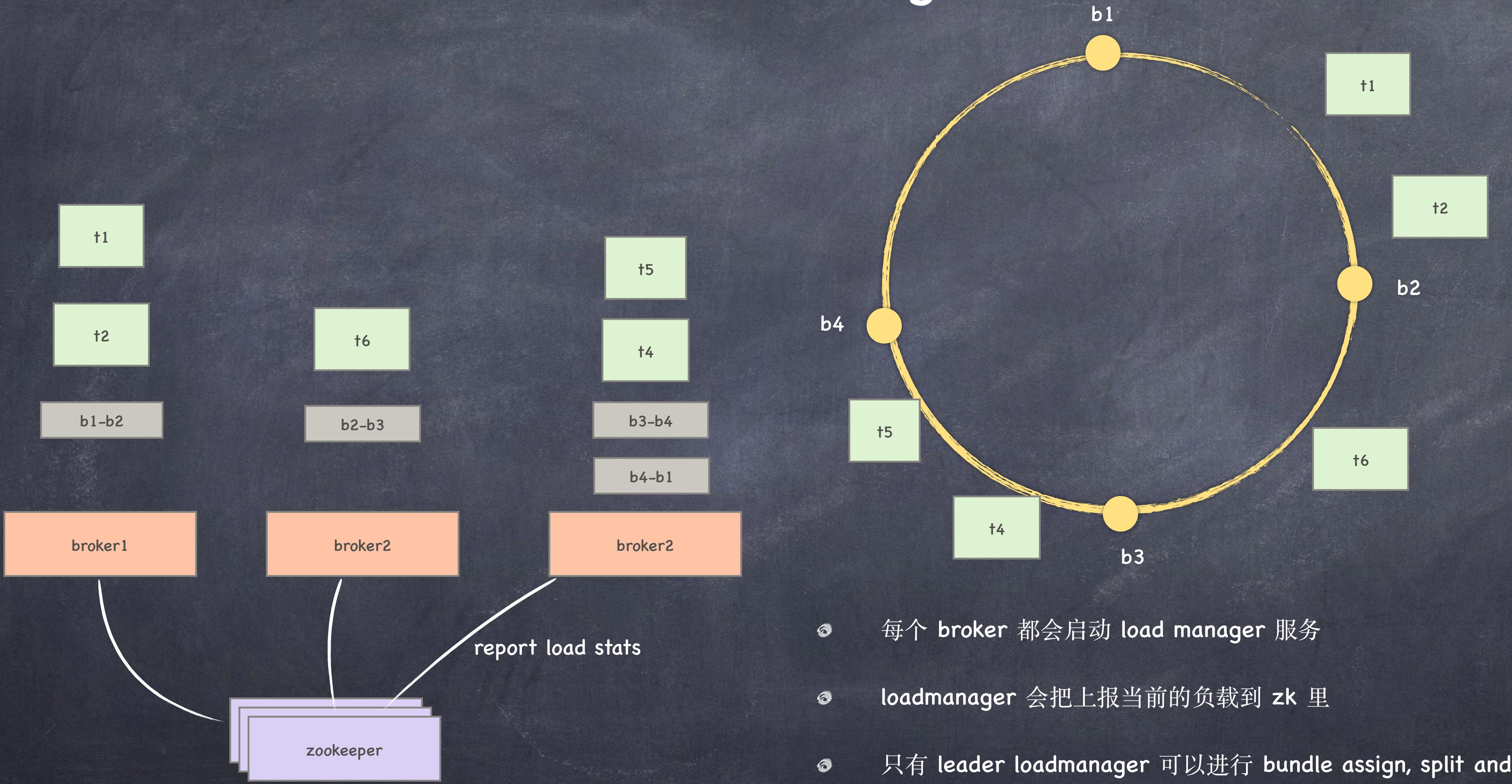


# leader election

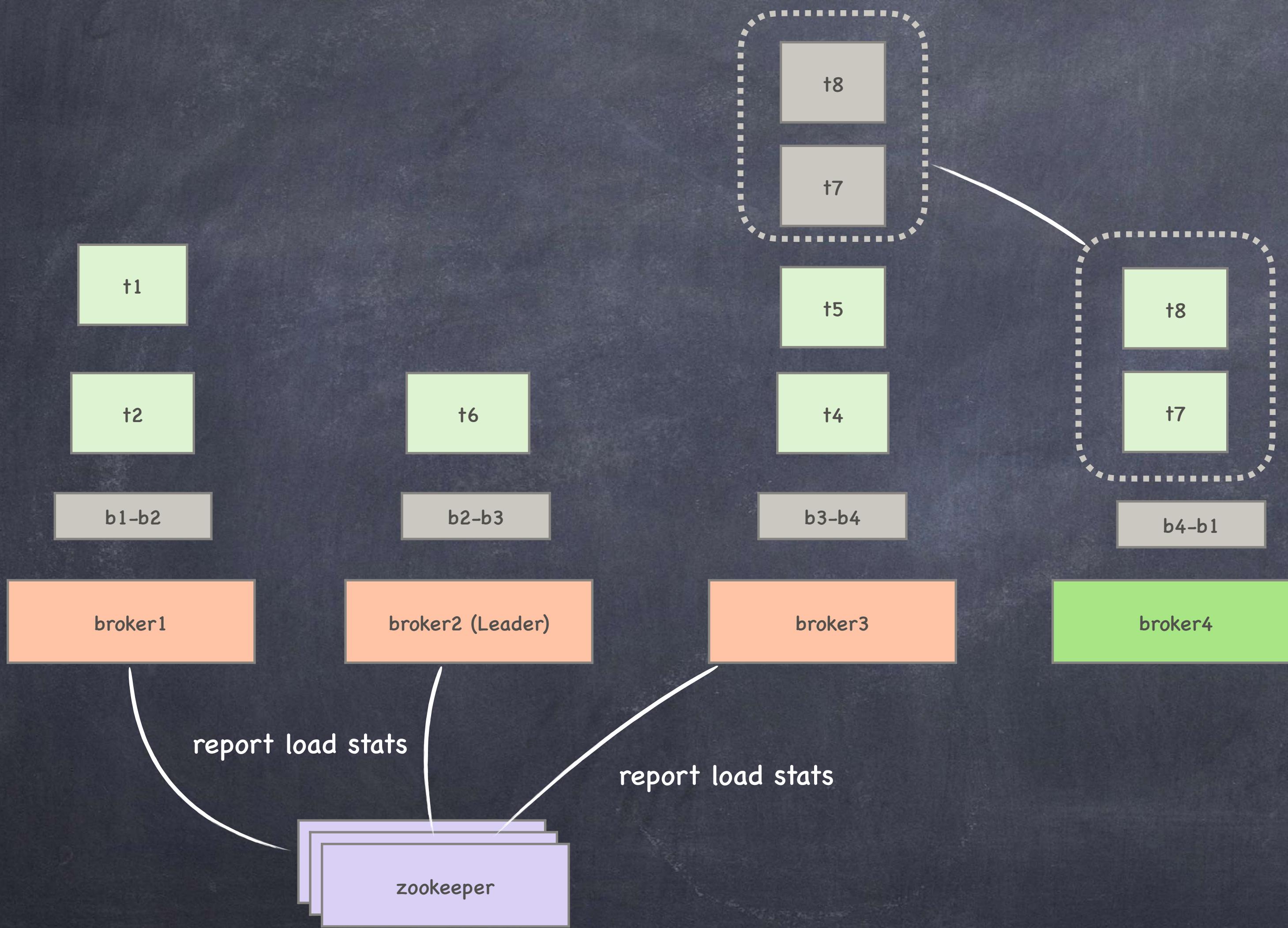


在“/loadbalancer/leader”目录创建临时节点，且 zxid 最小的客户端为 leader .

# load manager



# load manager

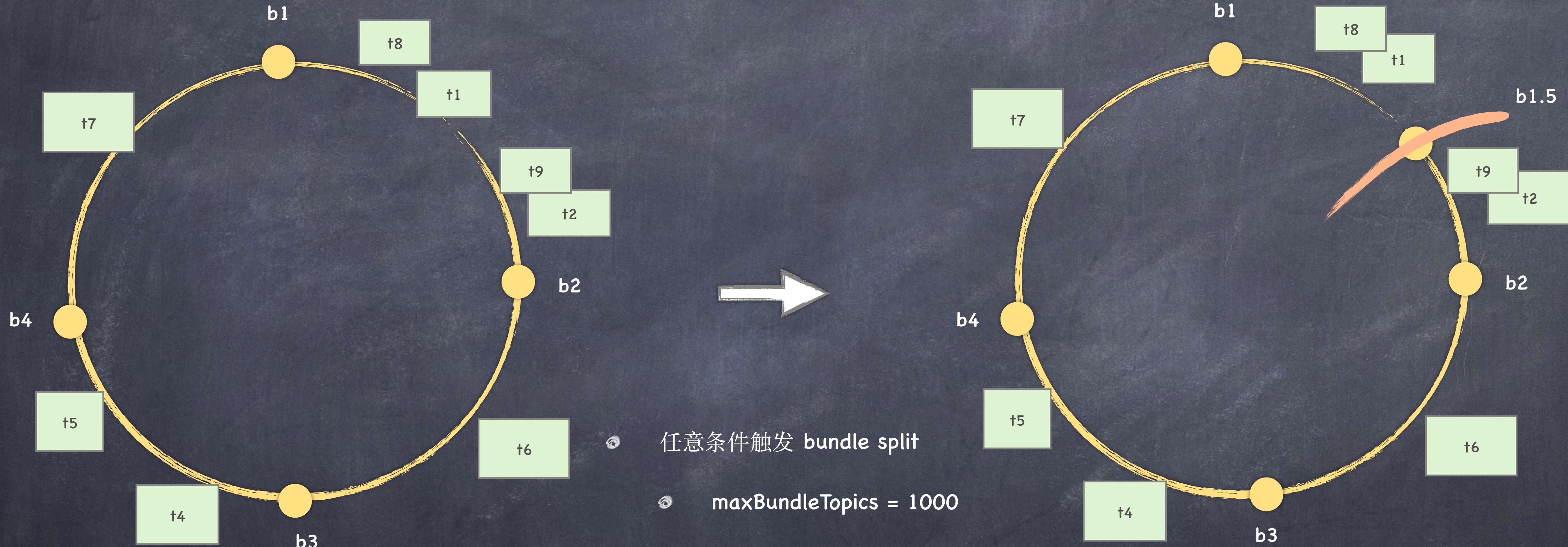


- 扩容一个新的 broker

- load manager 通过负载均衡算法迁移高负载的 broker bundle 到低负载的 broker 上.

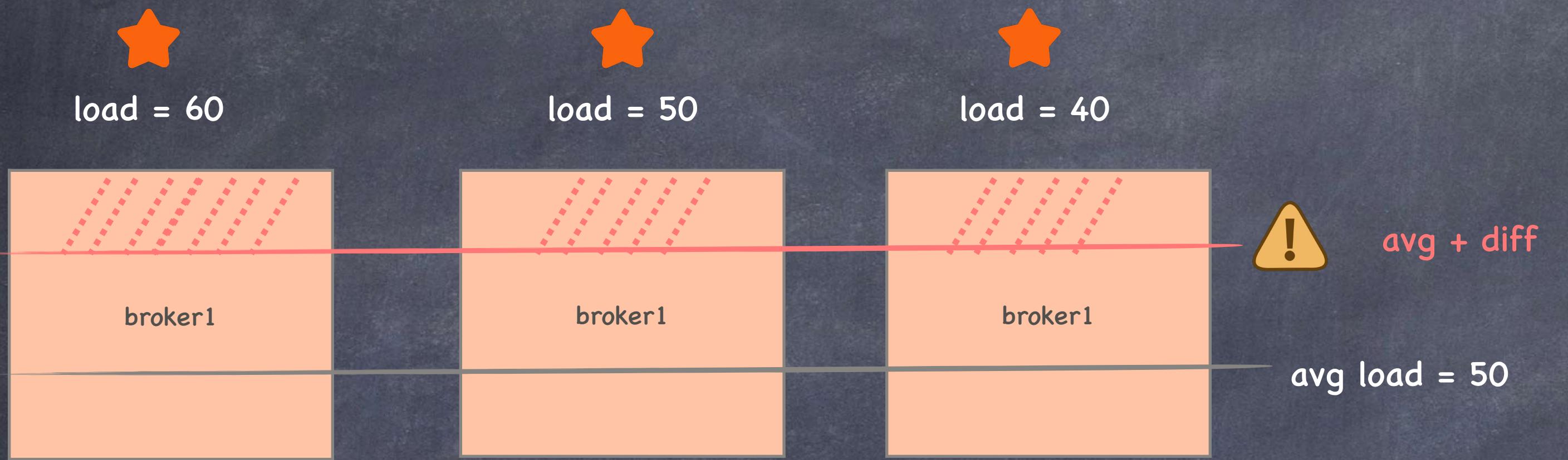
为避免不限扩张 **bundle**, 限定  
namespace 下最大 **bundle** 为 128.

# bundle split



- 任意条件触发 **bundle split**
- maxBundleTopics = 1000**
- maxBundleSessions = 1000**
- maxBundleMsgRate = 30000**
- maxBundleBandwidth = 100MB**

# load threshold



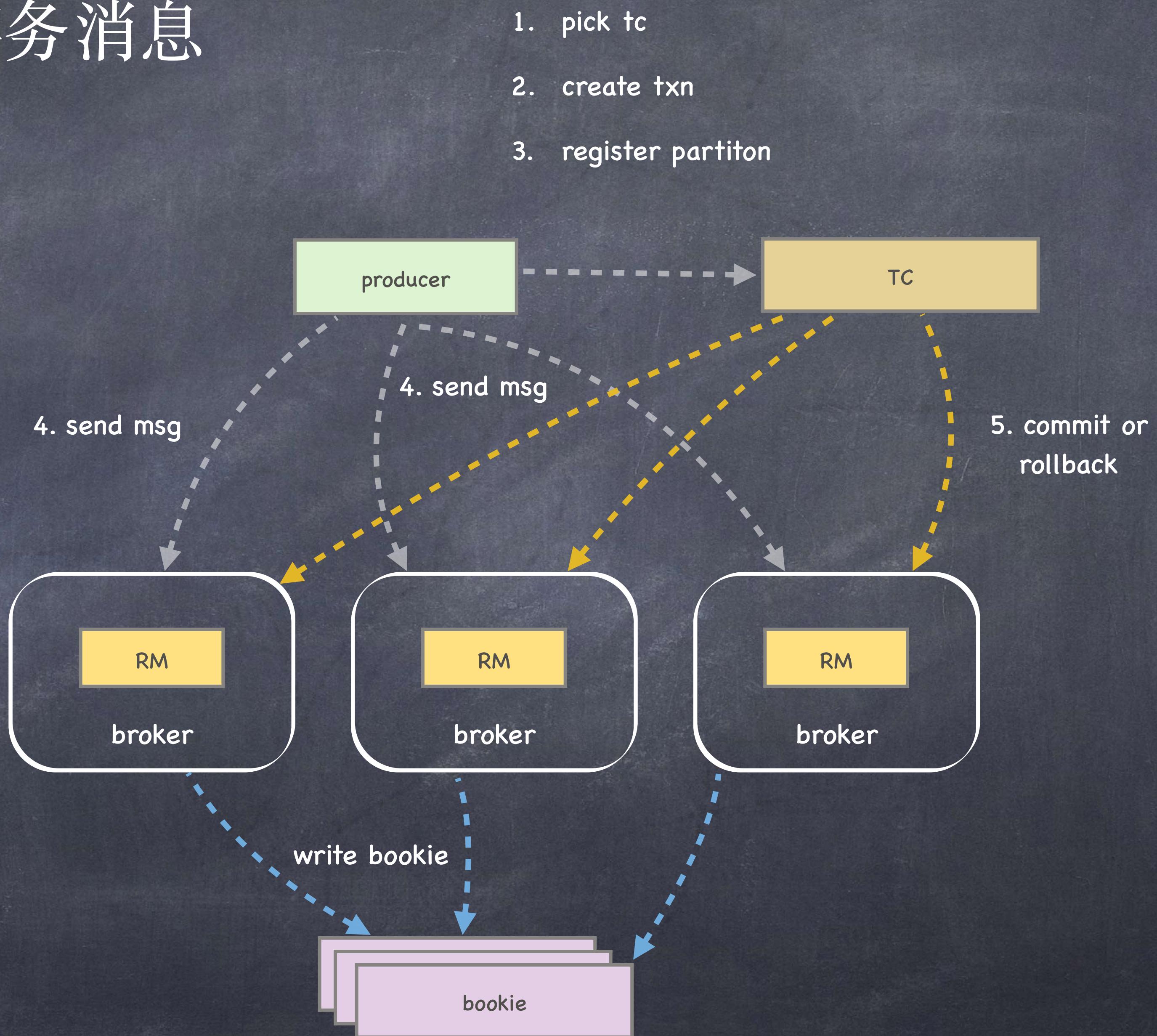
- 如何求平均负载 ?
  - $\text{load} = \text{sum( broker 负载值)} / \text{broker 的节点数}$
- 单个 broker 负载率 ?
  - $\text{broker avg} = (\text{cpu} \times \text{cpu 权重}) + (\text{流量/流量最大阈值} \times 100 \times \text{流量权重}) + (\text{qps / 最大值 qps} \times 100 \times \text{qps 权重}) \dots \dots$

- 按照静态预设的负载阈值 **overload**
  - 静态的配置负载阈值，默认配置是 **85%**。该策略为默认策略
- leader** 扫描所有的统计过的负载信息，大于阈值的对其做 **unload** 卸载 **bundle** 操作。

- 按照动态的阈值 **threshold**
  - 可以动态的对集群内 **broker** 进行均衡，扫描所有负载计算出 **avg** 平均值，如果高于平均值，则对该 **broker** 的 **bundle** 进行卸载迁移。
  - 为了避免过于敏感，判断是否卸载时加入了波动阈值，只有大于 **avg + diff threshold** 才会触发。

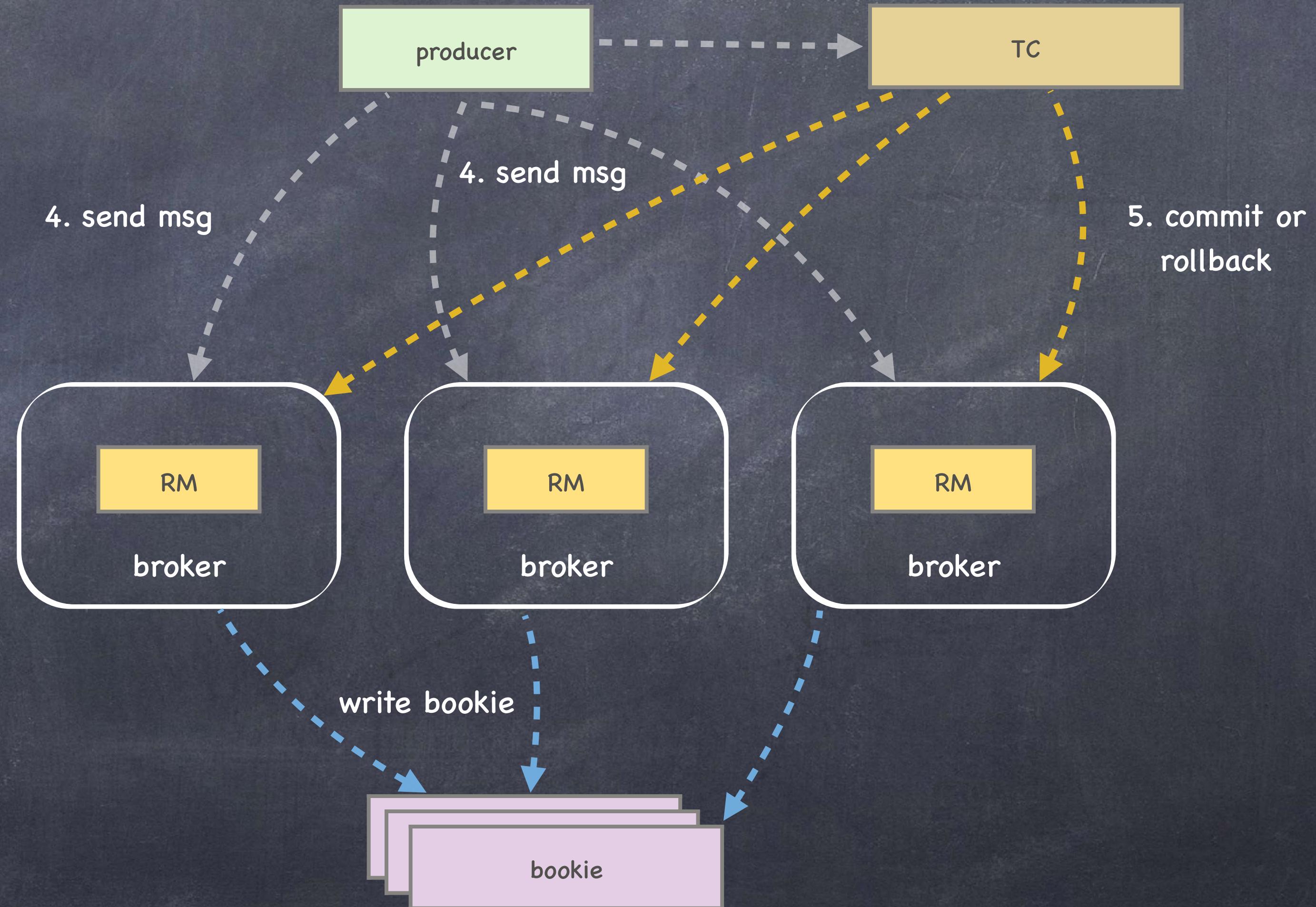
# 事务消息

- ⑥ TM 事务发起者
- ⑥ RM 资源管理者
- ⑥ TC 事务协调者
- ⑥ 选择 tc， 默认 system 有 16个 tc 事务协调者，我们通过轮训的方式选择一个可用的 tc。
- ⑥ 开启事务，从 tc 创建事务，且返回一个唯一的事务 tcid；
- ⑥ 注册分区，让 tc 知道消息会发到那些个 broker 么上；
- ⑥ 发送消息，跟普通消息相比，消息是先进到 broker 的 rm 里，由 rm 来写入，然后记录相关的元信息，但不更新 max read position 位置信息，这样消费者就读不到还未提交的我消息；
- ⑥ 提交事务， tc 会广播给 tcid 相关的 rm，让其进行提交，更新元数据，让消费者可以读。

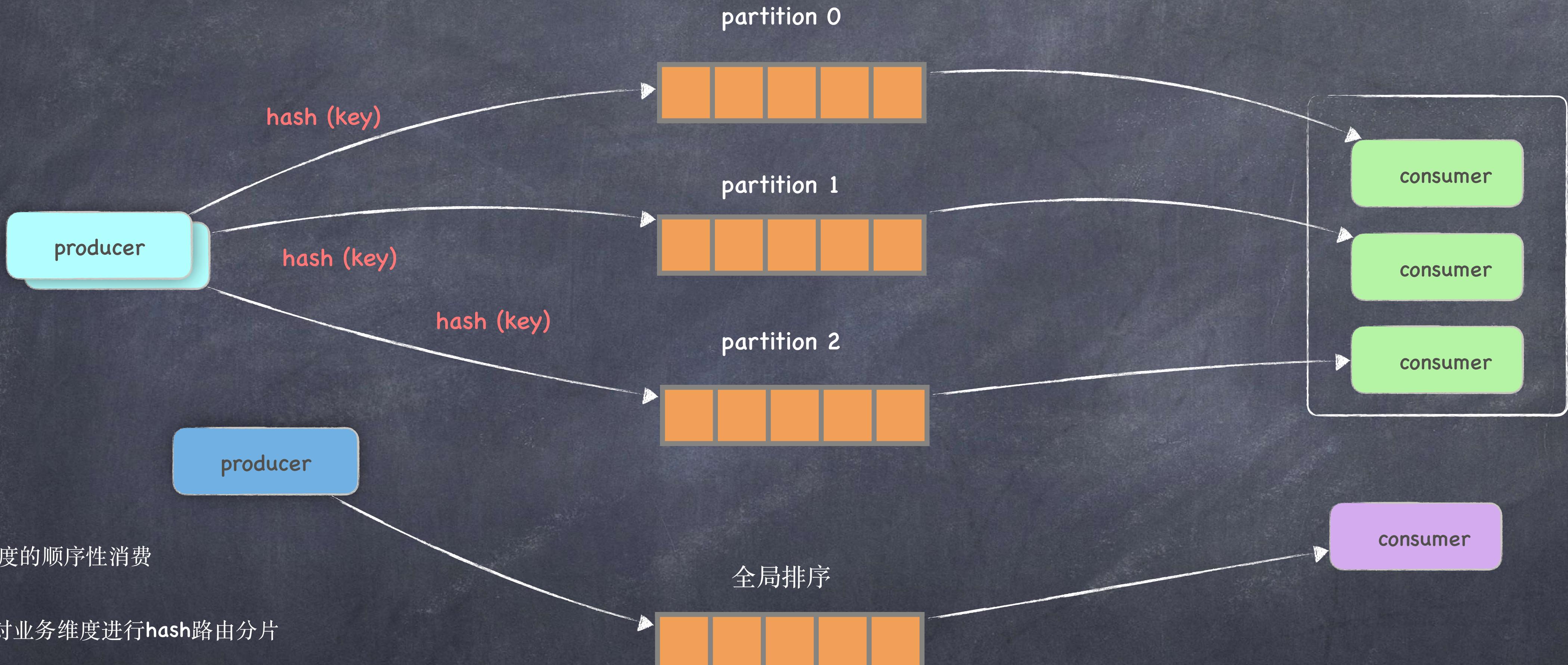


# 事务消息

- 已经写入到 **bookie**, 但未提交, 是否可见 ?
  - 不可见, 只有 **commit** 才移动 **read pos** 位置.
- 回滚如何实现 ?
  - 关闭事务, 给 **rm** 追加 **aborted** 消息.
- **commit** 时, 部分 **rm** 成功, 部分 **rm** 失败
  - **tc** 只要收到 **commit** 就认定消息已提交
  - 开启定时任务补偿一致性
- 事务不提交会出现什么问题 ?
  - 阻塞其他生产者, 直到 **client** 进行提交回滚
  - 触发事务超时关闭



# 消息的顺序性

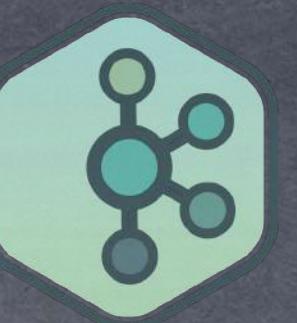


- 业务维度的顺序性消费

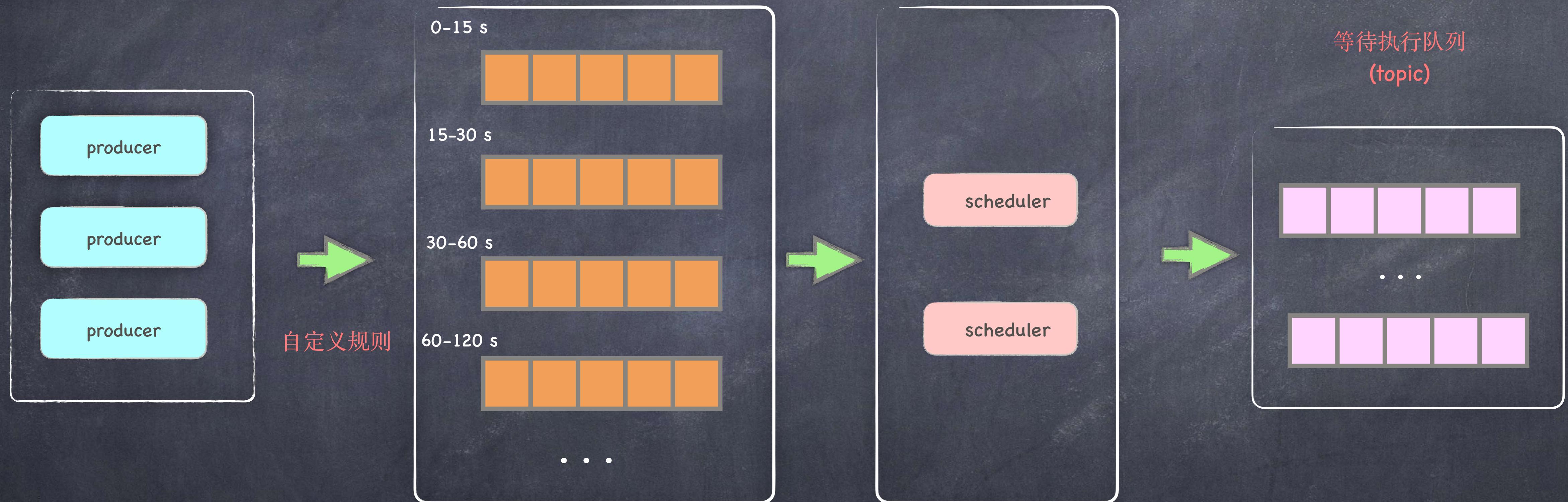
- 对业务维度进行hash路由分片

- 全局顺序性消费

- 设定单个 **partition** 分片

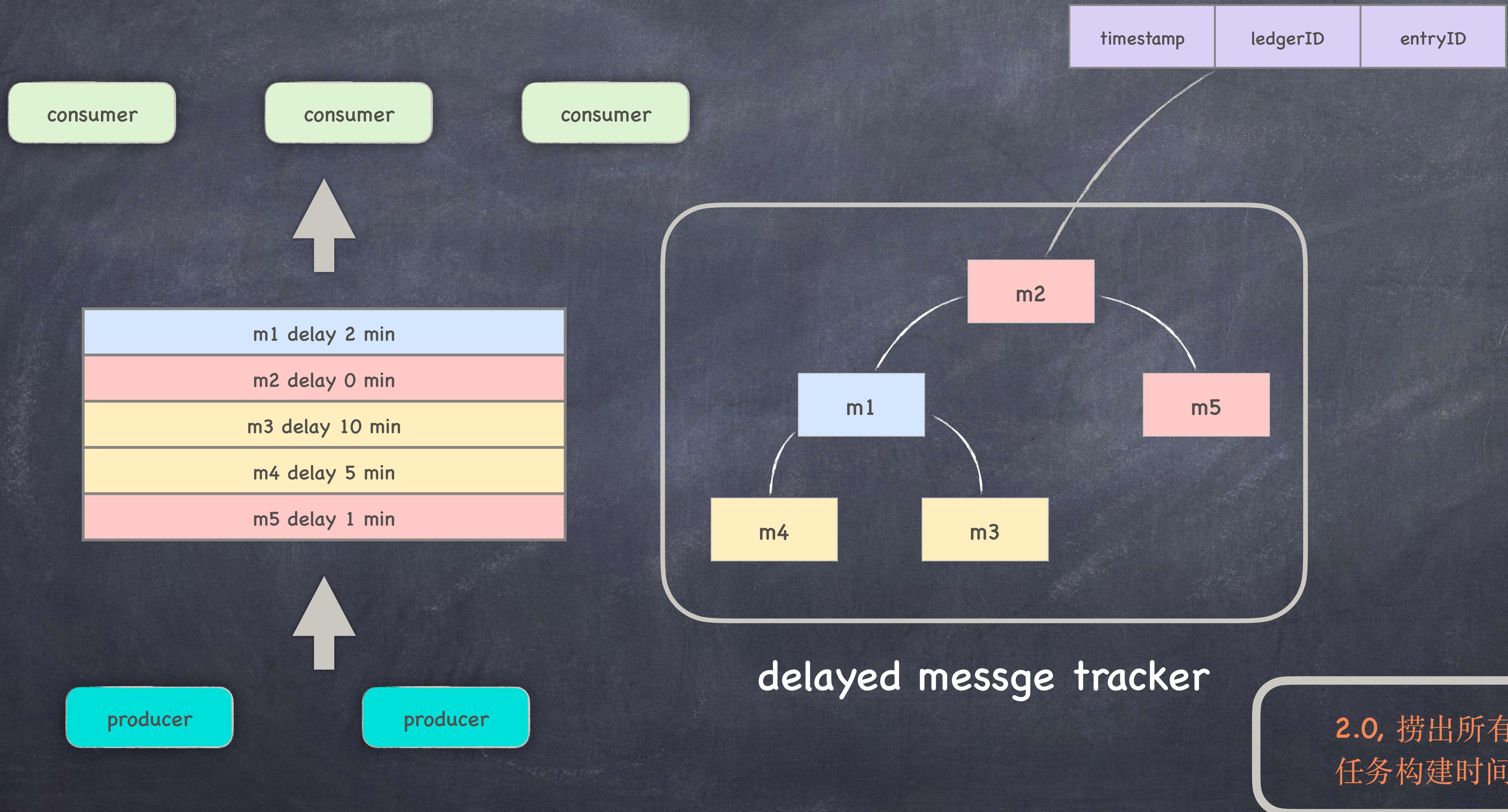


# delay queue

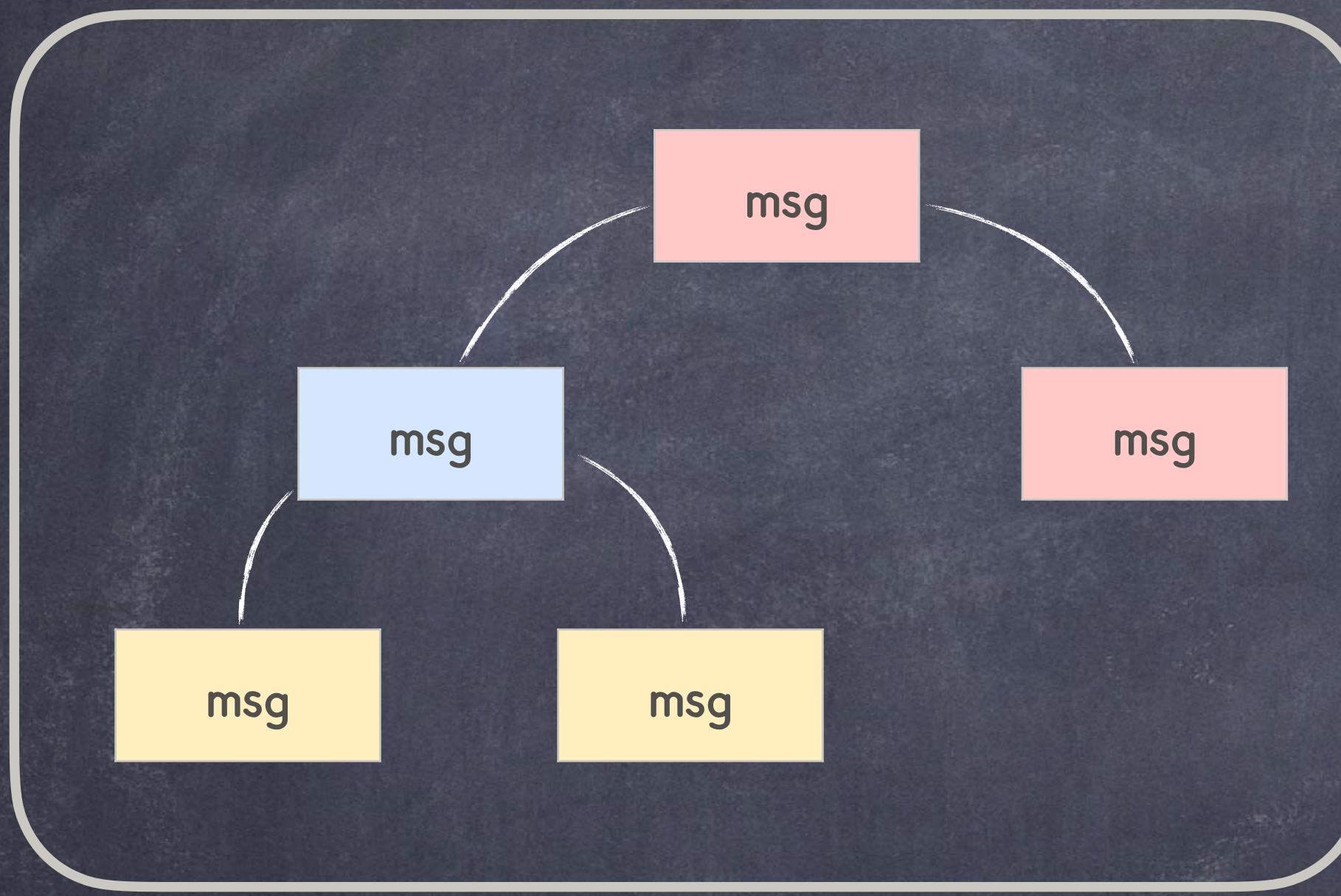


Rocketmq 延迟队列的实现方式

# delay queue



# delay queue



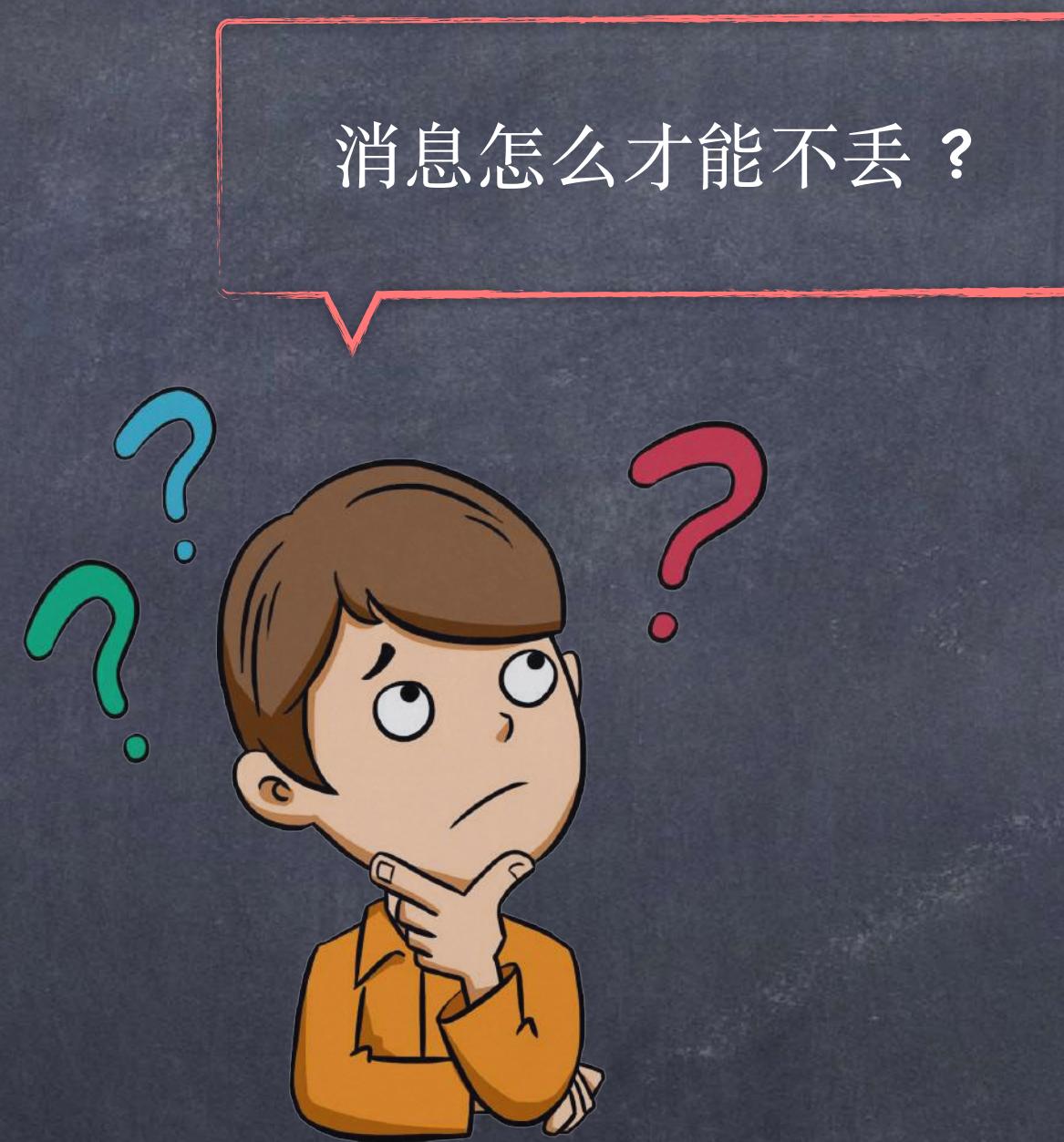
min heap

- \* delayed index 队列受到内存限制
- \* delayed index 队列重建时间开销

2.0 不能支撑海量的定时消息, 但 3.0 可以 !!!



# 消息的可靠性



- \* producer
- \* sync mode
- \* pulsar broker
  - \* 多副本
- \* pulsar bookie
- \* journal 同步策略
- \* consumer
  - \* 改为手动提交

thanks

hh, 感谢好友 小龙（pulsar pmc 成员）的解惑！

<https://github.com/wolfstudy>



“ Q&A ”

- [xiaorui.cc](http://xiaorui.cc)