6
LSM Storage Model
in comprison to Heap & BTREE storage

Reading: [Me] chapter 7
[Ha], chapter 10, p. 158 ff
[Kl], chapter 3
[Pe], chapter 7

2 3 1

LSMT - Log Structured Merge Tree Storage

NoSQL Databases

NoSQL CATEGORY	EXAMPLE DATABASES	DEVELOPER
Key-value database	Dynamo Riak Redis Voldemort	Amazon Basho Redis Labs LinkedIn
Document databases	MongoDB CouchDB OrientDB RavenDB	MongoDB, Inc. Apache OrientDB Ltd. Hibernating Rhinos
Column-oriented databases	HBase Cassandra Hypertable	Apache Apache (originally Facebook) Hypertable, Inc.
Graph databases	Neo4J ArangoDB GraphBase	Neo4j ArangoDB, LLC FactNexus

Source: Coronel,Morris, Data Base Systems, ch. 14

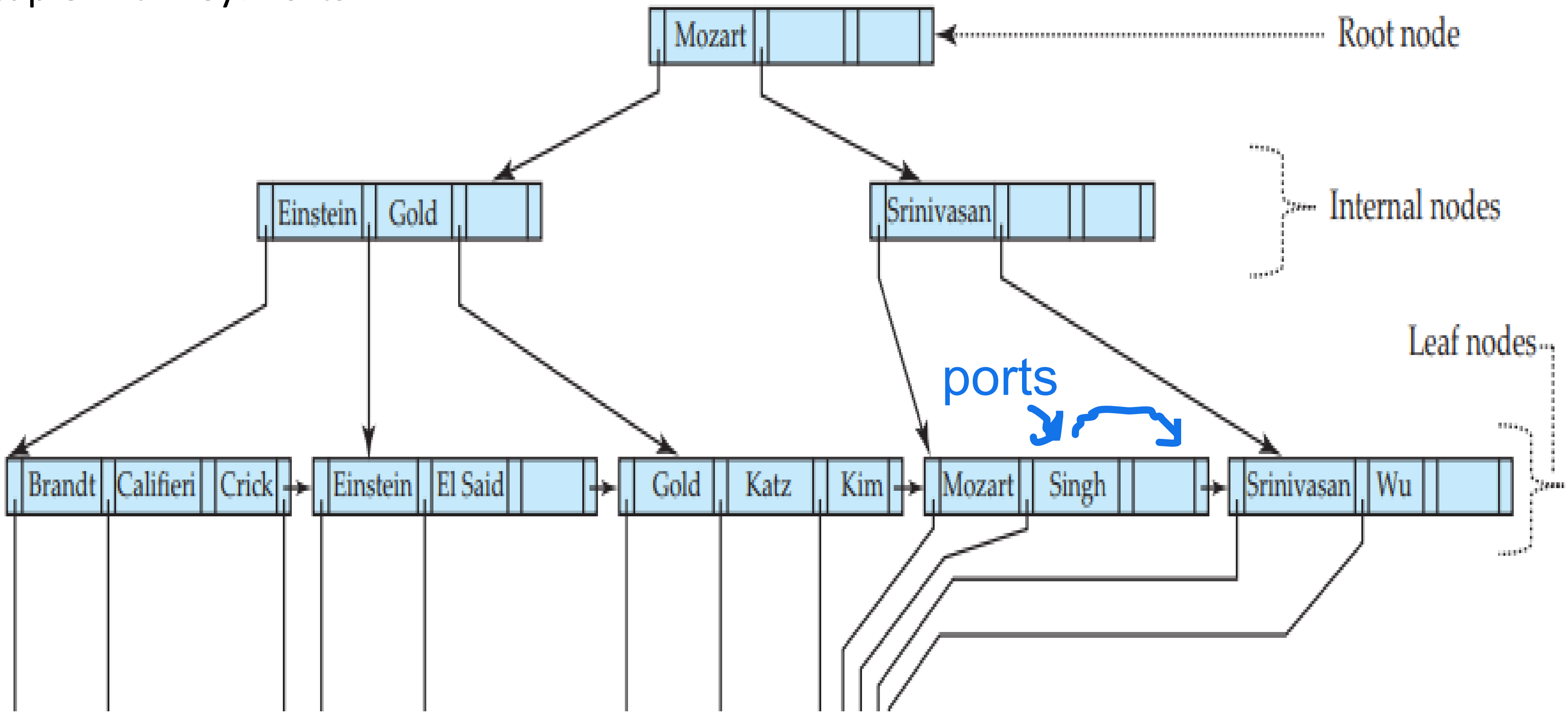
B+ Tree Storage Model

Advantages (properties) of the B+ Tree storage model:

balanced, sorted, page-oriented optimized for disk I/O(btree node always relates to one page in buffer so that when transfering from btree to buffer it's always one node)

B+ Tree Storage Model

Insert of tuple with key: Ports



[SI], p. 488

Heap File / B+ Tree Storage Model

Insert of tuple with key: Ports

→ traverse PK BTREE → locate correct leaf node → load leaf into buffer
1 random disk access, assuming that root node and internal nodes of PK BTREE are in buffer cache.

→ with the help of the BTREE pointer, locate heap file page → load heap file page into buffer
1 random disk access (heap file is fragmented)

→ insert tuple into heap file page and insert PK correctly into BTREE leaf node

→ write insert into WAL and flush WAL to disk
sequential write

→ write modified heap file page back to disk
1 random disk access

→ write modified PK BTREE leaf back to disk
1 random disk access

Heap-File reading is fast since data can be accessed directly or scanned, but writing may require searching the whole file or reorganizing data. B-Tree reads are efficient due to the sorted structure enabling fast lookups which takes logarithmic time but writes are slower because they may require node splitting, merging or rebalancing to maintain tree properties.

For each further secondary index, add two more random disk access.

this is faster

On spinning disks (HDDS), there is mechnical movement. SSDs erase the page and then re-write.

Heap-File & BTREE storage supprts read operations better than write operations

Random I/O versus Sequential I/O (simplified)

1000 blocks of 4KB each are to be read:

Random I/O:

Position arm each time (seek)

Latency time each time (wait)

→ 1000 * (5 ms + 3 ms)

→ 8000 ms → 8s + transfer time

slow because we repeat steps 1000 times

Sequential I/O (actually: Chained I/O)

Position once, then “scrape off the disk”

→ 5 ms + 3 ms

→ 8ms + transfer time

all blocks are next to each other on disk, only 1 seek and latency needed at the beginning and then disk reads all blocks in one go(scraping). So this is much faster.

Btree model relies on random writes which makes it very slow

Heap-File / B-Tree Storage

Bottleneck is the write-throughput (velocity)

slow:

- random writes on disk
- writes-in-place
  always have to go to right place and overwrite

Read operations are supported by indexes.

LSM Storage (Log-Structured-Merge TREE Storage)

Concept:

only sequental writes on disk

no writes-in-place so files on disk immutable

log

Tree: sorted keys

sorting is always time-consuming. if we do this in memory and not disk then it's completely fine. Once it's sorted then we flush into disk and everything becomes immutable. little bit change is made with merge which is last component of LSMT. It does not change logs individually, it just merges old logs into a new log

Log-Structured Merge Trees

Widely adopted because they balance read performance and ingestion

eva.knirsch@kiu.edu.ge

LSM-Storage

- Firt described by Patrick O'Neill, et.all in 1996
  https://www.cs.umb.edu/~poneil/lsmtree.pdf

  Implementation today differ from the description in the paper but concept stayed the same

- LSM-tree concept comprises 2 different data structures:
  - First data structure - which is entirely memory resident, Tree
  - Second data structure - which is resident on disk Log

- Compare this to BTREEs: same data structure in memory and on disk.

LSM-Storage Components

1. Memory Structure:
   - memtable - usually a tree, e.g. red-black-binary search tree
   - sorts incoming random write operations according to the key

2. Disk Structure:
   - Sorted String tables (SST file segments)
   - content of red-black-trees are flushed to disk sequentially into SSTs
   - SSTs are immutable

3. Compaction / Merge Process

eva.knirsch@kiu.edu.ge

LSM-Storage: 1. Memtable: Sorting Incoming Random Writes

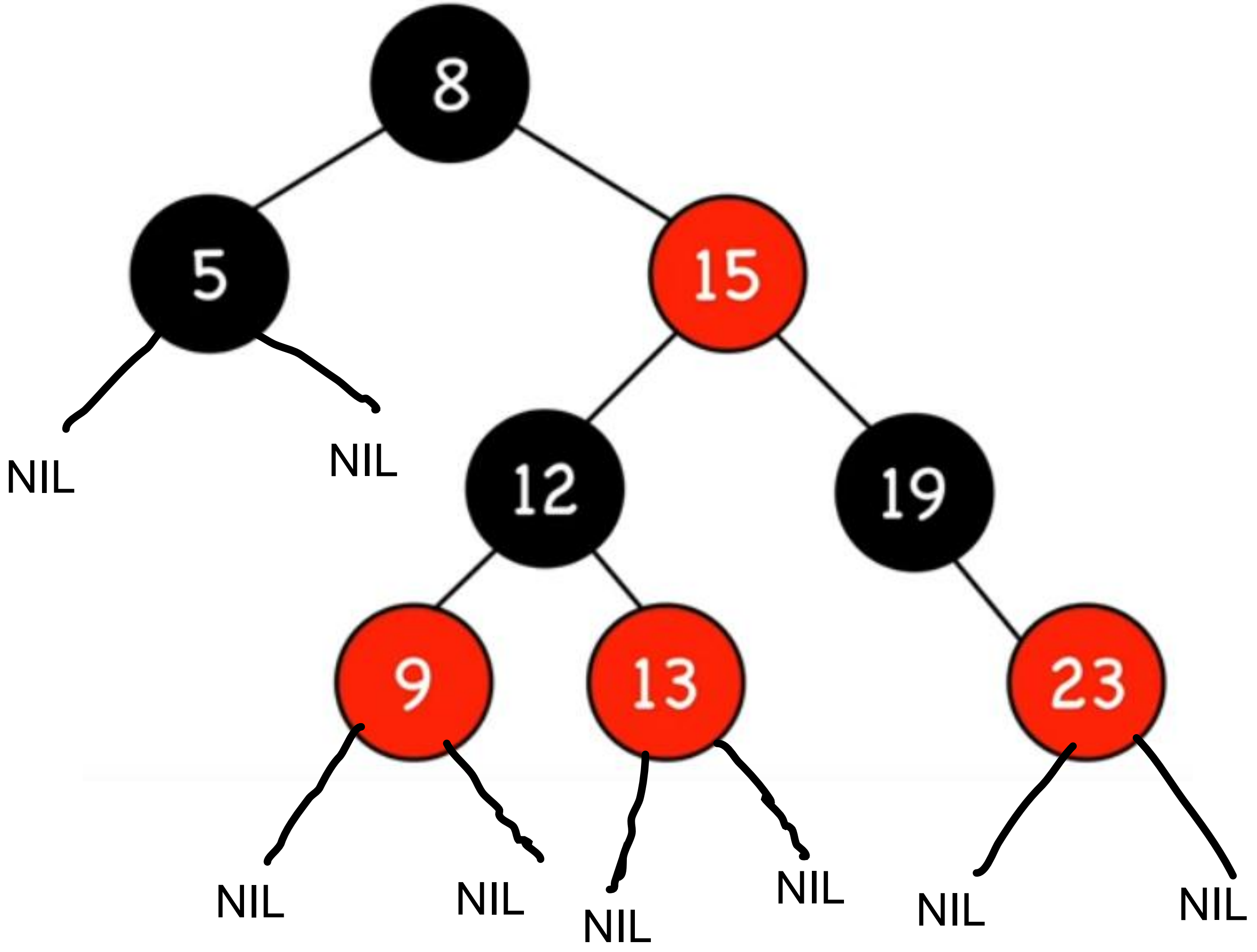
- Writes come in random and need to be sorted.
- Writes are sorted into an in-memory (only) tree-structure.
  → fast
- The tree structure usually is a self-balancing binary search tree (e.g. a red-black-binary tree).
- The depth (or height) of the tree that we are so concerned about in B-Trees is not an important aspect because
  - Tree only exists in memory
  - Tree does not get large
- memtable is Cassandra terminology. The term memtable does not tell the exact implementation. This could be any self-balancing tree-structure or even a B-tree or yet another structure used to sort random incoming writes.

Red-Black-Tree as memtable

- Red-Black-Trees were first described in 1972 by Rudolf Bayer, TUM professor.
- Red-Black-Trees are binary search trees.

Properties:

1. Each node is either red or black. Nodes can change their color.
2. Root and leaves (NIL-nodes) = black (NIL nodes usually are not drawn.)
3. !RR ("Not Red-Red").
4. Each path from a given node to leaves (path ends) contains the same number of black nodes.



Insertion Algorithm Red-Black-Tree

Let x be the node to insert

- Perform standard BST insertion and color newly inserted nodes as RED, if x is not the root.
- If x is the root, color x BLACK
- Do the following if the color of x parent is Red and x is not the root.

a) If x uncle is RED

(i) Recolor parent and uncle as BLACK.

(ii) Recolor grandparent as RED if grandparent is not root.

b) If x uncle is BLACK, then there can be four configurations for x, parent (p) and grandparent (gp)

(i) Left Left Case (line) rotate gp right (in the opposite direction of x),
recolor: former p (now root of subtree): black, former gp (now right child: red)

(ii) Left Right Case (triangle) – rotate parent left (in the opposite direction of x)

(iii) Right Right Case (line) - rotate gp left ((in the opposite direction of x),
recolor: former p (now root of subtree): black, former gp (now left child: red)

(iv) Right Left Case (triangle) – rotate parent right (in the opposite direction of x)

eva.knirsch@kiu.edu.ge

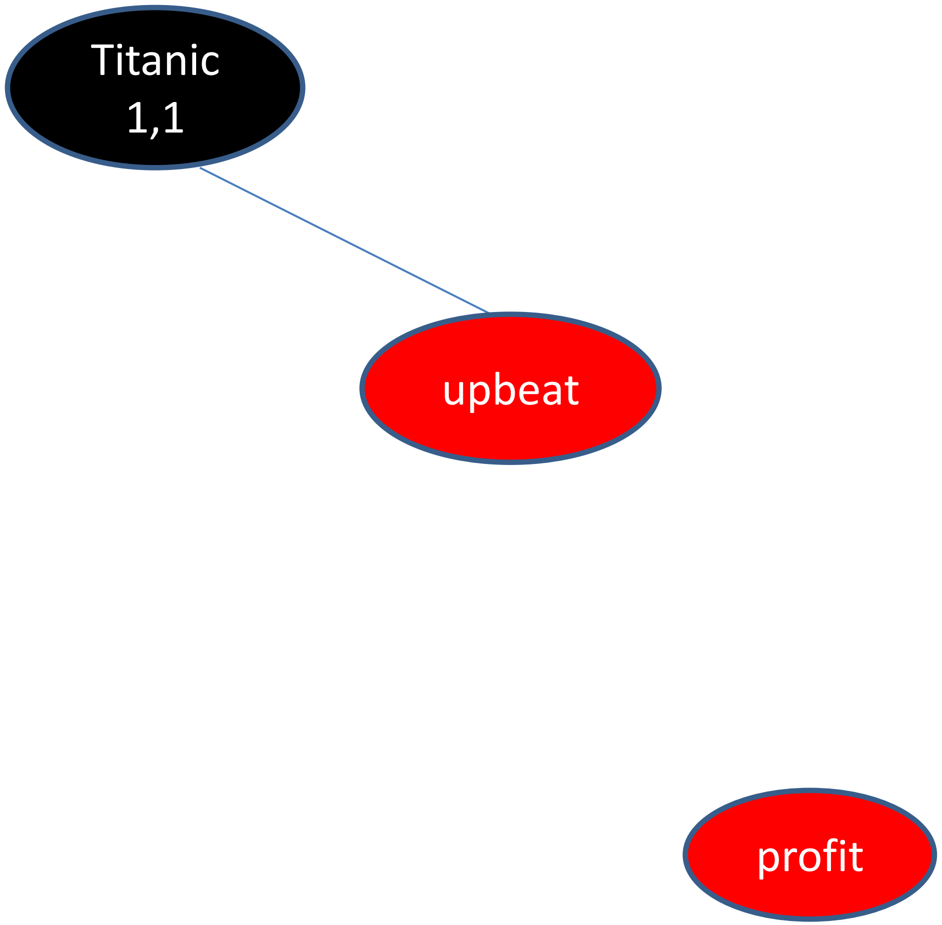
Red-Black-Tree as memtable: Incoming Writes

{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}

eva.knirsch@kiu.edu.ge

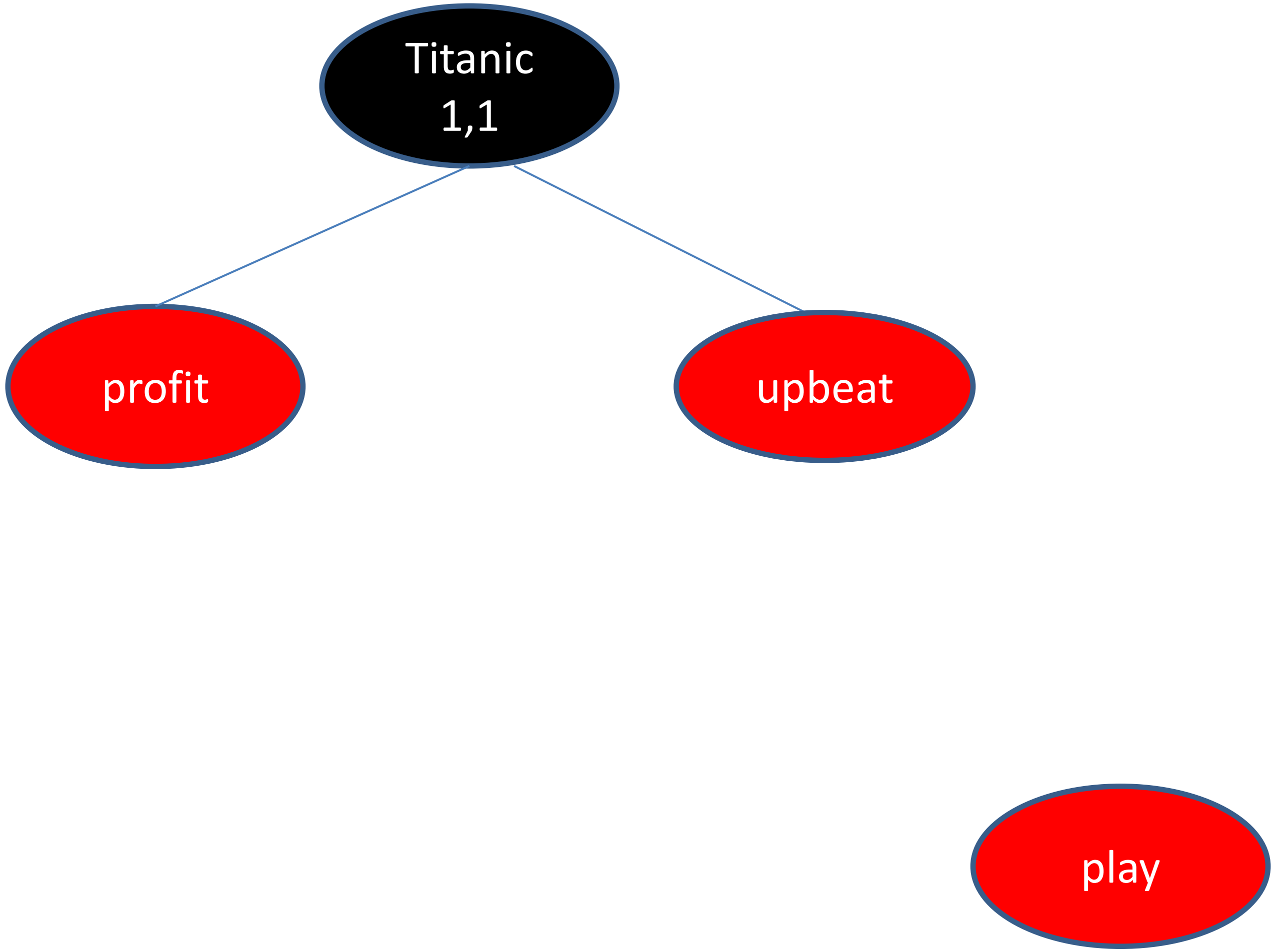
Red-Black-Tree as memtable: Incoming Writes

{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}



Red-Black-Tree as memtable: Incoming Writes

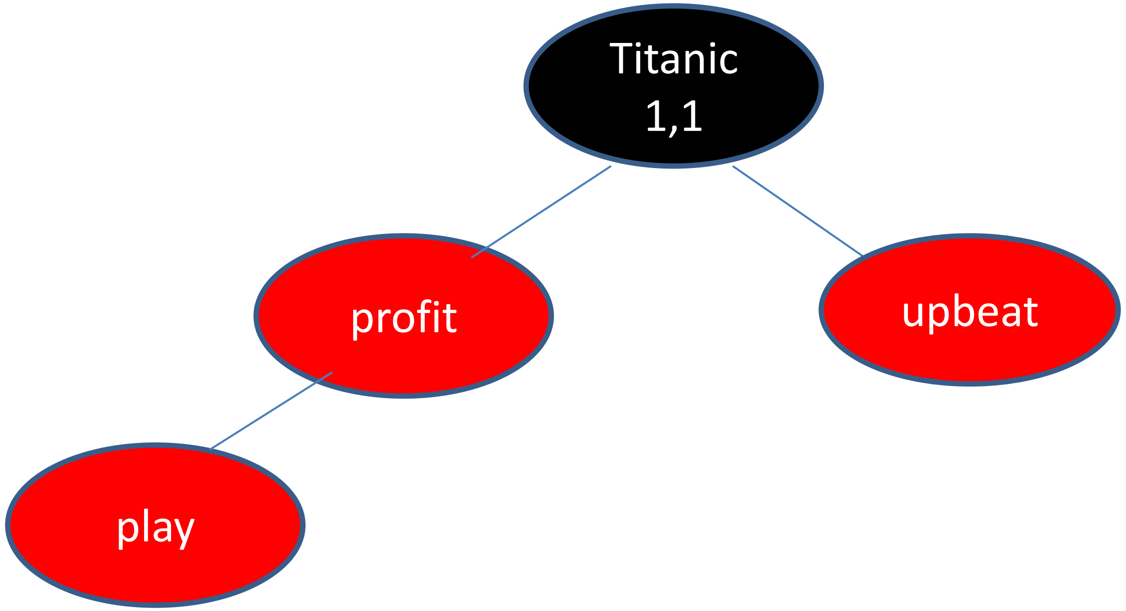
{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}



eva.knirsch@kiu.edu.ge

Red-Black-Tree as memtable: Incoming Writes

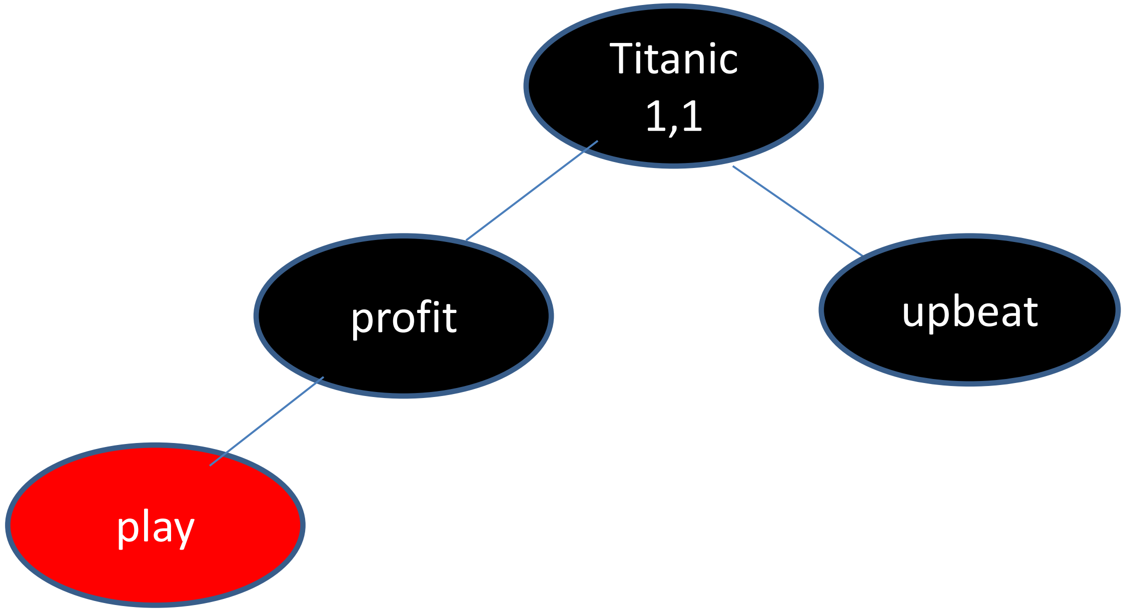
{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}



eva.knirsch@kiu.edu.ge

Red-Black-Tree as memtable: Incoming Writes

{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}

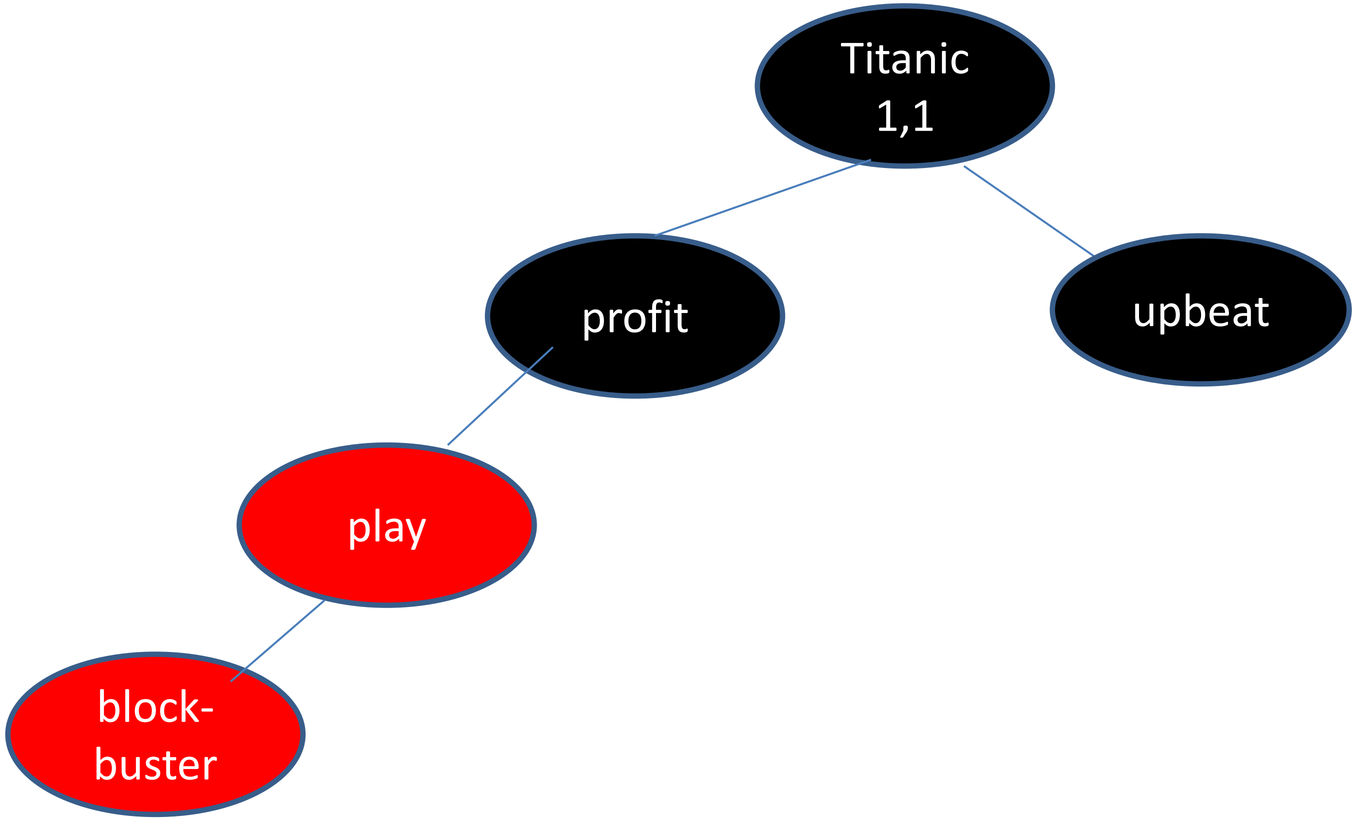


eva.knirsch@kiu.edu.ge

KUTAISI INTERNATIONAL UNIVERSITY

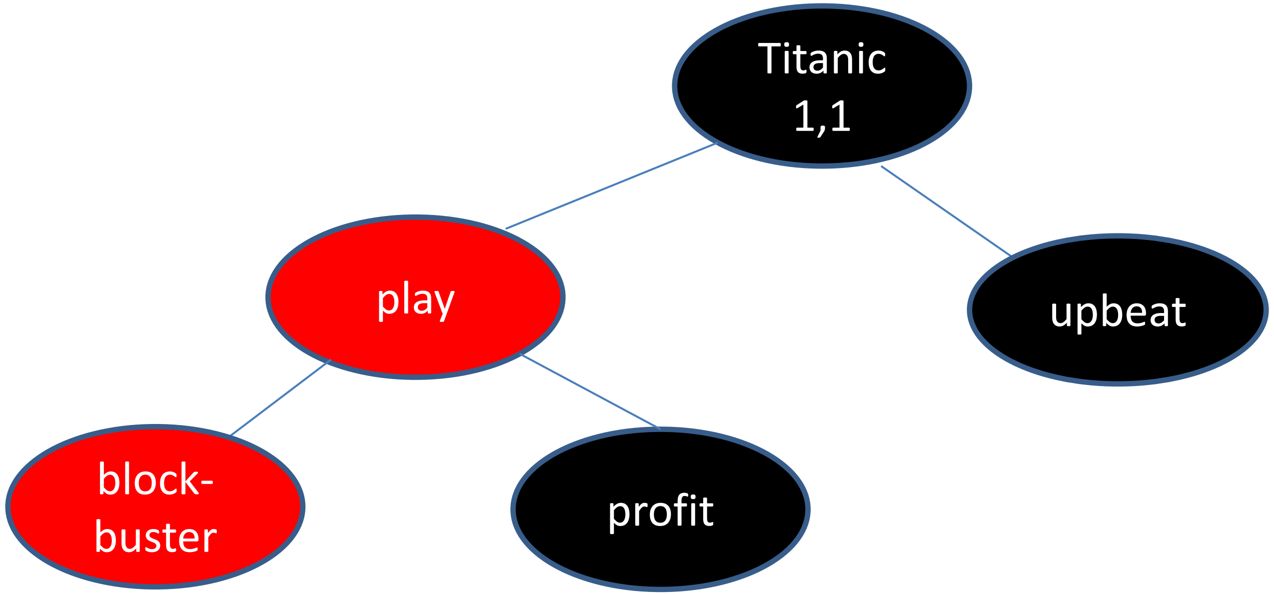
Red-Black-Tree as memtable: Incoming Writes

{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}



Red-Black-Tree as memtable: Incoming Writes

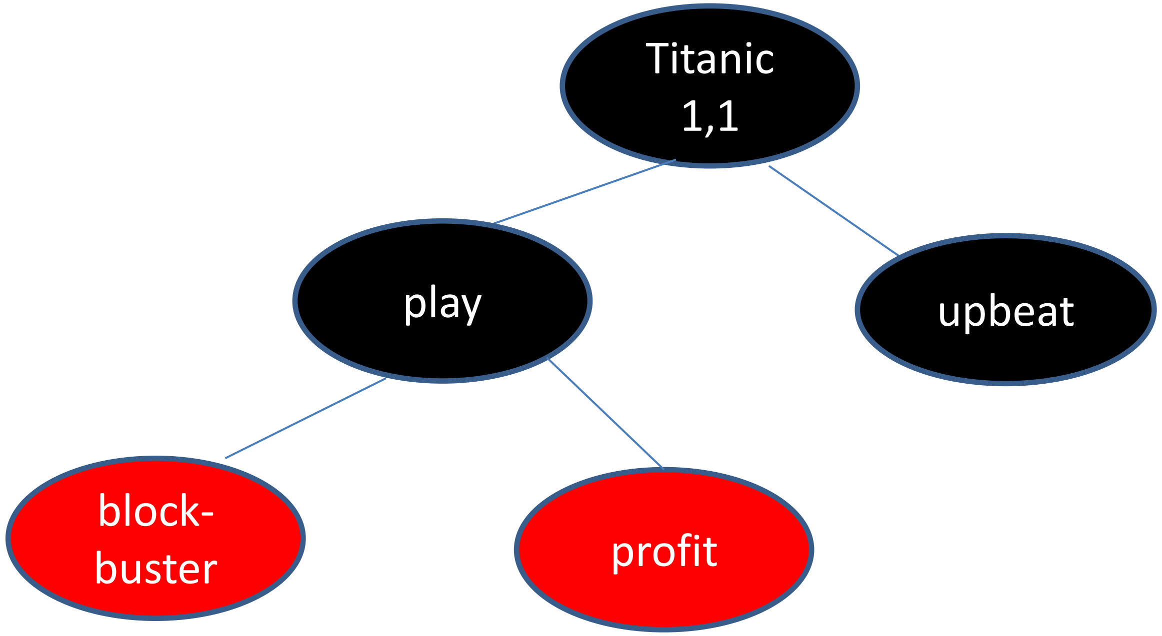
{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}



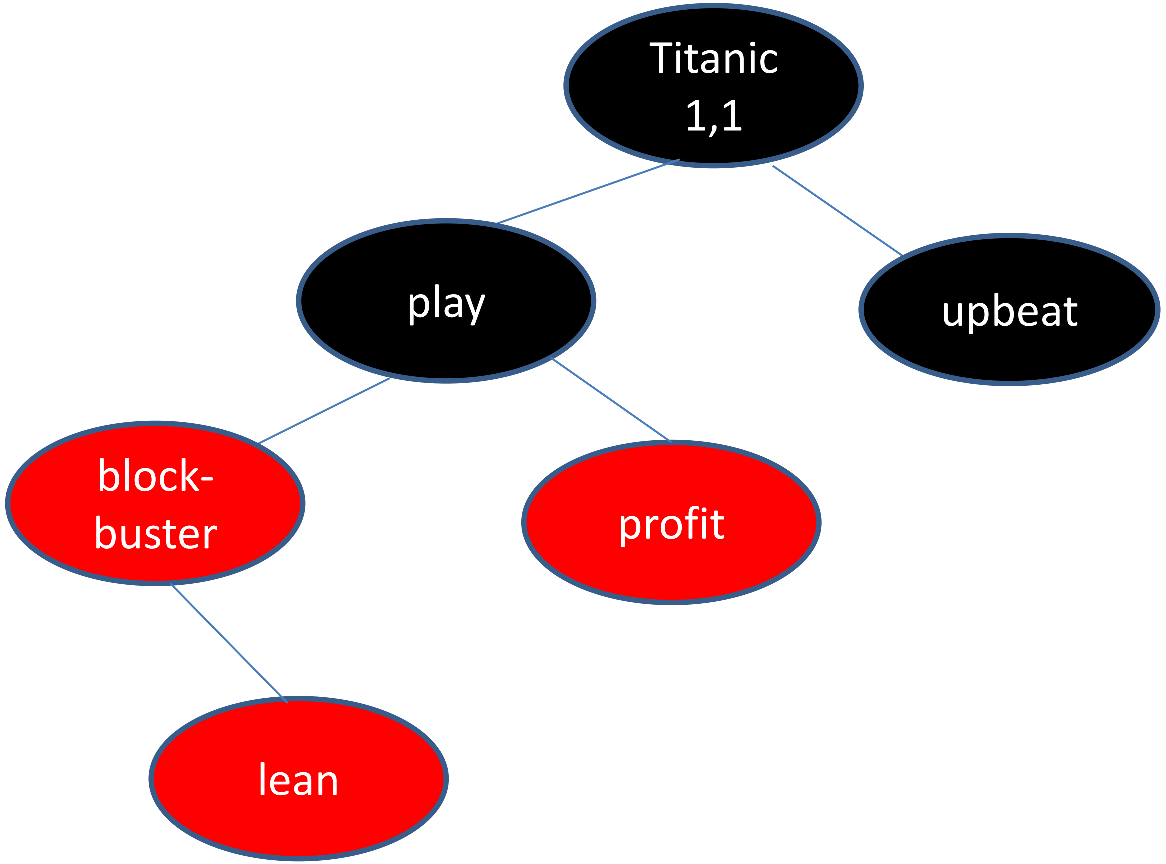
eva.knirsch@kiu.edu.ge

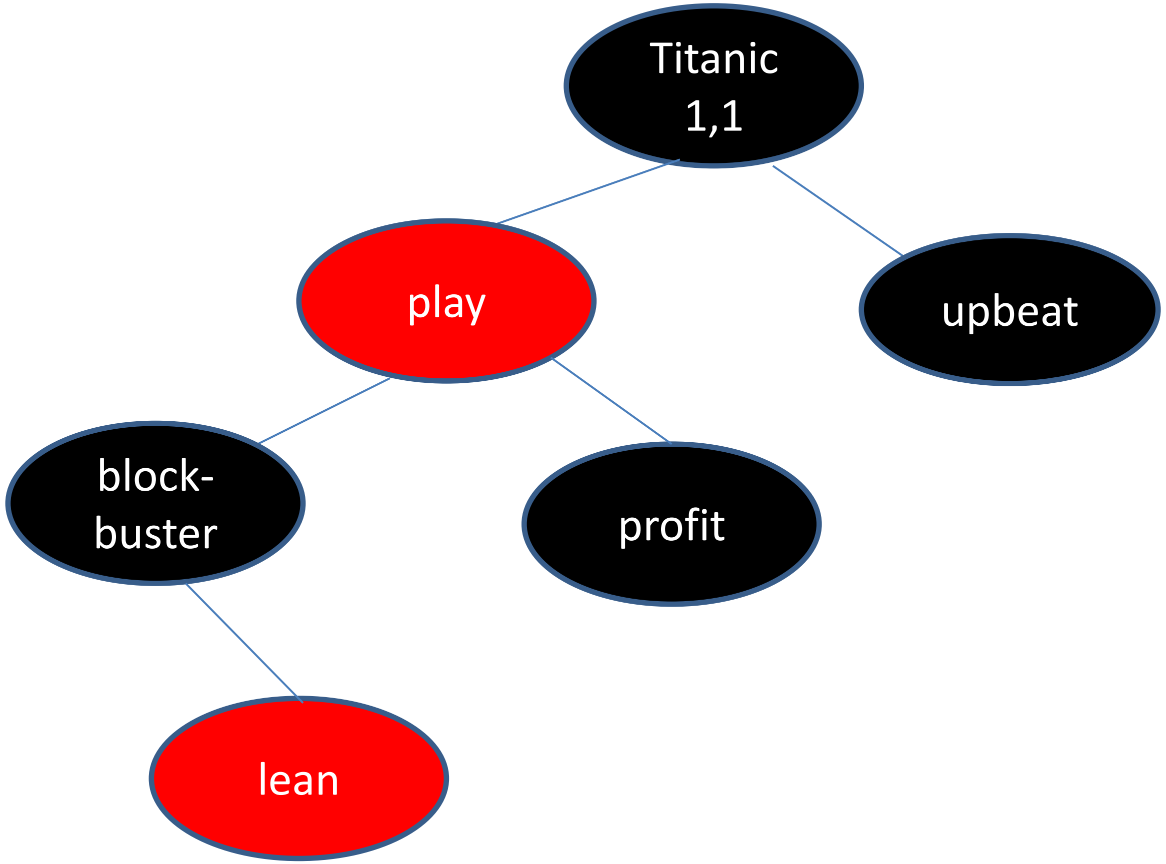
Red-Black-Tree as memtable: Incoming Writes

{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}

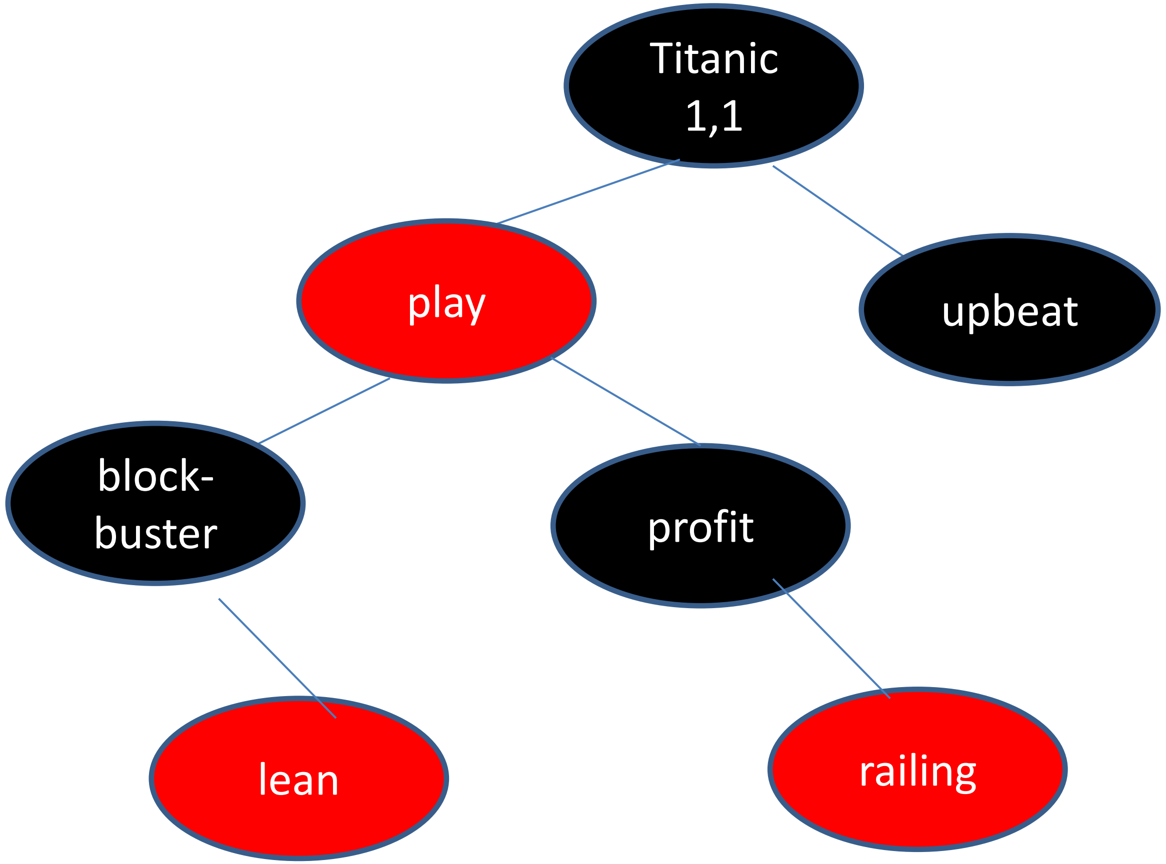


{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}



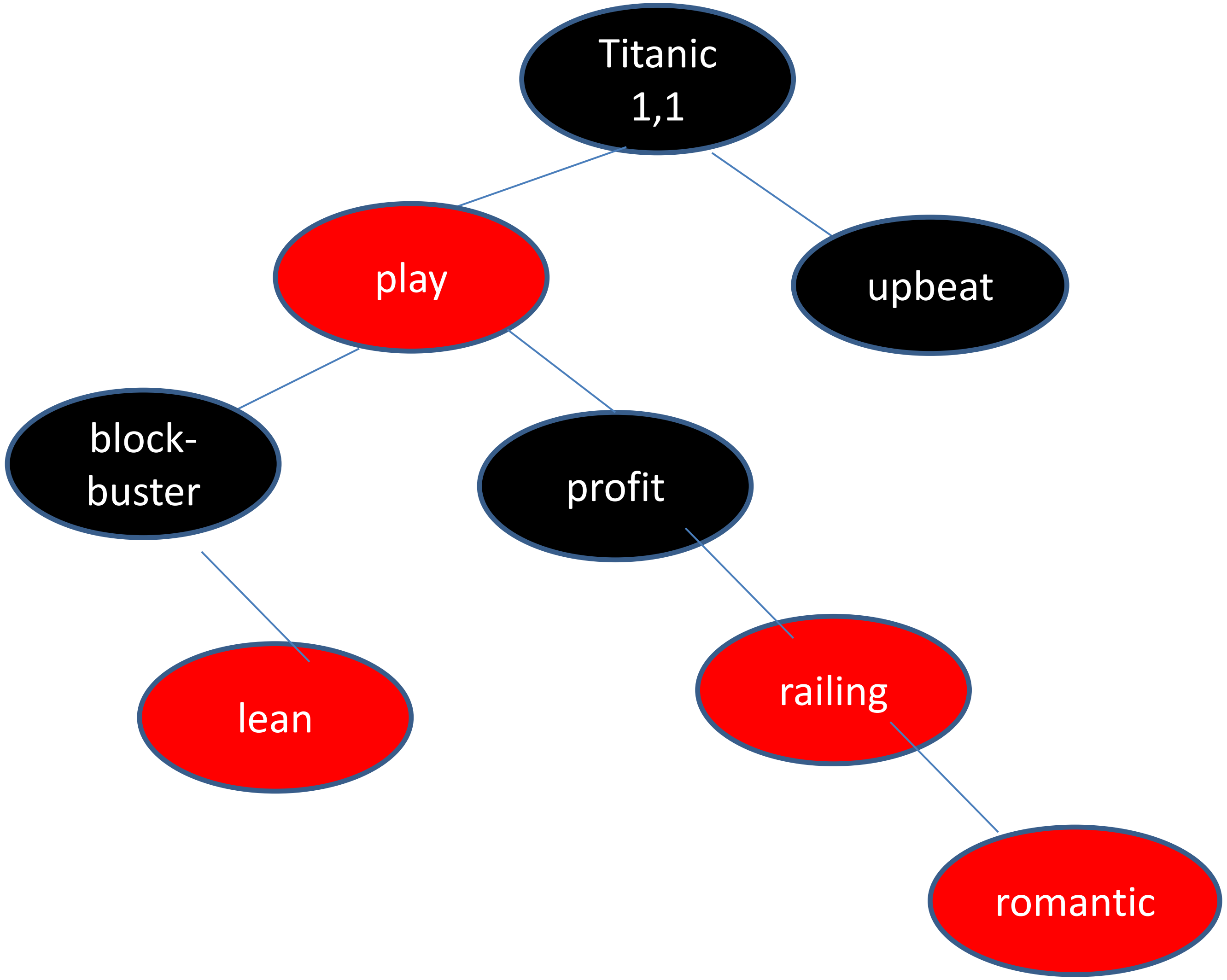


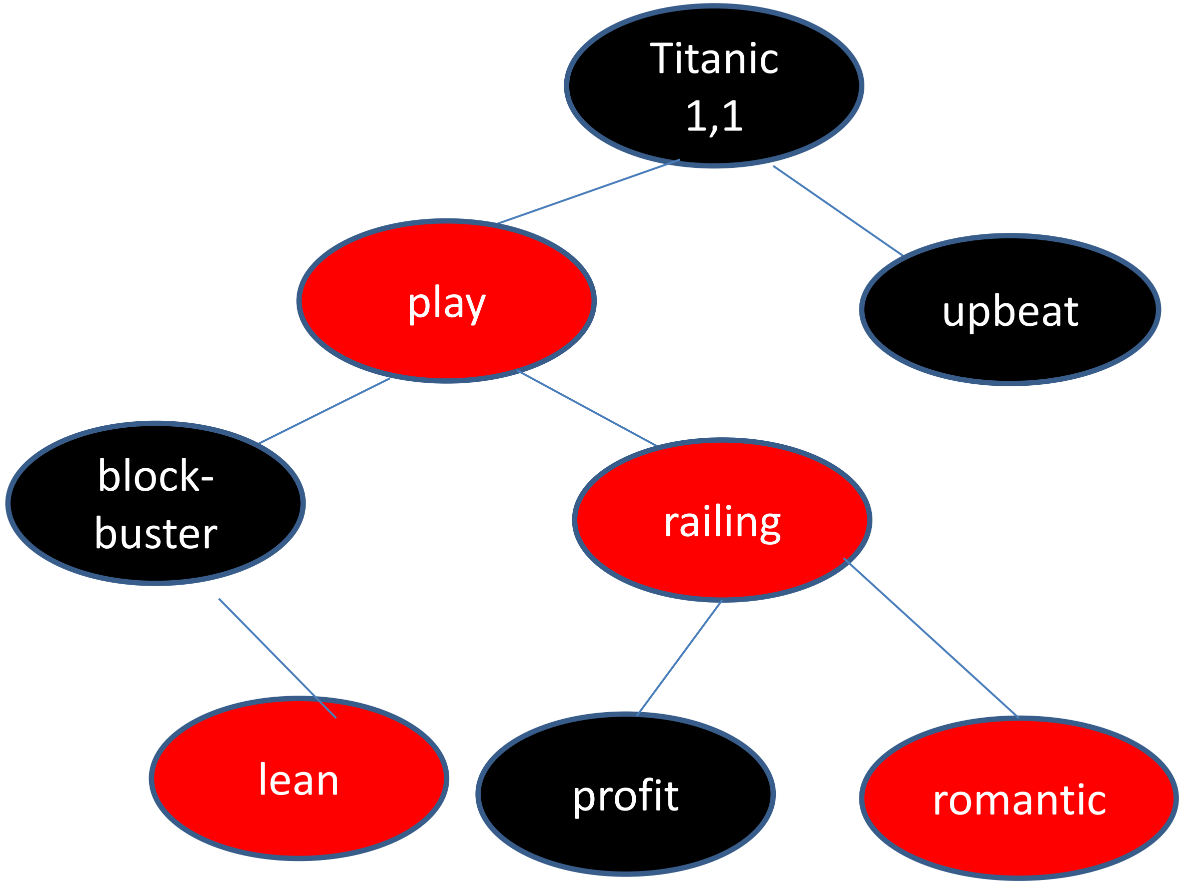
{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}



{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}

{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}

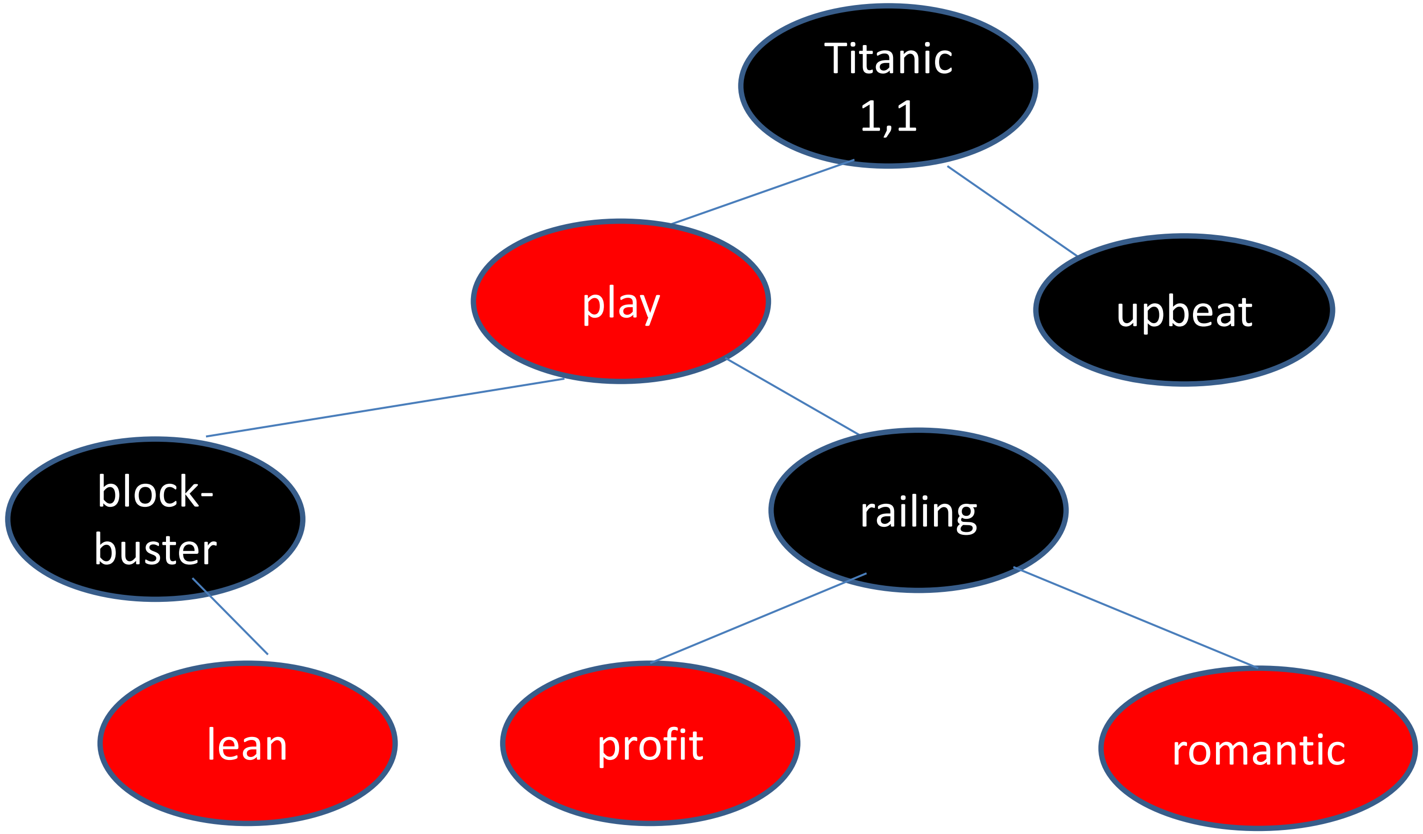




{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}

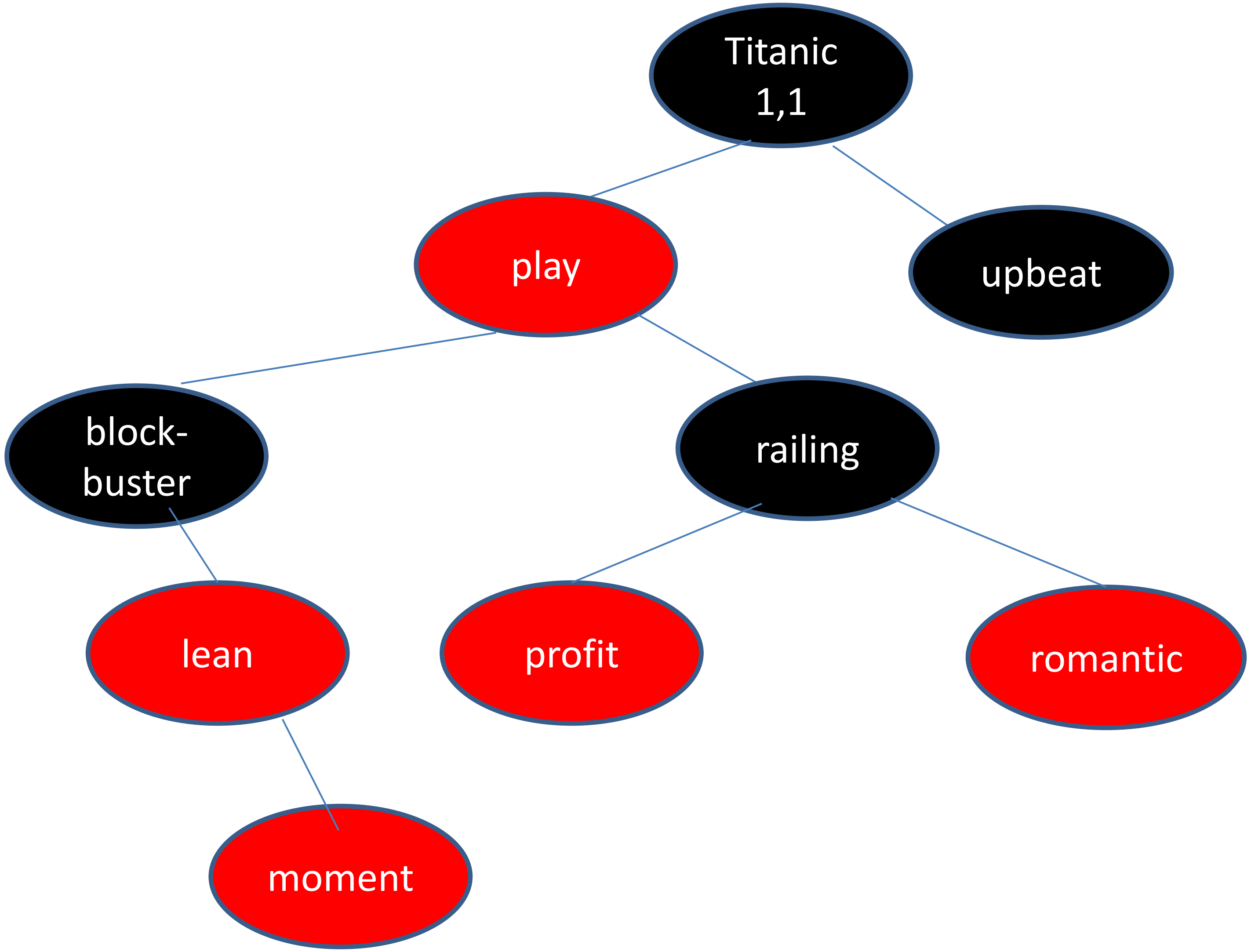
eva.knirsch@kiu.edu.ge

{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}

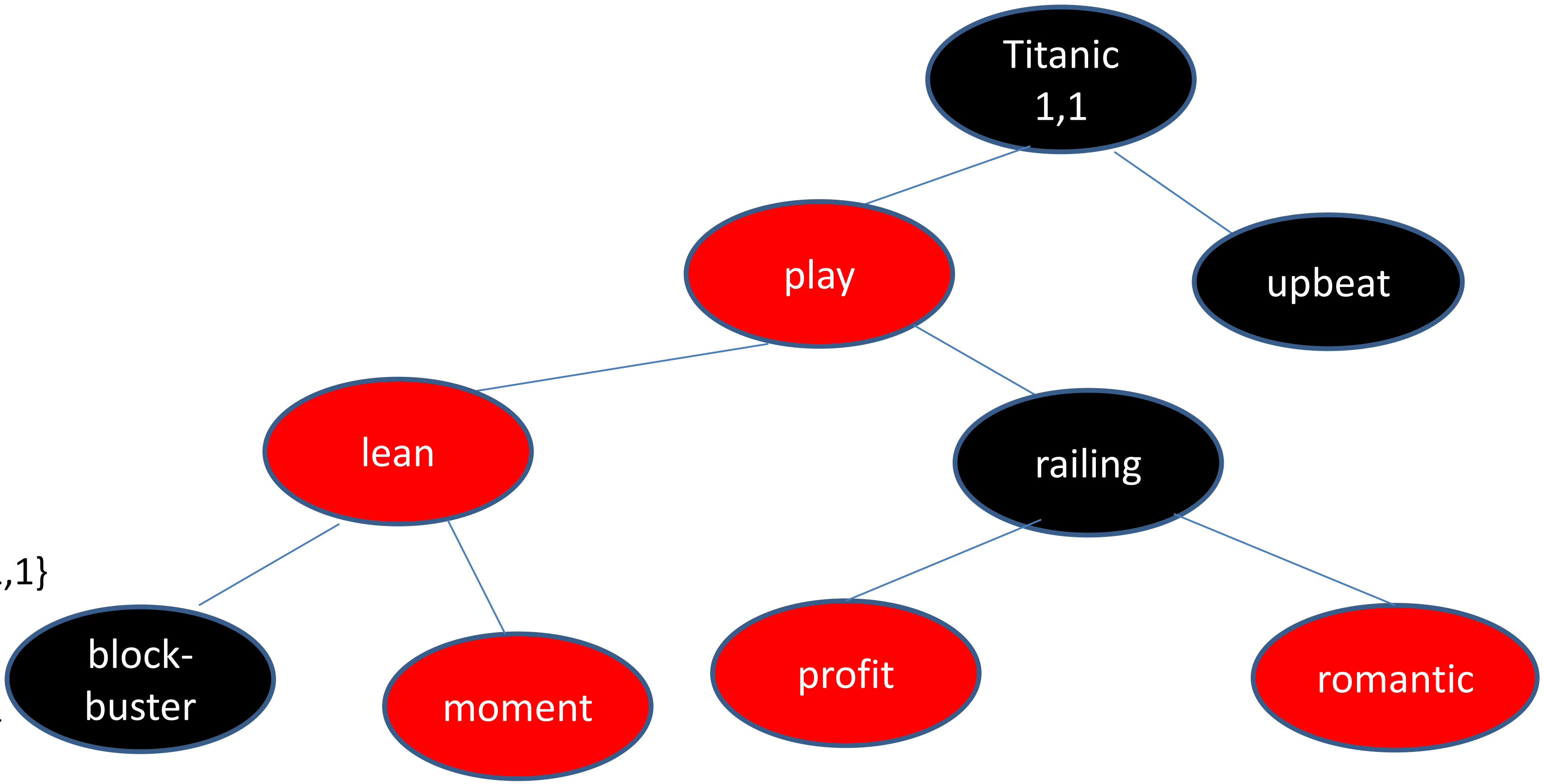


eva.knirsch@kiu.edu.ge

{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}
role

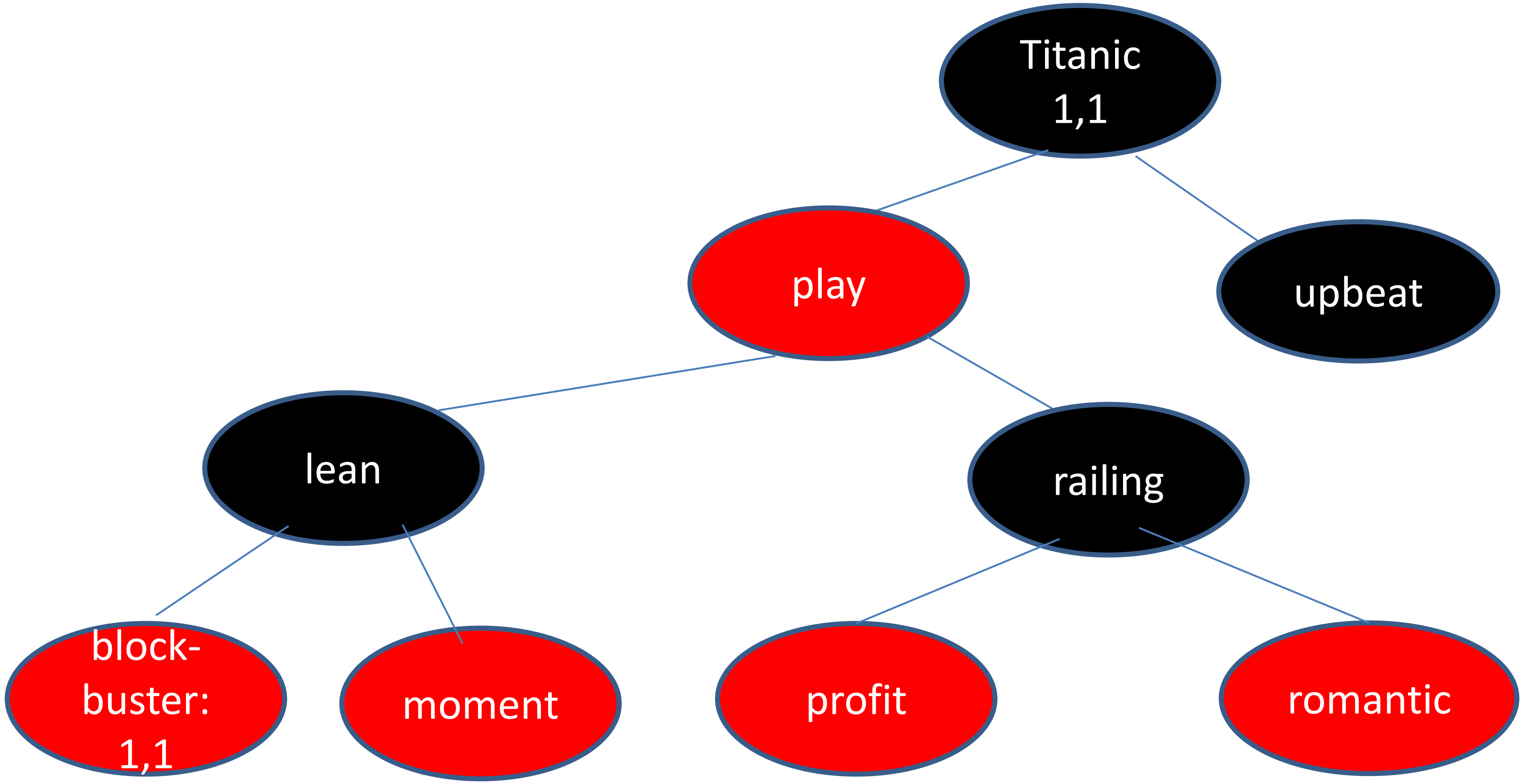


eva.knirsch@kiu.edu.ge



{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}
role

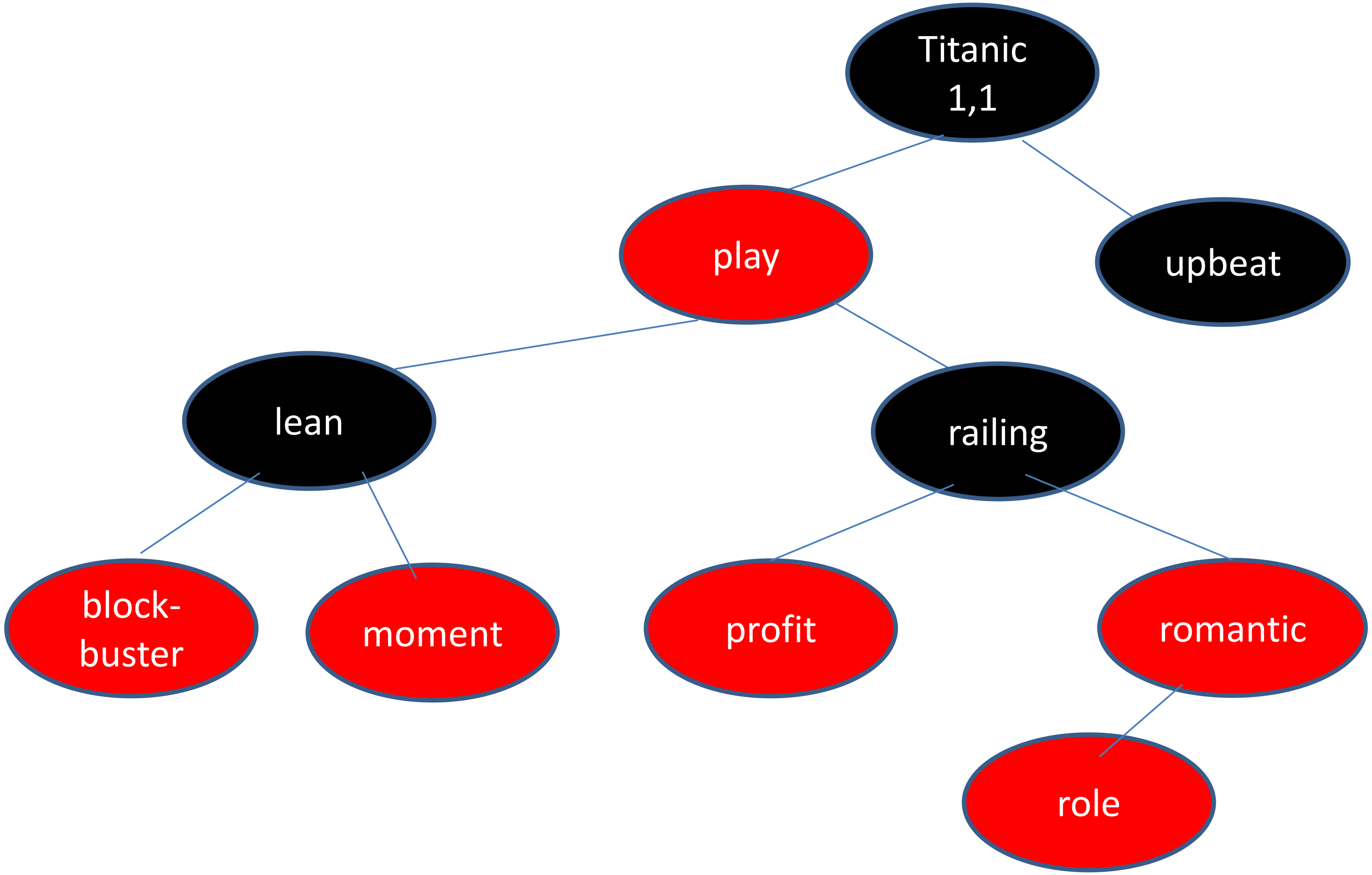
{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}
role



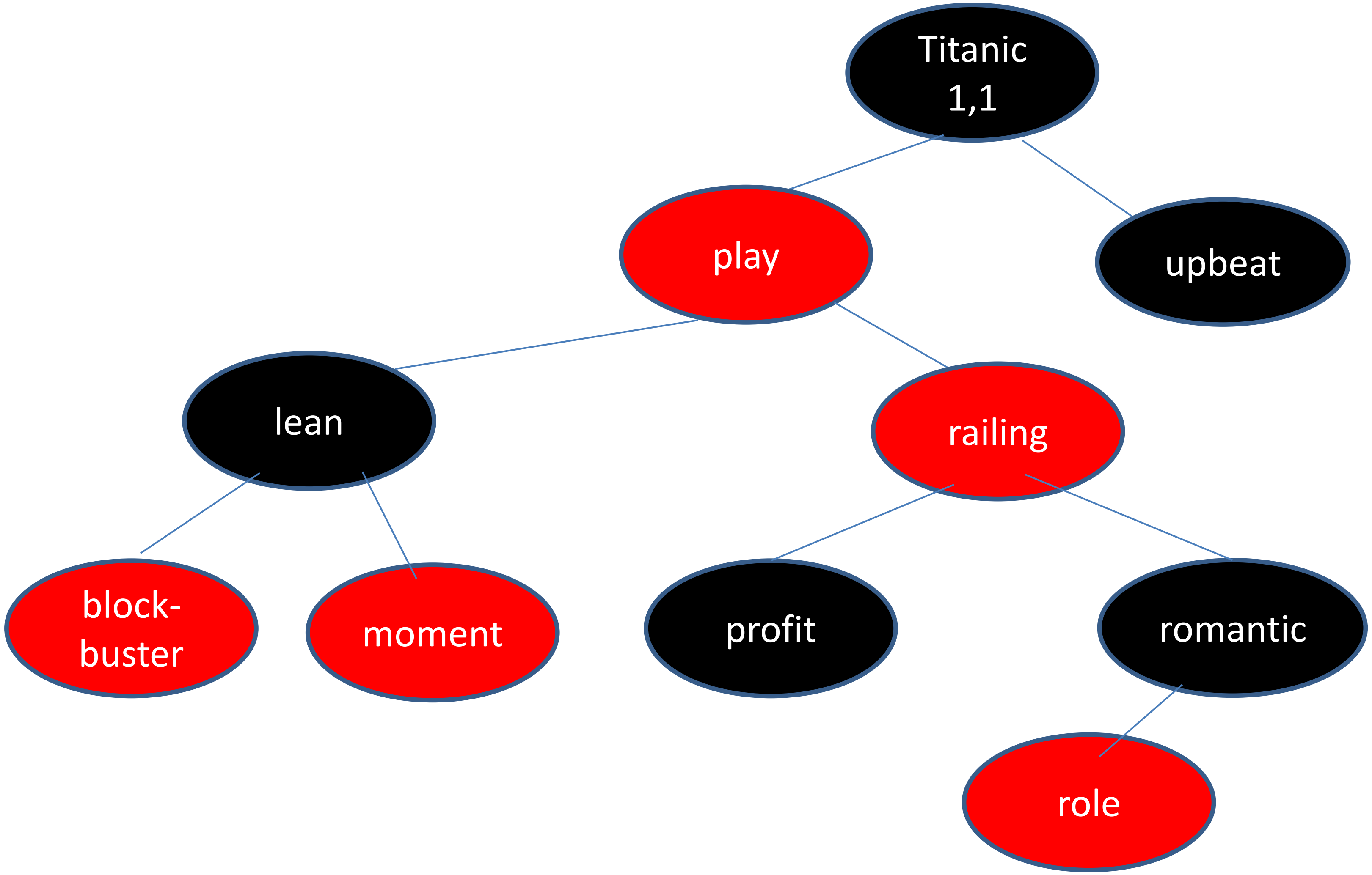
Update of existing key in memtable

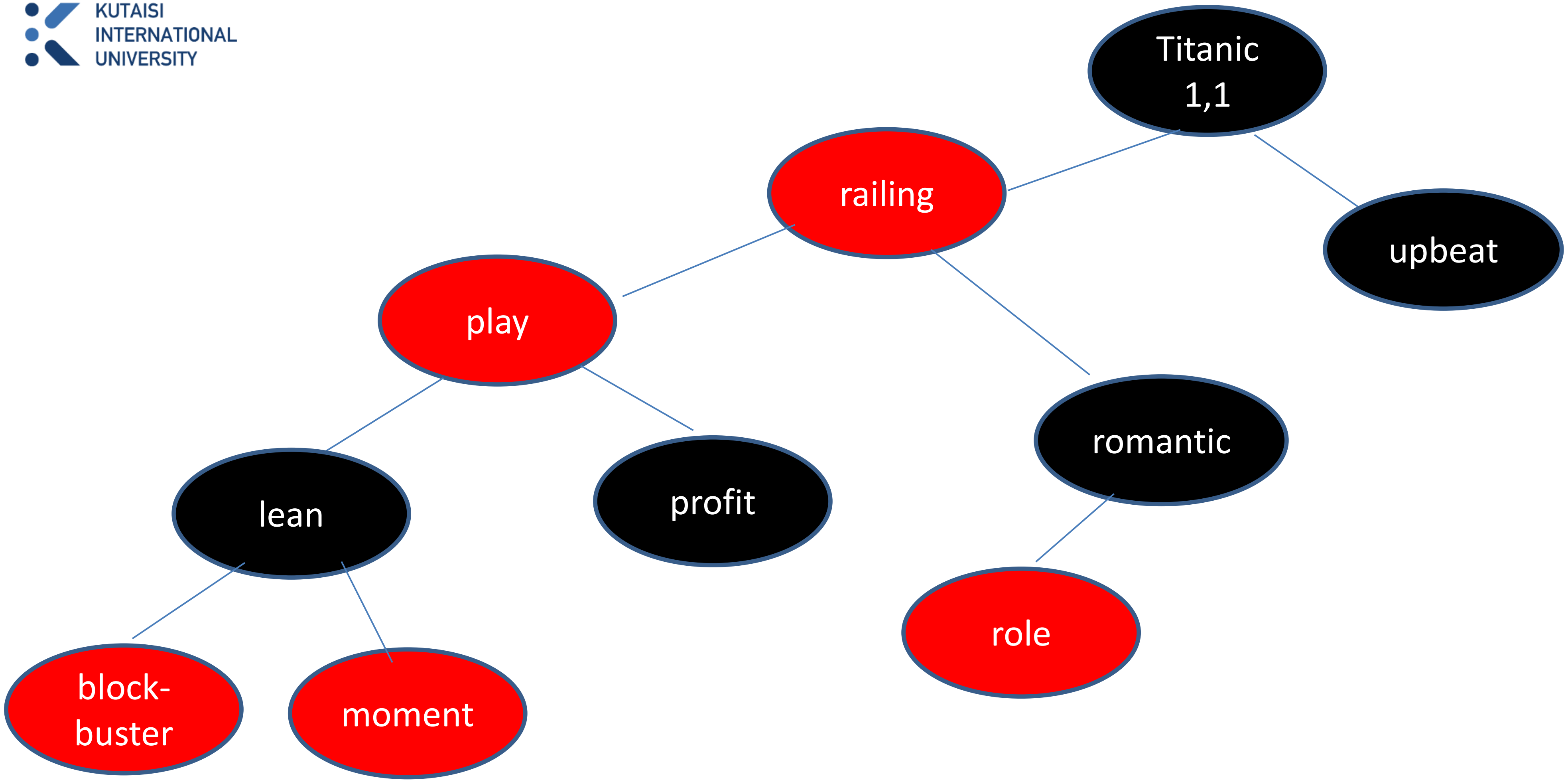
eva.knirsch@kiu.edu.ge

{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}
role

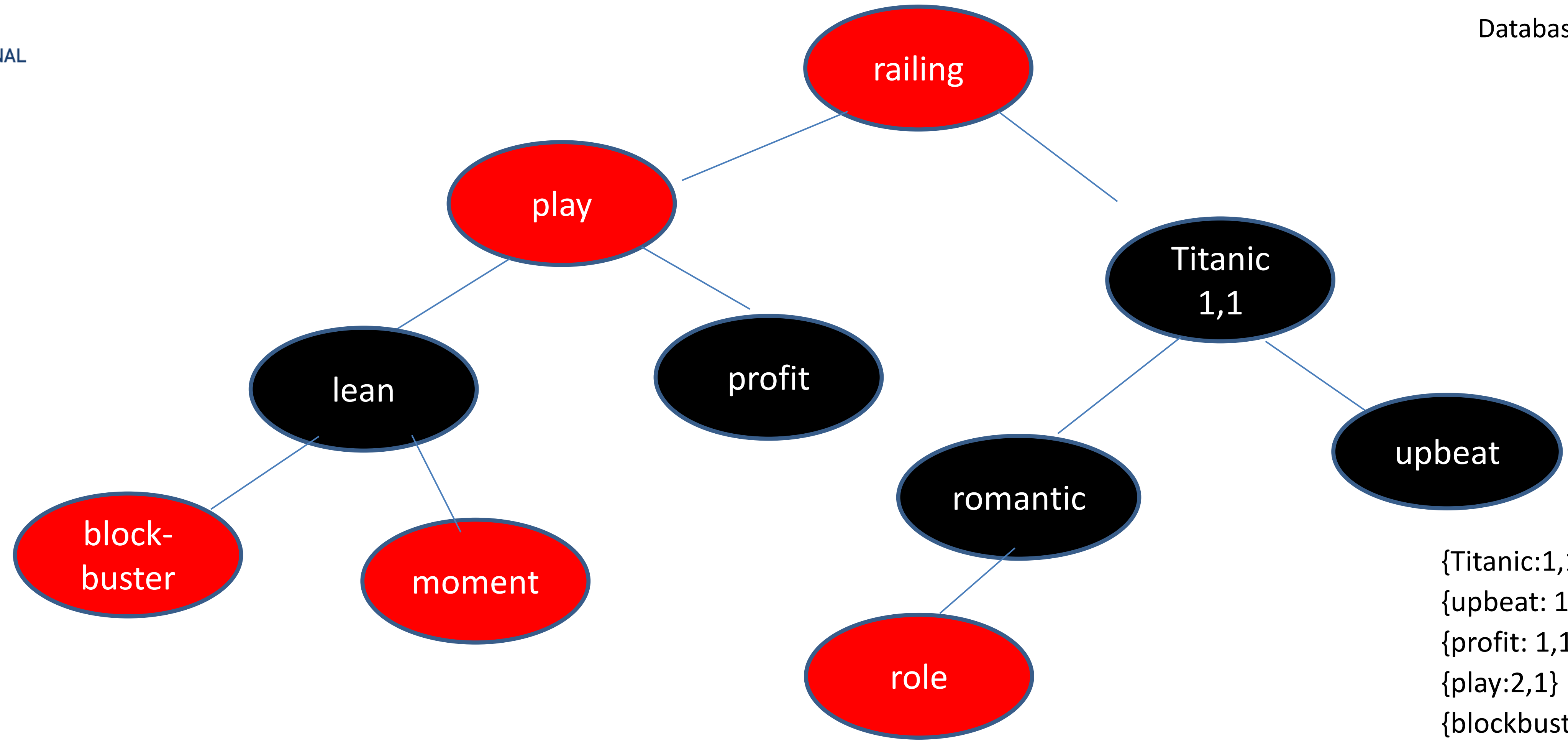


{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}
role

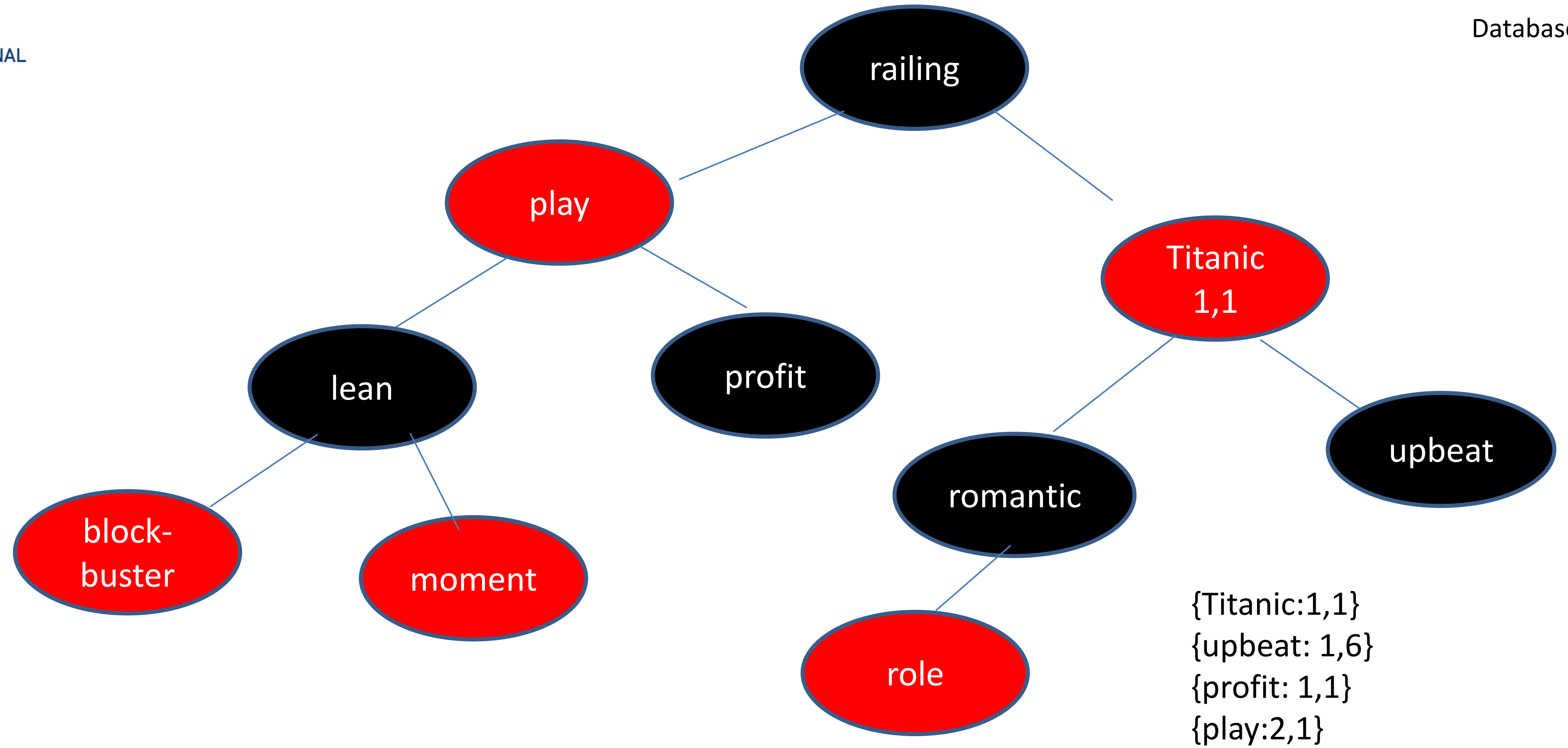




{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}
role



{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1;
2,1}
role



{Titanic:1,1}
{upbeat: 1,6}
{profit: 1,1}
{play:2,1}
{blockbuster: 1,1}
{lean: 1,1}
{railing:1,1}
{romantic: 1,1}
{moment:1,1}
{blockbuster: 1,1; 2,1}
role

Why is a rb tree and not a BTREE used as memtable?

Memtable Node Structure

rb_node_structure {

key
value
pointer left child
pointer right child
color flag: boolean

- small node size
- inserts: simple, modifications to some pointers according to rb-tree rules

BTREE Node Structure

btree_leaf_node_structure {

num_keys: number of keys

keys[array_of_keys]

values[array_of_values]

pointer_next_leaf

}

- large node size, many keys to keep the tree shallow
- inserts: need shifting the keys array, potential need for node-splitting

LSM-Storage: 2. SSTables
Writing memtables to persistent disk structure

- Once a memtable reaches a certain size, it is written out to disk into a persistent SSTable.
- The writing to disk ("flushing to disk") is performant because the memtable is written to disk sequentially.
- As soon as the flushing-to-disk process starts, a new memtable is started in memory. All incoming writes now go to the new memtable.

- As long as the flushing process continues, both memtables need to be accessible in memory.
  Why? old memtable for read requests, new for writes

- After the flush is complete and confirmed, the old memtable in memory can be discarded.
- Once the next memtable in memory reaches its threshhold limit, it is flushed out to a 2nd SSTable, the second segment. The next memtable becomes the 3rd SSTable, ...
- SSTables are immutable.

eva.knirsch@kiu.edu.ge

LSM-Storage: Updates

- Updates are always written into the current memtable in memory.
- If the key is in the current memtable
  → then update is done directly in memtable(example:blockbuster update on previous slides)

- If the key is not in the current memtable
  → treated like insert

  This is the case when the key and original value have already been flushed to SSTable on disk.

- In B trees, a key is only stored once. How is that in SSTables?

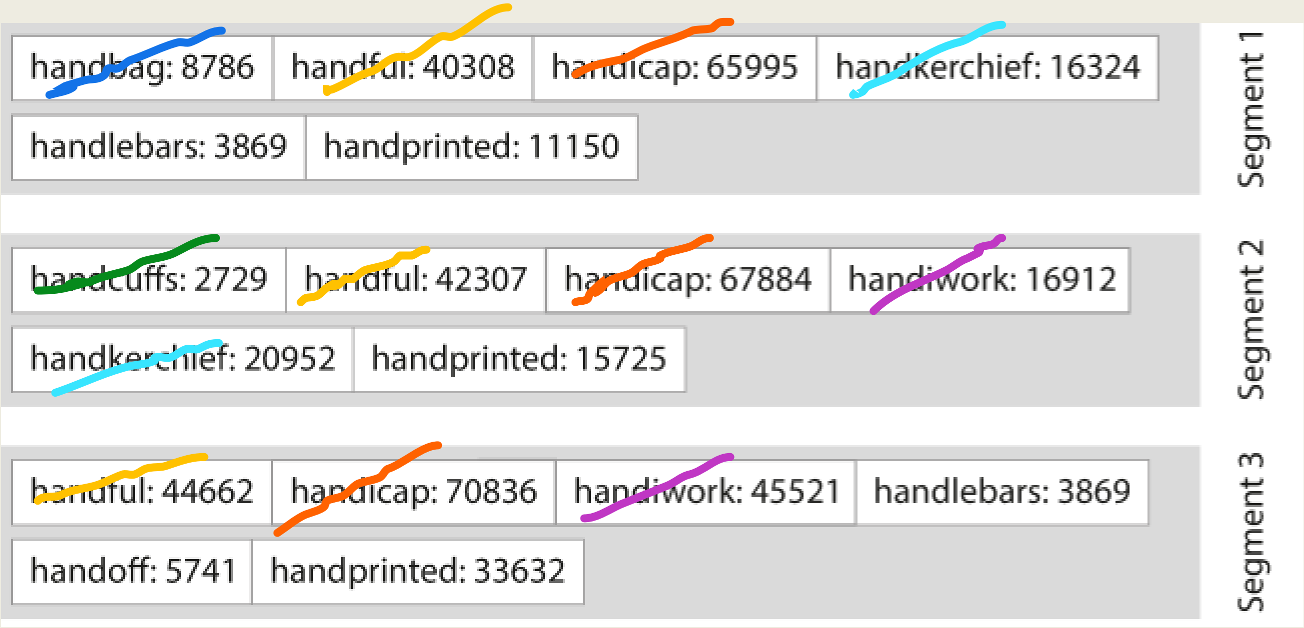
  duplicate keys

As time goes by, there are more and more SSTables segments on disk with more and more duplicate keys and different values.

Example taken out of [Kl], p. 77

LSM-Storage: 3. Compaction / Merge

- Compaction means to merge segments on disk and discard duplicate keys with old values.



Segment 1 is oldest, segment 3 is most recent.

If there are duplicate keys in different segments, the key with the most recent value is kept and the keys with stale values are discarded.

What does the merged segment look like?

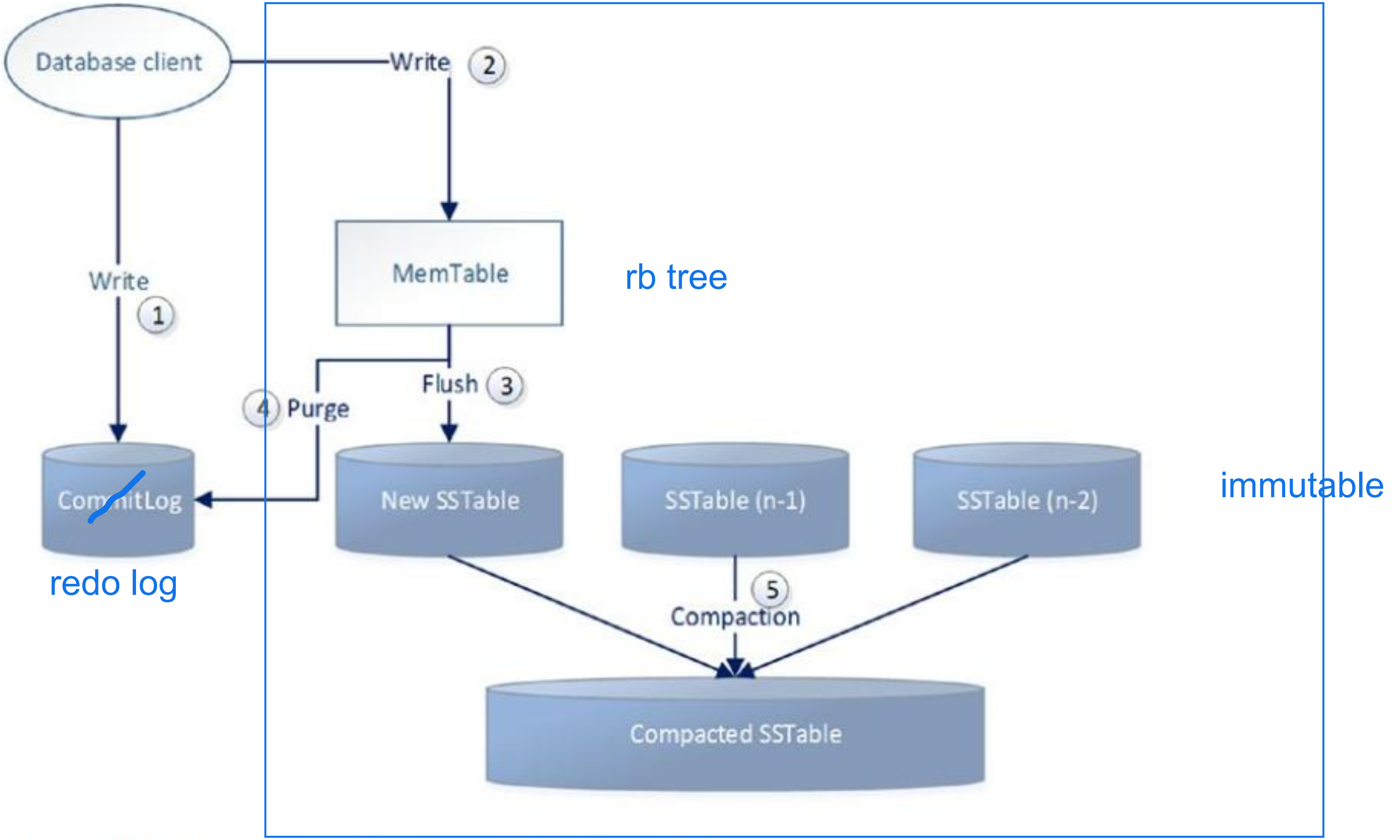
handbag:8786, handcuffs:2729, handful:4466, handicap:70836, handwork: 45521, handkerchef:20952,

Example taken out of [Kl], p. 76

eva.knirsch@kiu.edu.ge

LSM-Storage: 3. Compaction

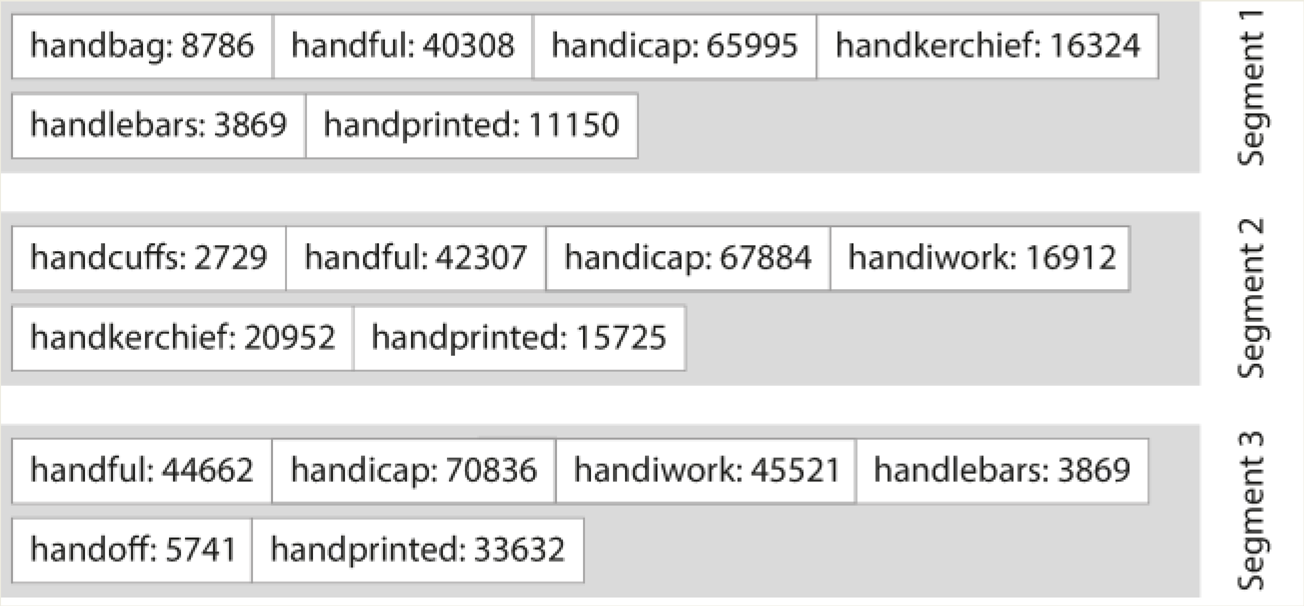
- Compaction always writes merged segments into a new file.
- Compaction can happen in the background: While compaction goes on, writes are still served in memtable and reads are served in the old, still accessible segments.
- After compaction is complete, read requests are routed to the new segment and the old segments are discarded.



[Ha], p161

Figure 10-11. *LSM architecture (Cassandra terminology)*

LSM-Storage: Searches / Read Operations



get(key = handbag)

How is this read executed?

1. search memtable
2.most recent SSTable segment
3. searches trhough all segments from most recent to oldest

Example taken out of [Kl], p. 76

eva.knirsch@kiu.edu.ge

LSM-Storage: Searches / Read Operations

Segment 1

handbag: 8786	handful: 40308	handicap: 65995	handkerchief: 16324
handlebars: 3869	handprinted: 11150		

Segment 2

handcuffs: 2729	handful: 42307	handicap: 67884	handiwork: 16912
handkerchief: 20952	handprinted: 15725		

Segment 3

handful: 44662	handicap: 70836	handiwork: 45521	handlebars: 3869
handoff: 5741	handprinted: 33632		

get(key = handbag)
How is this read executed?

Optimization:
Sparse index in memory per segment file, for example, one key for each page of the segment file. Index tells whether key could be in the segment and if so, on what page.

sparse index do not have all the keys just some of them.

Index for Segment 1

key	offset
handbag	0
handprinted	5
handsome	10
handwritten	15
...	

Index for Segment 2

key	offset
handcuffs	0
handprinted	5
handwashed	10
handyman	15
...	

Index for Segment 3

key	offset
handful	0
handprinted	5
handwritten	10
handzipped	15
...	

it just needs to load 1 page of segment 1

Example adapted from [Kl], p. 76

KUTAISI INTERNATIONAL UNIVERSITY

LSM-Storage: Searches / Read Operations

Segment 1

handbag: 8786	handful: 40308	handicap: 65995	handkerchief: 16324
handlebars: 3869	handprinted: 11150		

Segment 2

handcuffs: 2729	handful: 42307	handicap: 67884	handiwork: 16912
handkerchief: 20952	handprinted: 15725		

Segment 3

handful: 44662	handicap: 70836	handiwork: 45521	handlebars: 3869
handoff: 5741	handprinted: 33632		

get(key = handwashed)
How is this read executed?

Optimization:
Sparse index in memory per segment file, for example, one key for each page of the segment file. Index tells whether key could be in the segment and if so, on what page.

Index for Segment 1

key	offset
handbag	0
handprinted	5
handsome	10
handwritten	15
...	

Index for Segment 2

key	offset
handcuffs	0
handprinted	5
handwashed	10
handyman	15
...	

Index for Segment 3

key	offset
handful	0
handprinted	5
handwritten	10
handzipped	15
...	

it should go segment 3 first because value can also be in there and then segment 2

Example adapted from [Kl], p. 76

eva.knirsch@kiu.edu.ge

KUTAISI INTERNATIONAL UNIVERSITY

LSM-Storage: Searches / Read Operations

Segment 1

handbag: 8786	handful: 40308	handicap: 65995	handkerchief: 16324
handlebars: 3869	handprinted: 11150		

Segment 2

handcuffs: 2729	handful: 42307	handicap: 67884	handiwork: 16912
handkerchief: 20952	handprinted: 15725		

Segment 3

handful: 44662	handicap: 70836	handiwork: 45521	handlebars: 3869
handoff: 5741	handprinted: 33632		

get(key = handsome)
How is this read executed?

Optimization:
Sparse index in memory per segment file, for example, one key for each page of the segment file. Index tells whether key could be in the segment and if so, on what page.

Index for Segment 1

key	offset
handbag	0
handprinted	5
handsome	10
handwritten	15
...	

Index for Segment 2

key	offset
handcuffs	0
handprinted	5
handwashed	10
handyman	15
...	

Index for Segment 3

key	offset
handful	0
handprinted	5
handwritten	10
handzipped	15
...	

it can be in segment 3 and 2 because handsome is between handprinted and handwritten. And it of course is in segment 1. So in this case sparse index did not help at all.

Example adapted from [Kl], p. 76

eva.knirsch@kiu.edu.ge

LSM-Storage: Bloom Filters

- The search in the SSTables could further be optimized if one could say for certain that a key is not in a segment (even though the index says that it could be in the segment).

- This is exactly what a Bloom filter does: A Bloom filter is a data structure that tells if an element is definitely not in a set. It cannot tell whether a key is positively in a set.

- LSM: A Bloom filter can tell whether a key is definitely not in a segment.

  - **True negative:** means that the Bloom filter returns correctly that **$k \notin S$.** ✓
  - **False negative:** would mean that the Bloom filter returns **$k \notin S$ when actually $k \in S$.** False negatives do not happen with Bloom filters.
  - **True positive:** means that the Bloom filter correctly returns k **$\in S$** ✓
  - **False positive:** means that the Bloom filter wrongly returns k **$\in S$** ✓

- When the database system uses a Bloom filter, this is checked first, then the sparse indexes.
- Bloom filters are kept in memory.

eva.knirsch@kiu.edu.ge

LSM-Storage: Searches / Read Operations

Segment 1

handbag: 8786	handful: 40308	handicap: 65995	handkerchief: 16324
handlebars: 3869	handprinted: 11150		

Segment 2

handcuffs: 2729	handful: 42307	handicap: 67884	handiwork: 16912
handkerchief: 20952	handprinted: 15725		

Segment 3

handful: 44662	handicap: 70836	handiwork: 45521	handlebars: 3869
handoff: 5741	handprinted: 33632		

get(key = handsome)
How is this read executed?

Optimization: Bloom filter: would, e.g. return that handsome is NOT in segment 3 but could be in segment 2 and / or segment 1.

Index for Segment 1

key	offset
handbag	0
handprinted	5
handsome	10
handwritten	15
...	

Index for Segment 2

key	offset
handcuffs	0
handprinted	5
handwashed	10
handyman	15
...	

Index for Segment 3

key	offset
handful	0
handprinted	5
handwritten	10
handzipped	15
...	

Example adapted from [Kl], p. 76

LSM-Storage: Searches / Read Operations

Segment 1

handbag: 8786	handful: 40308	handicap: 65995	handkerchief: 16324
handlebars: 3869	handprinted: 11150		

Segment 2

handcuffs: 2729	handful: 42307	handicap: 67884	handiwork: 16912
handkerchief: 20952	handprinted: 15725		

Segment 3

handful: 44662	handicap: 70836	handiwork: 45521	handlebars: 3869
handoff: 5741	handprinted: 33632		

get(key = handsome)
How is this read executed?

Optimization: Bloom filter: would, e.g. return that handsome is NOT in segment 3 and not in segment 2 but could be in segment 1.

Index for Segment 1

key	offset
handbag	0
handprinted	5
handsome	10
handwritten	15
...	

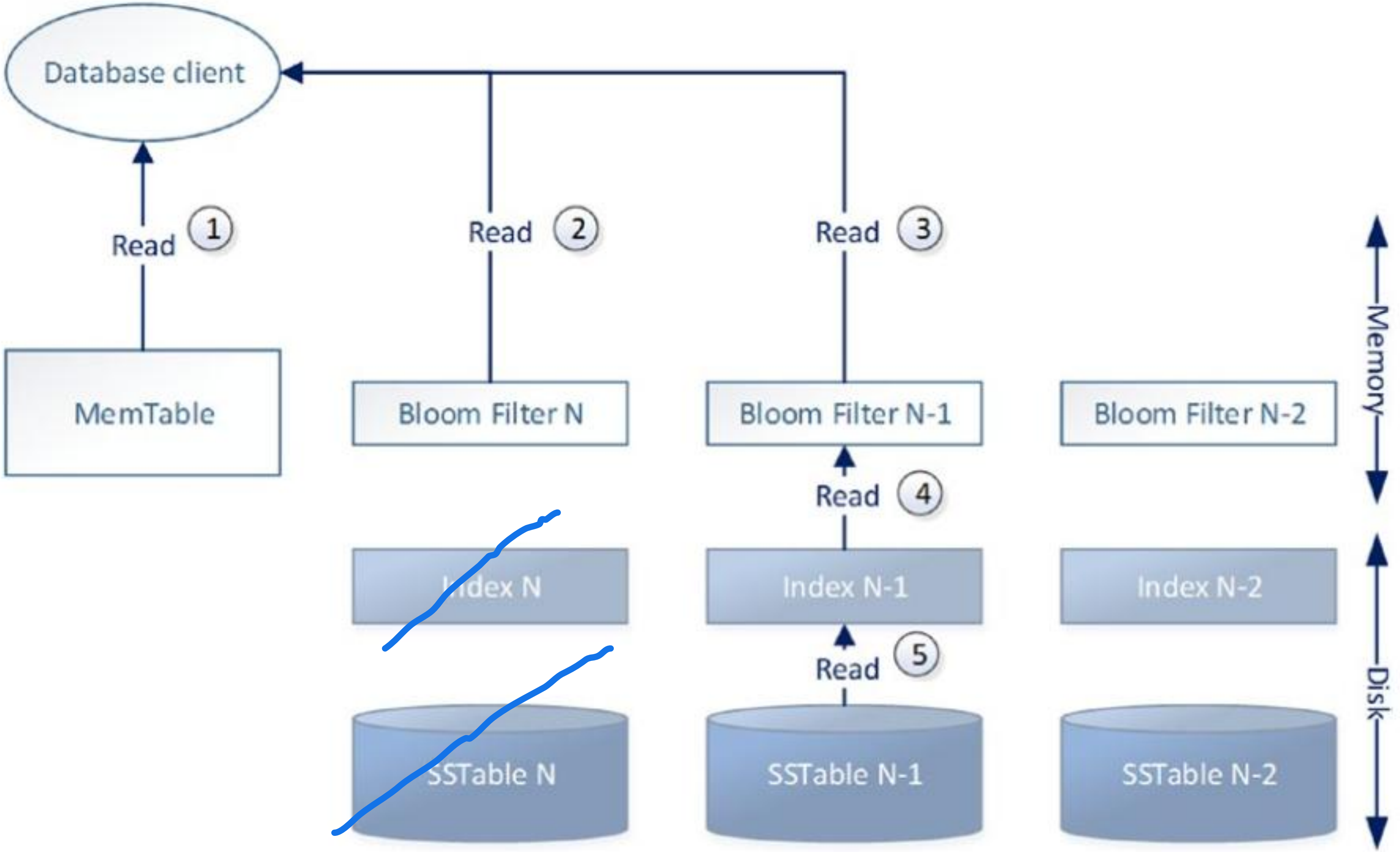
Index for Segment 2

key	offset
handcuffs	0
handprinted	5
handwashed	10
handyman	15
...	

Index for Segment 3

key	offset
handful	0
handprinted	5
handwritten	10
handzipped	15
...	

Example adapted from [Kl], p. 76



[Ha], p161

Figure 10-12. *Log-structured merge tree reads (Cassandra terminology)*