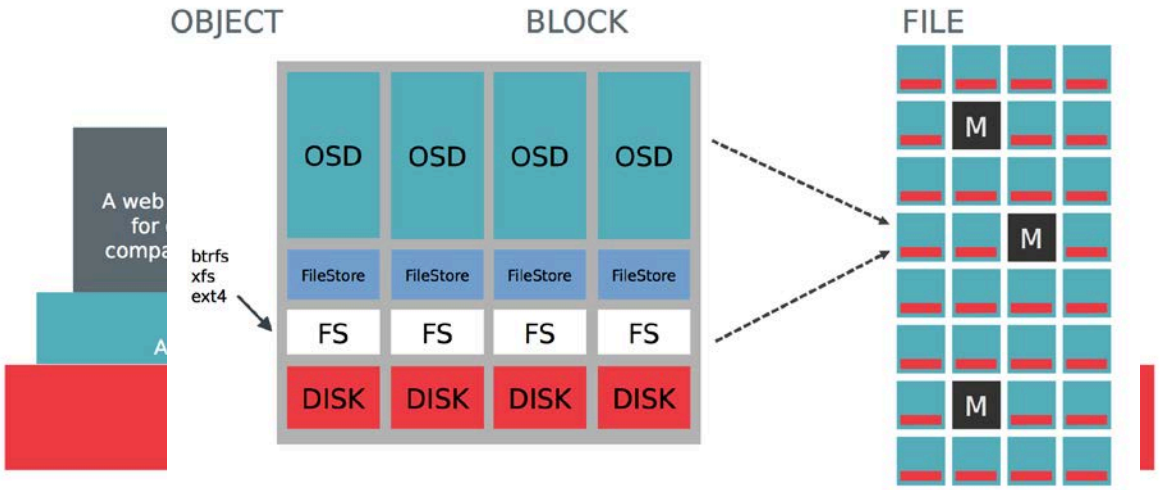
CEPH中的写优化系统

2016-11-1

易怀杰

MG1533078

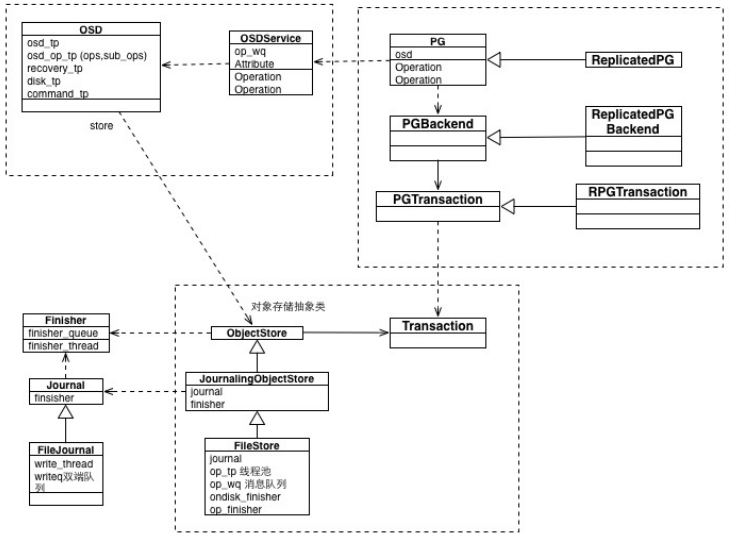
背景

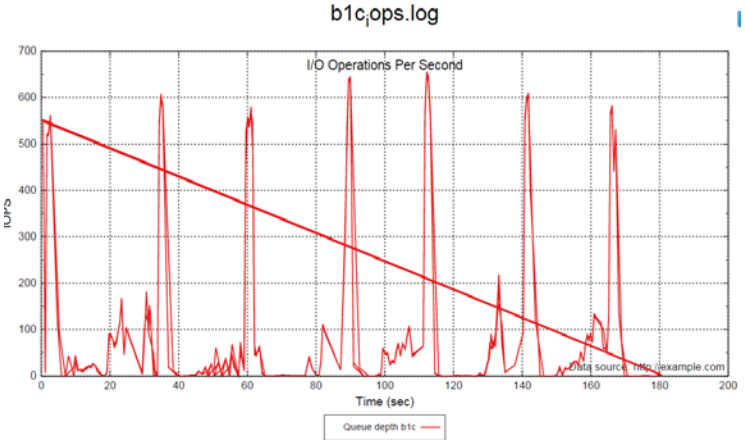


- CEPH
  1. 同一个集群
  2. 所有的组件
  3. 无单点故障
  4. 运行在异构
  5. 尽可能的自治。
- POSIX文件系统的问题
  1. 不合理的元数据管理——小文件的随机写性能极差。
  2. 基于POSIX的WAL机制让磁盘I/O利用率下降了一半。

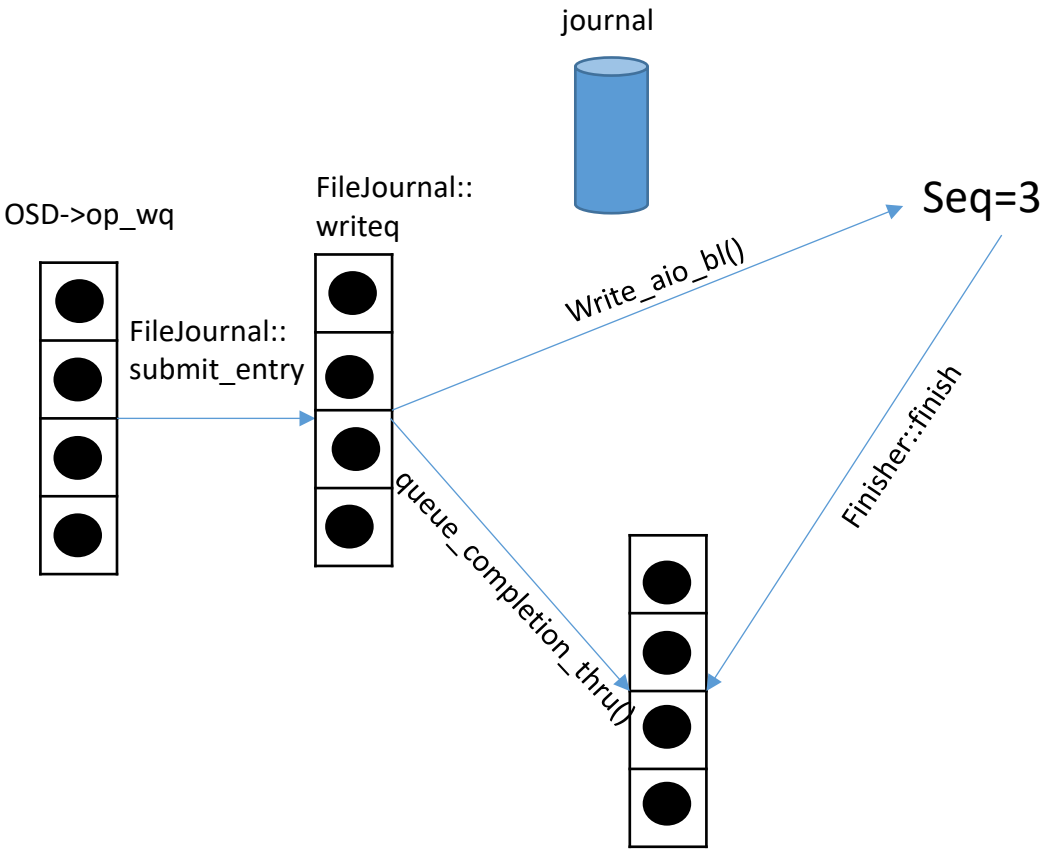
背景[2]

- FileStore的I/O抖动问题
  - 3个OSD
  - RBD+fio
  - SSD做journal





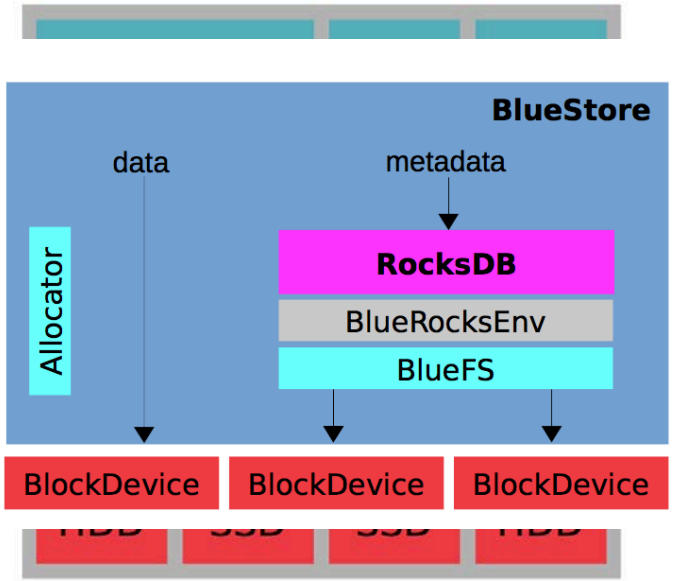
I/O抖动的原因



- 原因
  1. WAL的二次写
  2. journal落盘的机制不对

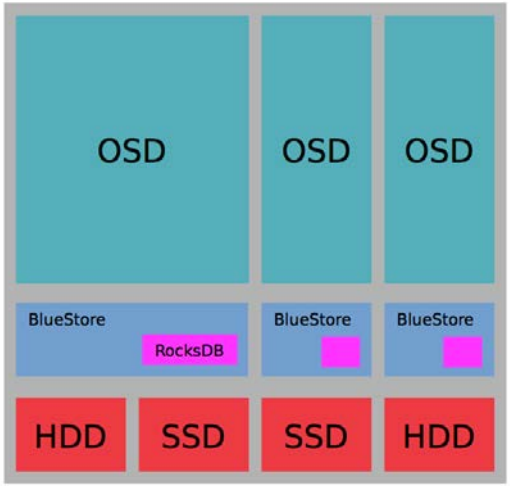
解决方案

- 思路
  - 采用写优化的系统（KV存储）将随机写变成顺序写。
  - 使用基于间接索引的COW日志替代WAL日志。
- 社区方案——BlueStore
  - NewStore=RocksDB+对象文件
  - BlueStore=块设备上的NewStore
  - 使用XFS/ext4来保存文件。
    - 对象数据（object data）
  - 用RocksDB来维护元数据（WAL）
    - 对象元数据（onode）
    - ceph的KV omap数据
    - 数据在磁盘上的映射元数据。



TokuDB

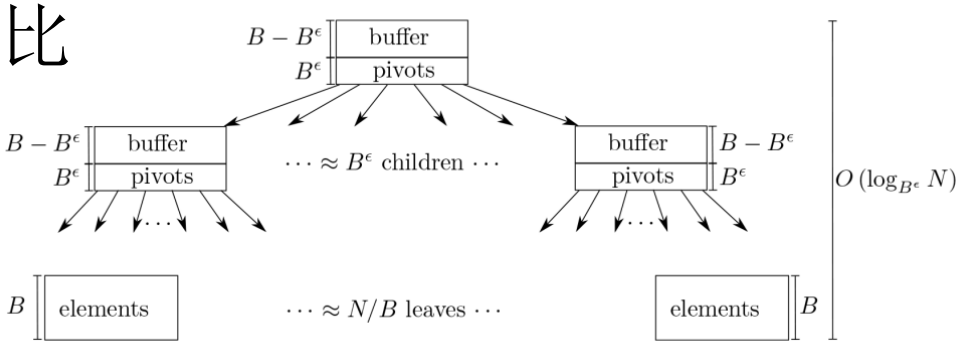
- BlueStore可以使用任何Kv存储管理元数据
- TokuDB对比RocksDB
  1. RocksDB——LSM Tree
  2. TokuDB——B$^{\varepsilon}$-Tree
  3. B$^{\varepsilon}$-Tree查询性能优于LSM Tree



Data Structure	Insert	Point Query		Range Query
		no Upserts	w/ Upserts	
B$^{\varepsilon}$-tree	$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$	$\frac{\log_B N}{\epsilon}$	$\frac{\log_B N}{\epsilon}$	$\frac{\log_B N}{\epsilon} + \frac{k}{B}$
B$^{\varepsilon}$-tree ($\epsilon = 1/2$)	$\frac{\log_B N}{\sqrt{B}}$	$\log_B N$	$\log_B N$	$\log_B N + \frac{k}{B}$
B-tree	$\log_B N$	$\log_B N$	$\log_B N$	$\log_B N + \frac{k}{B}$
LSM	$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$	$\frac{\log_B^2 N}{\epsilon}$	$\frac{\log_B^2 N}{\epsilon}$	$\frac{\log_B^2 N}{\epsilon} + \frac{k}{B}$
LSM+BF	$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$	$\log_B N$	$\frac{\log_B^2 N}{\epsilon}$	$\frac{\log_B^2 N}{\epsilon} + \frac{k}{B}$

B^{ε}-Tree

- 每一个节点包含多个磁盘块
- 每个节点被分为两部分
  - Pivots：指向子节点的指针。
  - Buffer：缓存更新和写入消息。
- Buffer是在磁盘上的
- **ε**参数——Pivots的百分比

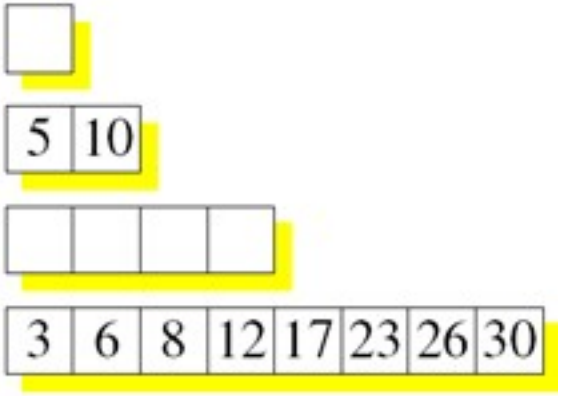


B^{ε}-Tree操作

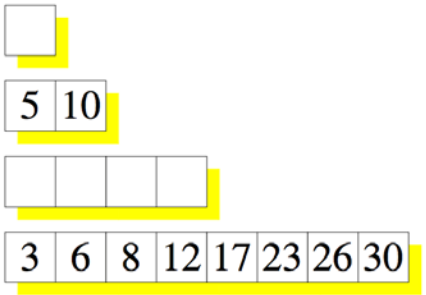
- 插入/更新
  1. 构建一个insert消息
  2. 插入到B^{ε}-Tree的根节点的Buffer中，写入成功返回。
  3. 如果节点的Buffer满了，则将消息根据键的值写入对应的子节点的Buffer中
  4. 重复第3步直到写入叶子节点。
- 删除
  1. 在根节点插入tombstone消息
  2. Buffer满后分发到子节点，直到到达叶子节点
  3. 删除叶子节点对应的KV，丢弃该消息
- 节点的分裂与合并——与B^{+}树类似
- 点查询和范围查询
  - 点查询操作与B^{+}树类似，性能也相差无几
  - 范围查询由多个点查询构成，性能比B^{+}树差
    - B^{+}树有一个指向兄弟节点的指针

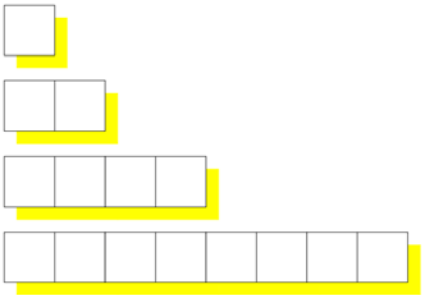
分形树（Fractal Tree）

- 分形树是TokuDB中对B^{ε}-Tree的一个开源实现
- 数据结构（简化版）
  - 相邻行空间加倍
  - 每一行要么全满要么全空
  - 全满行的数据都是排好序的



- 辅助数组
  - 辅助插入操作
  - 让插入和合并由两个独立的线程操作



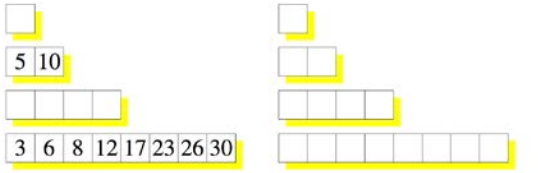


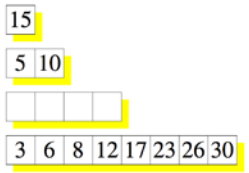
分形树的写入与合并

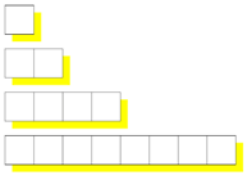
- 写入线程
  - 写入第一级（根节点）
  - 如果主数组空，写入主数组否则写入辅助数组
  - 主数组与辅助数组必有一个为空
- 合并线程
  - 当某一级的主辅数组都满的时候，将其合并写入下一级
  - 写入下一级的时候如果下一级的主数组为空，写入主数组，否则写入辅数组
  - 直到没有任何一级主辅数组都满

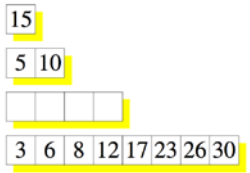
分形树的写入与合并

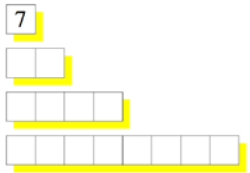

islab
distributed computing laboratory

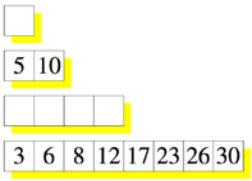

5 10
3 6 8 12 17 23 26 30

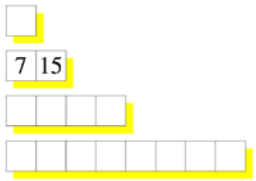

15
5 10
3 6 8 12 17 23 26 30

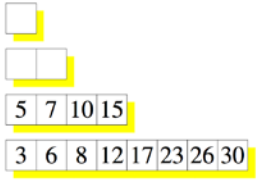


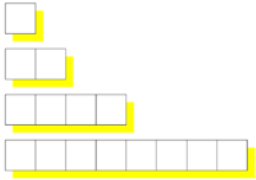

15
5 10
3 6 8 12 17 23 26 30

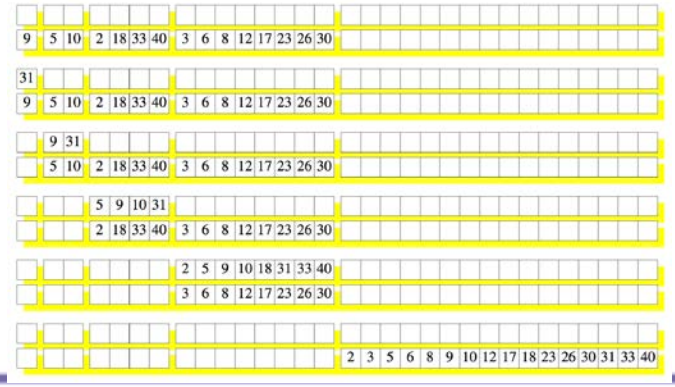

7


5 10
3 6 8 12 17 23 26 30


7 15

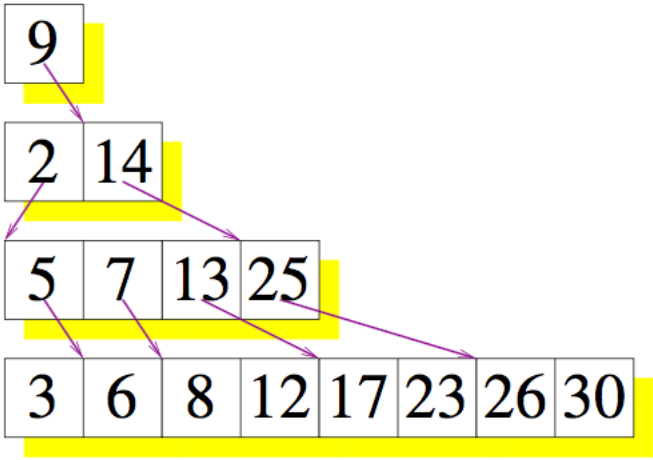

5 7 10 15
3 6 8 12 17 23 26 30




9 5 10 2 18 33 40 3 6 8 12 17 23 26 30
31
9 5 10 2 18 33 40 3 6 8 12 17 23 26 30
9 31
5 10 2 18 33 40 3 6 8 12 17 23 26 30
5 9 10 31
2 18 33 40 3 6 8 12 17 23 26 30
2 5 9 10 18 31 33 40
3 6 8 12 17 23 26 30
2 3 5 6 8 9 10 12 17 18 23 26 30 31 33 40

简单分形树的性能

- 插入
  - O(1)
- 合并
  - 合并两个大小为X的数组O(X/B)
  - 每一个元素的代价是O(1/B)
  - 每一个元素的合并代价是O(logN)
  - 平均插入代价是O((logN)/B)
- 查询性能是$O(\log^2 N)$
  - 与LSM-Tree相同
  - 加入前向指针



- 前向指针
  - 每一个元素都有一个指向下一级中第一个键值大于本身的元素
  - 同一级采用二分查找
  - 查询性能为O(logN)

分形树的实现


```
158 // TODO: class me up
159 struct ftnode {
160     MSN      max_msn_applied_to_node_on_disk; // max_msn_applied that will be written to disk
161     unsigned i
162     BLOCKNUM b
163     int    lay
164     int    lay
165     int    lay
166     uint32_t b
167     int    hei
168     int    dir
169     uint32_t f
170
171     // for int
172     // for lea
173     int n_chil
174     ftnode_piv
```

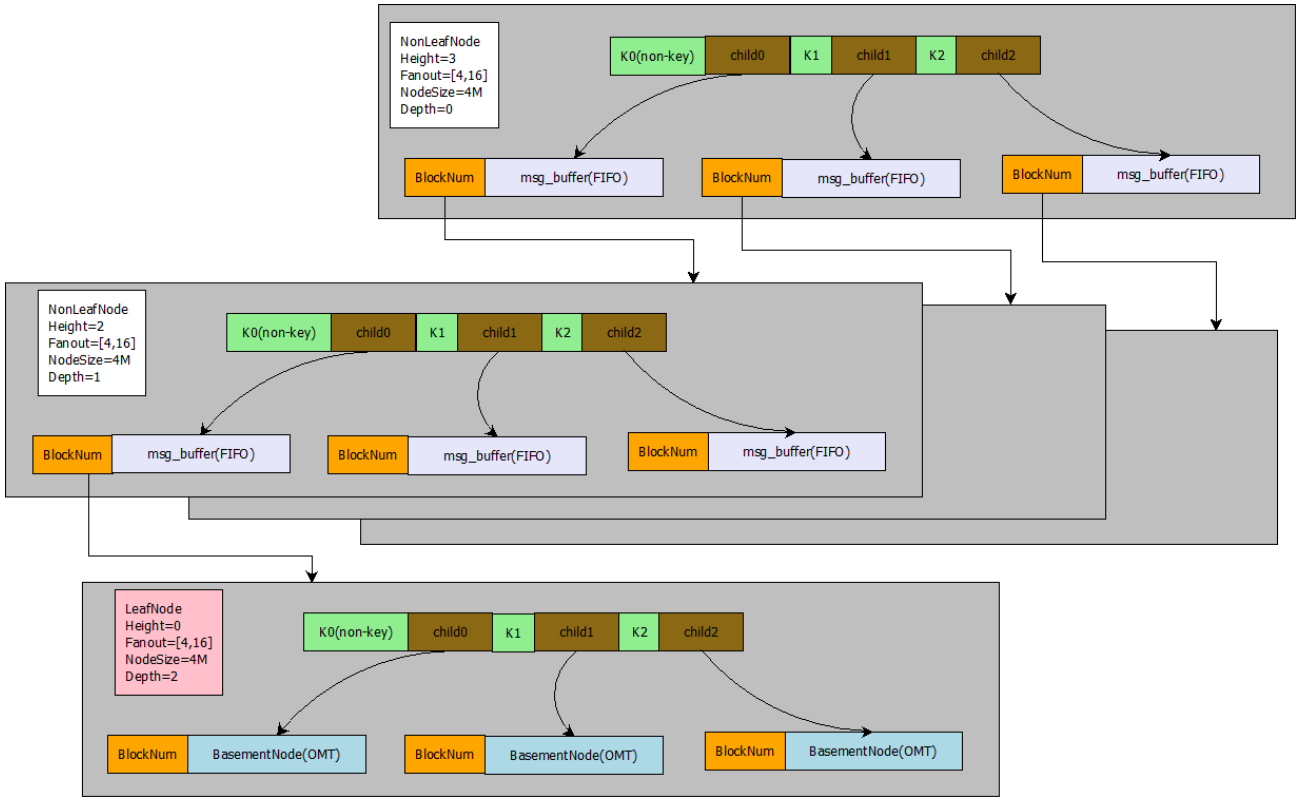
```
45  /* Pivot keys.
46   * Child 0's keys are <= pivotkeys[0].
47   * Child 1's keys are <= pivotkeys[1].
48   * Child 1's keys are > pivotkeys[0].
49   * etc
50   */
51  class ftnode_pivot_keys {
52  public:
53      // effect: create an empty set of pivot keys
54
55
56
57
58
```

```
233 typedef struct ftnode_child_pointer {
234     union {
235         struct sub_block *subblock;
```

```
223 // data of an available partition of a nonleaf ftnode
224 struct ftnode_nonleaf_childinfo {
225     message_buffer msg_buffer;
226     off_omt_t broadcast_list;
227     marked_off_omt_t fresh_message_tree;
228     off_omt_t stale_message_tree;
229     uint64_t flow[2];  // current and last checkpo
230 };
231 typedef struct ftnode_nonleaf_childinfo *NONLEAF_CI
232
```

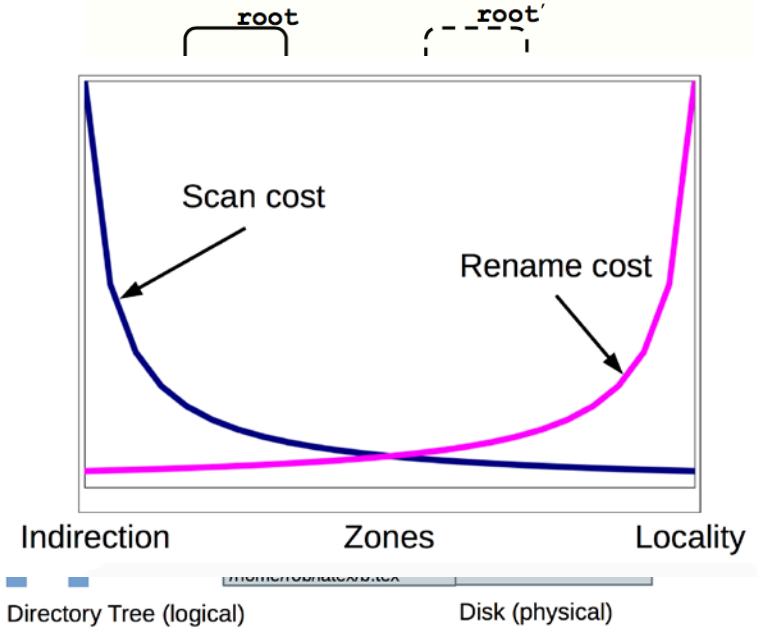
```
195 // data of an available partition of a leaf ftnode
196 struct ftnode_leaf_basement_node {
197     bn_data data_buffer;
198     unsigned int seqinsert;         // number of sequential inserts to this leaf
199     MSN max_msn_applied;            // max message sequence number applied
200     bool stale_ancestor_messages_applied;
201     STAT64INFO_S stat64_delta;      // change in stat64 counters since basement was last written
202     int64_t logical_rows_delta;
203 };
204 typedef struct ftnode_leaf_basement_node *BASEMENTNODE;
```

分形树的实现


NonLeafNode
Height=3
Fanout=[4,16]
NodeSize=4M
Depth=0
K0(non-key)
child0
K1
child1
K2
child2
BlockNum
msg_buffer(FIFO)
BlockNum
msg_buffer(FIFO)
BlockNum
msg_buffer(FIFO)
NonLeafNode
Height=2
Fanout=[4,16]
NodeSize=4M
Depth=1
K0(non-key)
child0
K1
child1
K2
child2
BlockNum
msg_buffer(FIFO)
BlockNum
msg_buffer(FIFO)
BlockNum
msg_buffer(FIFO)
LeafNode
Height=0
Fanout=[4,16]
NodeSize=4M
Depth=2
K0(non-key)
child0
K1
child1
K2
child2
BlockNum
BasementNode(OMT)
BlockNum
BasementNode(OMT)
BlockNum
BasementNode(OMT)

BetrFS

- 使用分形树
- late-binding日志（BlueStore中的延迟写日志？）
- 分区（zoning）
- Range Deletion
  - 删除消息的广播



未来工作

- 用TokuDB替代RocksDB
- 测试性能，TokuDB和RocksDB的性能对比
- 引入late-binding日志和分区

谢谢！

Q&A