Distributed Data Management
Storage and Retrieval

Thorsten Papenbrock
Felix Naumann

F-2.03/F-2.04, Campus II
Hasso Plattner Institut

Introduction

Layering Data Models

1. Conceptual layer
   - Data structures, objects, modules, …
     - ➢ Application code
2. Logical layer
   - Relational tables, JSON, XML, graphs, …
     - ➢ Database management system (DBMS) or storage engine

our focus now

3. Representation layer
   - Bytes in memory, on disk, on network, …
     - ➢ Database management system (DBMS) or storage engine
4. Physical layer
   - Electrical currents, pulses of light, magnetic fields, …
     - ➢ Operating system and hardware drivers

Overview

Objective

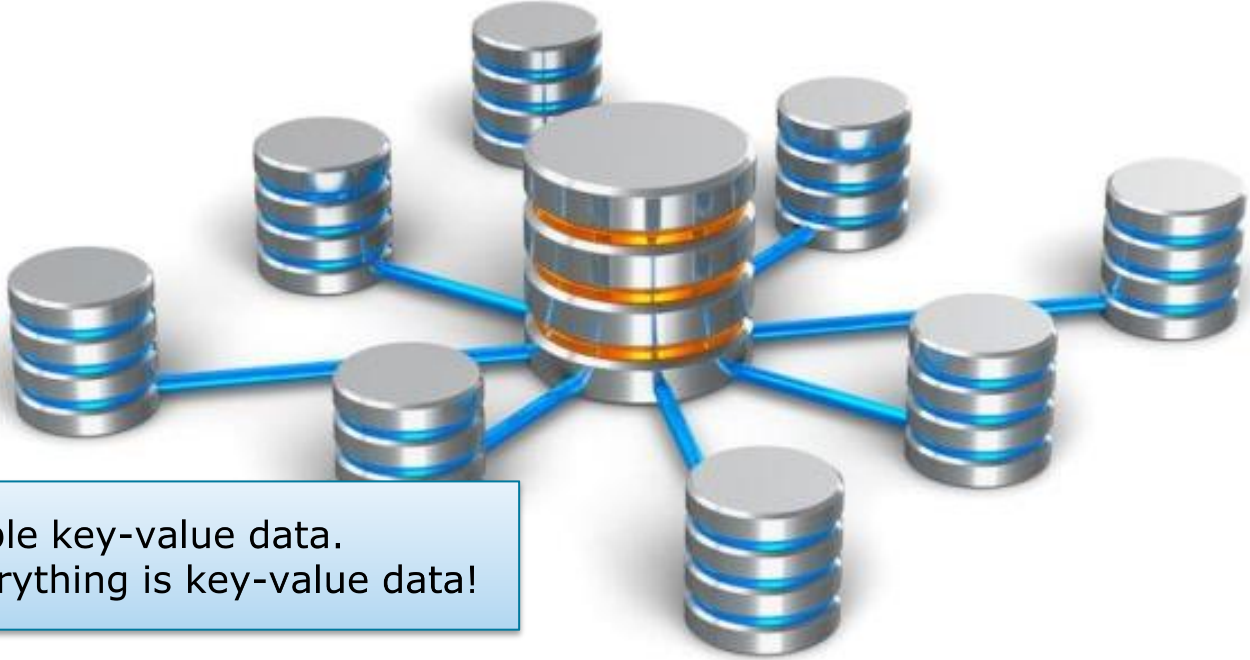
**Design a distributed DBMS
for fast storage and retrieval
of huge and evolving datasets**

Objective

With techniques used by …

Design a distributed DBMS
for fast storage and retrieval
of huge and evolving datasets

Objective



Most data models resemble key-value data.
- Let’s pretend that everything is key-value data!

Design a distributed DBMS
for fast storage and retrieval
of huge and evolving datasets

A Tiny Database

- Basic database tasks: (a) write given data, (b) read specific data
- A tiny key-value store in two Bash functions:

```
#!/bin/bash
db_set () {
   echo "$1,$2" >> database
}
db_get () {
   grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

Concatenate the first two parameters by "," and write/append them to the file named "database"

Find all lines starting with first parameter, remove first parameter from lines, and select the last line

Why?

- It works:

```
$ db_set 1234 '{"name":"Berlin","type":"city"}'
$ db_get 1234
'{"name":"Berlin","type":"city"}'
```

A Tiny Database

- Assume the following input-sequence:

```
$ db_set 1234 '{"name":"Berlin","type":"city"}'
$ db_set 42 '{"name":"Germany","type":"country"}'
$ db_set 42 '{"name":"Germany","type":"country","capital":"Berlin"}'
```

- The according "database"-file (= CSV-file):

```
$ cat database
1234,{"name":"Berlin","type":"city"}
42,{"name":"Germany","type":"country"}
42,{"name":"Germany","type":"country","capital":"Berlin"}
```

"database" is a **Log** file:

- Append only, no removal of old values
  - ➢ Only the last entry for each key is valid.
- Fast writes (O(1)) but slow reads (O(*n*) with *n* records in the log)
- To speed-up reads: Indexes!

Overview

Objective

Design a distributed DBMS

for fast storage and retrieval

of huge and evolving datasets

Index

- An additional data structure that helps to locate data by some search criterion, i.e., the key
  - Key = one or more identifying attributes
- Basically a key-value store, where values can be actual data or pointers to relational records, documents, graph nodes/edges, …
- Improves data retrieval operations
  - Usually O(n) to O(log(n)) or O(1)
- Costs additional writes to index structure and storage space
  - ➢ Use indexes carefully (not too many)!
- Different index implementations (data structures) have different strengths
  - ➢ Choose the right index for your queries (workload)!

Hash Index

Definition

- A hash index is a hash map (dictionary) that maps keys to the addresses (in memory, on disk, on the network, …) of their values/records.
- The hash map uses a hash function to calculate mapping of keys and positions and is usually kept in memory.

Uses

- key-value stores, multilayered indexes, data distribution (load balancing, sharding, …)

Strength

- Point queries: An index look-up delivers a value's position in O(1).

Weaknesses

- Range queries require to look up each key individually.
- Hash map must fit into main memory; hash maps on disk perform poorly.

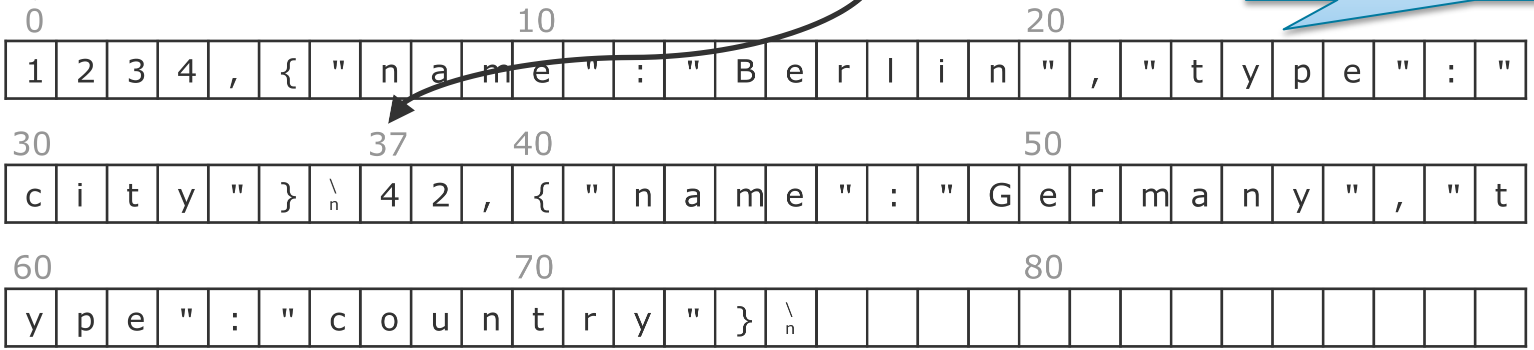
Hash Index – Example

hash function

key	byte offset
1234	0
42	37

In-memory hash map

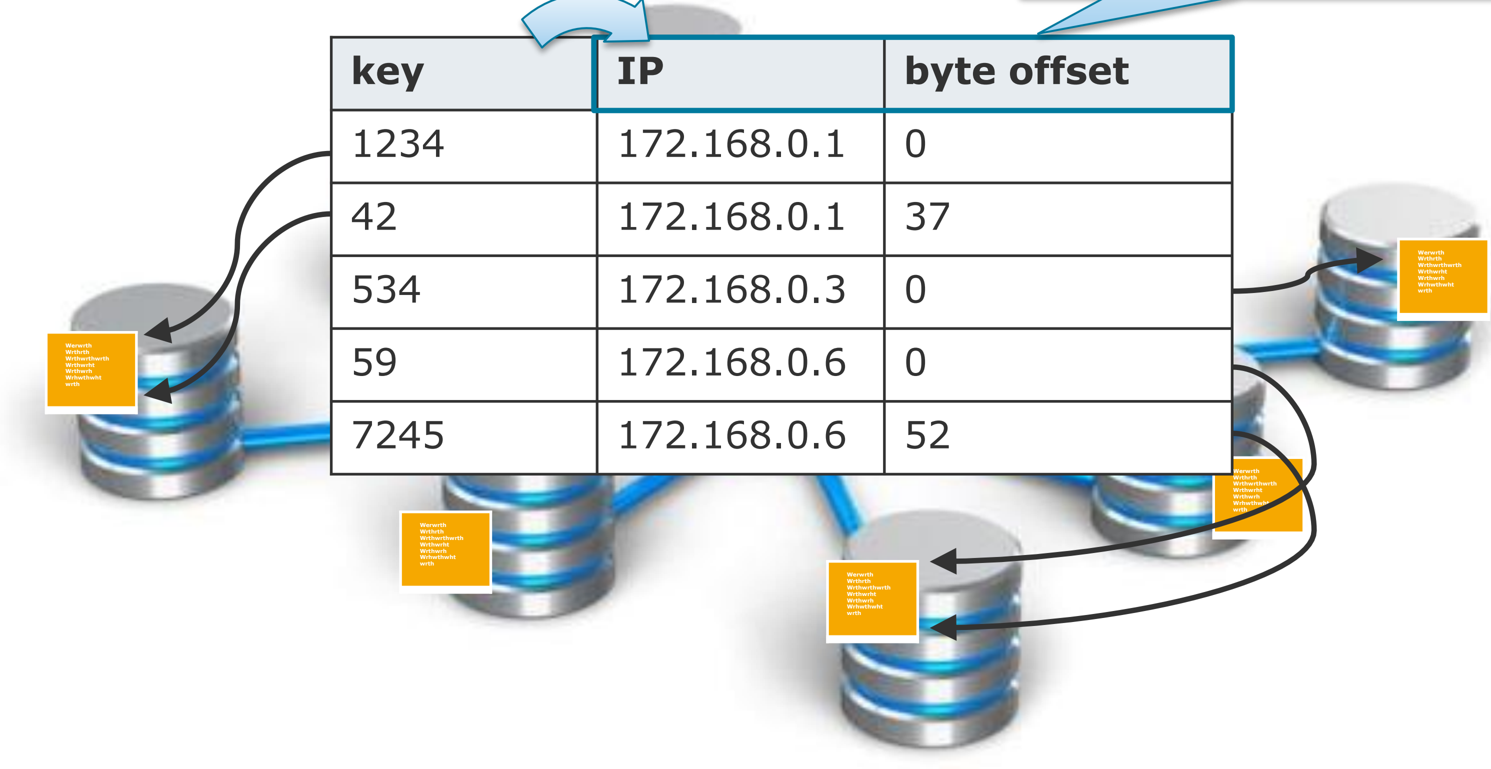
Log-structured file on disk (each box is one byte)



Objective

Design a distributed DBMS
for fast storage and retrieval
of huge and evolving datasets

Remote Pointers



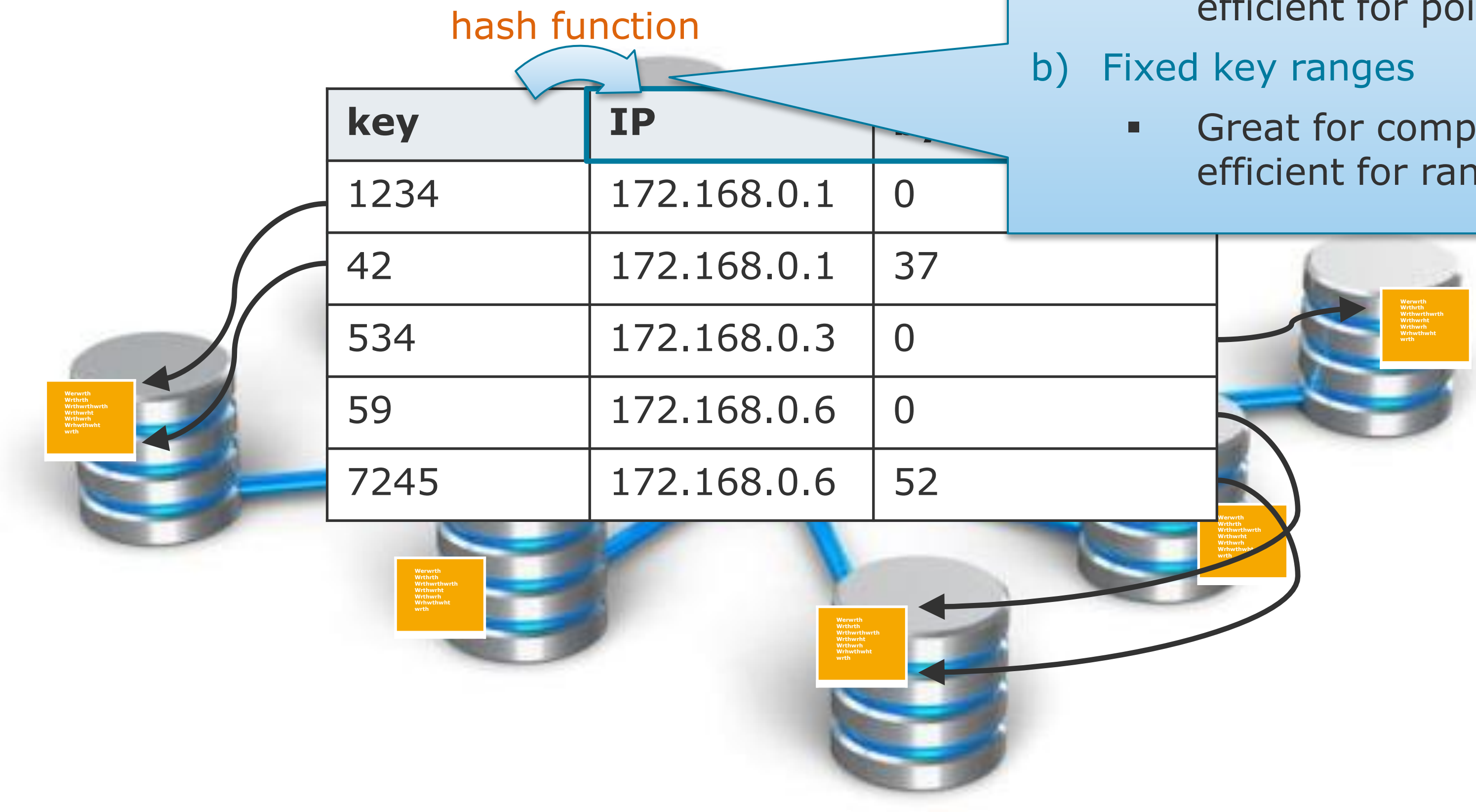
key	IP	byte offset
1234	172.168.0.1	0
42	172.168.0.1	37
534	172.168.0.3	0
59	172.168.0.6	0
7245	172.168.0.6	52

Remote Pointers

Key-to-node assignment strategies:

a) Random
   - Great for load balancing and efficient for point queries

b) Fixed key ranges
   - Great for compression and efficient for range queries



key	IP	
1234	172.168.0.1	0
42	172.168.0.1	37
534	172.168.0.3	0
59	172.168.0.6	0
7245	172.168.0.6	52

Objective

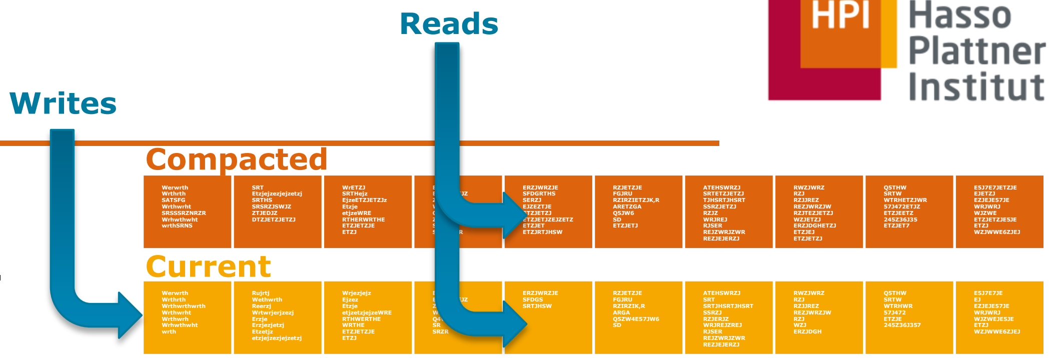
Design a distributed DBMS
for fast storage and retrieval
of huge and evolving datasets

Segmentation

Controlling file growth

- Indexed data, i.e., log is insert-only (for good write performance).
  - ➢ Frequent updates make files unnecessarily large.
  - ➢ Example: a store that maps products to stock-counts
    - Each purchase increments a stock-count → new record!
    - Each sale decrements a stock-count → new record!
    - But: Collection of products is almost constant…
- Solution: Consolidate/compact the log regularly freeing up disk space.
  - ➢ How do we do this on a running system?
    - ➢ Segmentation!

Segmentation

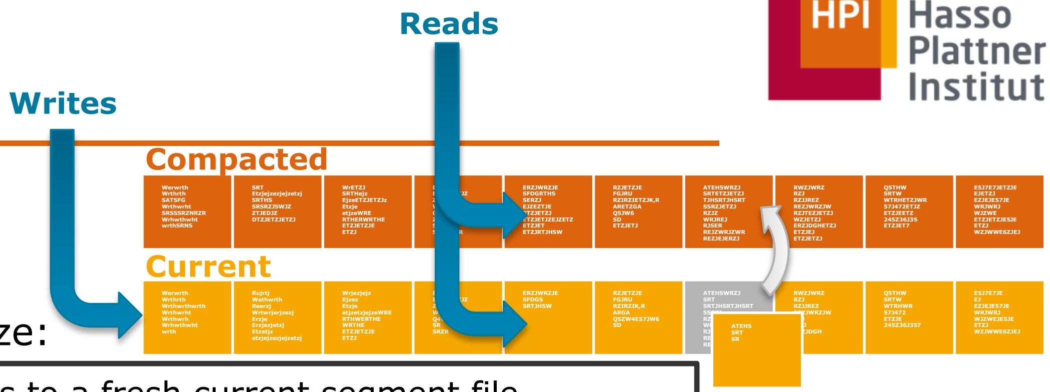


Segmentation

- Break log into segments of fixed size.
- Each segment …
  - stores a range of keys.
  - can be subject for distribution!
  - has two representations:
    - **Compacted**
      - Static (= does not allow writes)
      - Purged (= only most recent value for each key)
    - **Current**
      - Dynamic (= allows appending writes)
      - Unchecked (= same key might appear multiple times)

Huge and Evolving Datasets

Segmentation



Segmentation

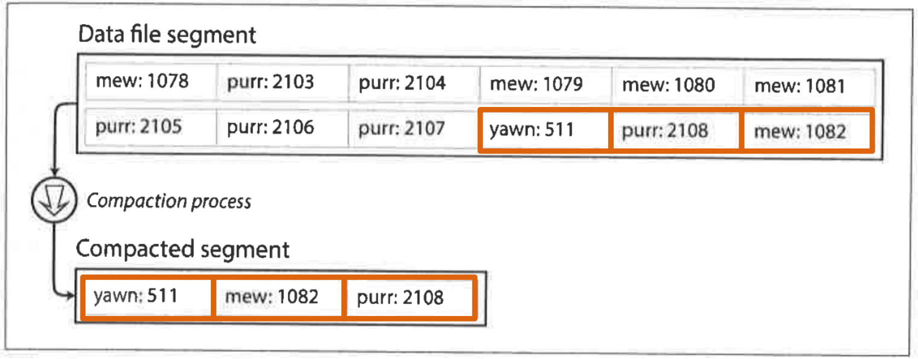
- Whenever a segment reaches max size:

1. Close the segment and redirect writes to a fresh current segment file.
2. Compact the closed segment file:
   1. Create a new compacted segment file.
   2. Read the closed segment file backwards.
   3. If a key is read for the first time:
      - Write the entry (key + value) into a compacted segment file.
3. Merge the old compacted segment file into the new compacted segment file:
   1. Read old the old compacted segment file.
   2. If a key is not present in the new compacted segment file:
      - Write the entry (key + value) into the new compacted segment file.
4. Delete the old segment file and the old compacted segment file.

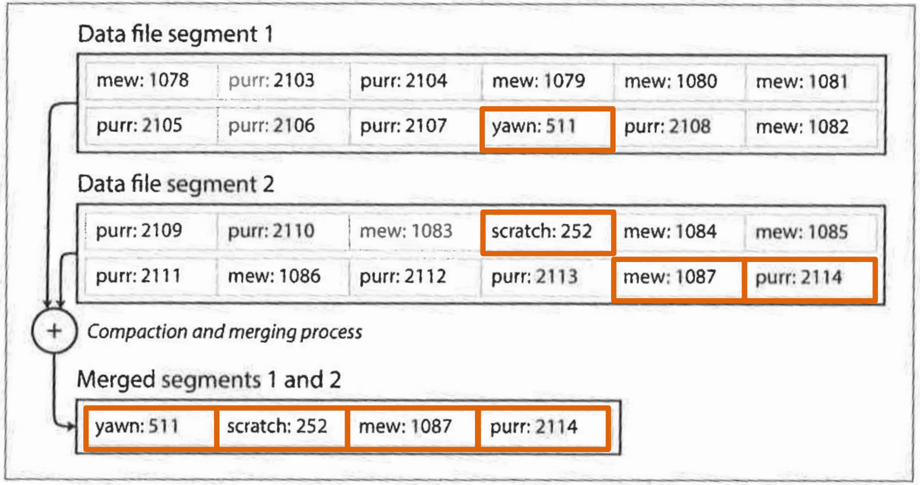
Segmentation

Segmentation

- Compact:



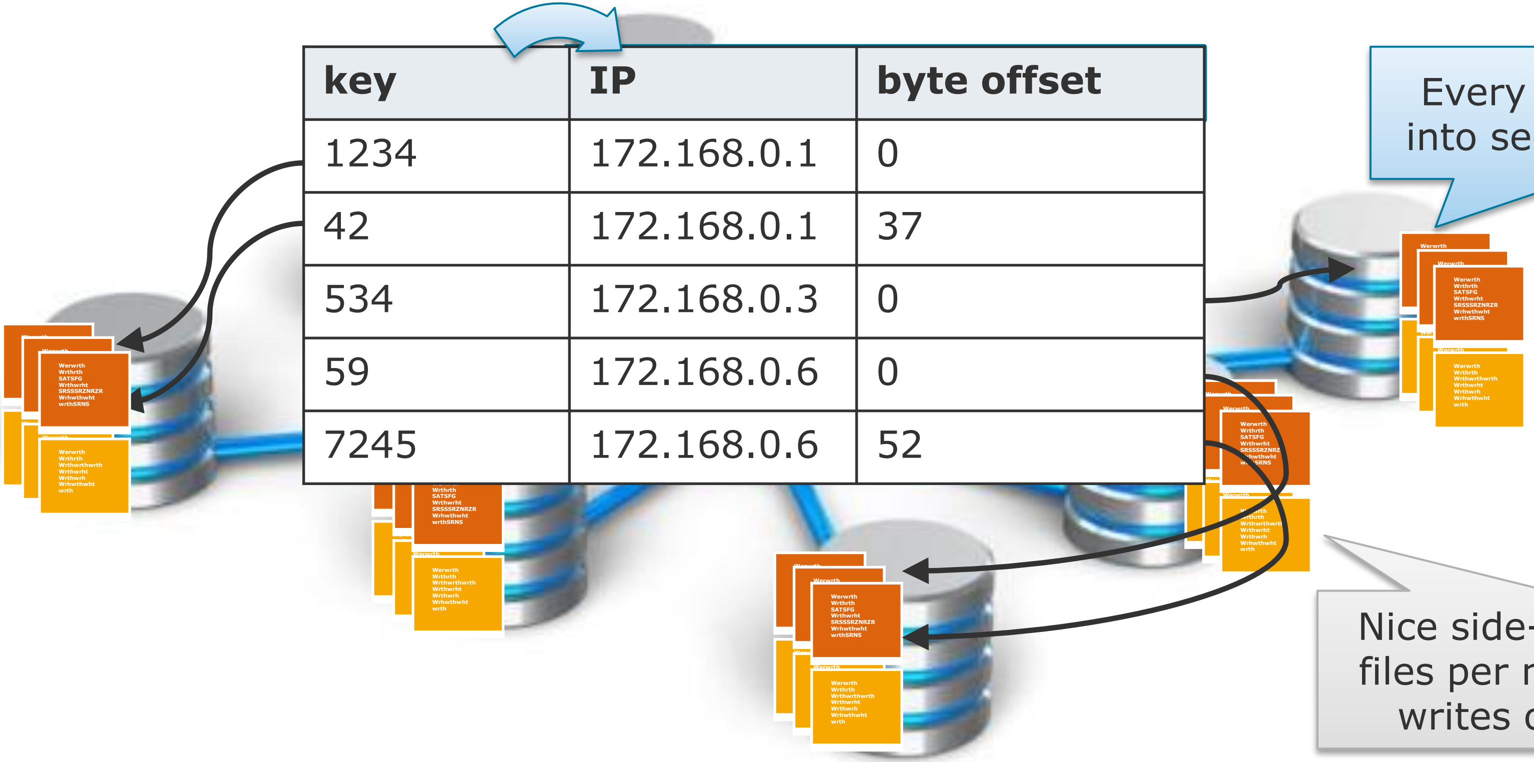
- Merge:
  (+ Compact)



Segmentation

hash function

key	IP	byte offset
1234	172.168.0.1	0
42	172.168.0.1	37
534	172.168.0.3	0
59	172.168.0.6	0
7245	172.168.0.6	52



Every log is now split into segmentation files

Nice side-effect: multiple files per node for parallel writes on each node!

Distributed Data Management

Storage and Retrieval

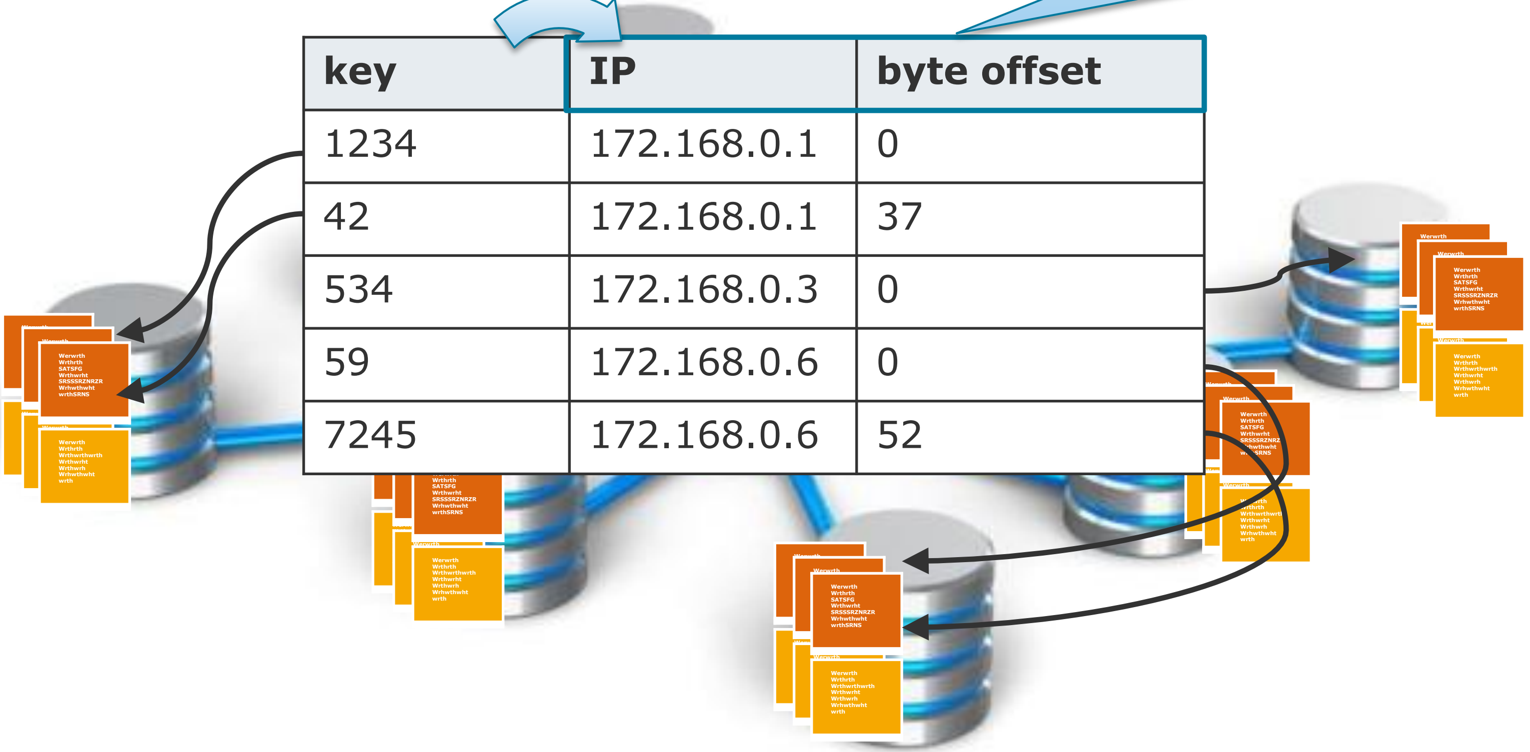
Huge and Evolving Datasets

Segmentation

If the data is really large, then this dense index will also be too large and updating it too expensive!

hash function

key	IP	byte offset
1234	172.168.0.1	0
42	172.168.0.1	37
534	172.168.0.3	0
59	172.168.0.6	0
7245	172.168.0.6	52



Segmentation

Index = key-value store

➢ Segmentation!

key	IP	byte offset
153	172.168.0.1	243
362	172.168.0.1	134

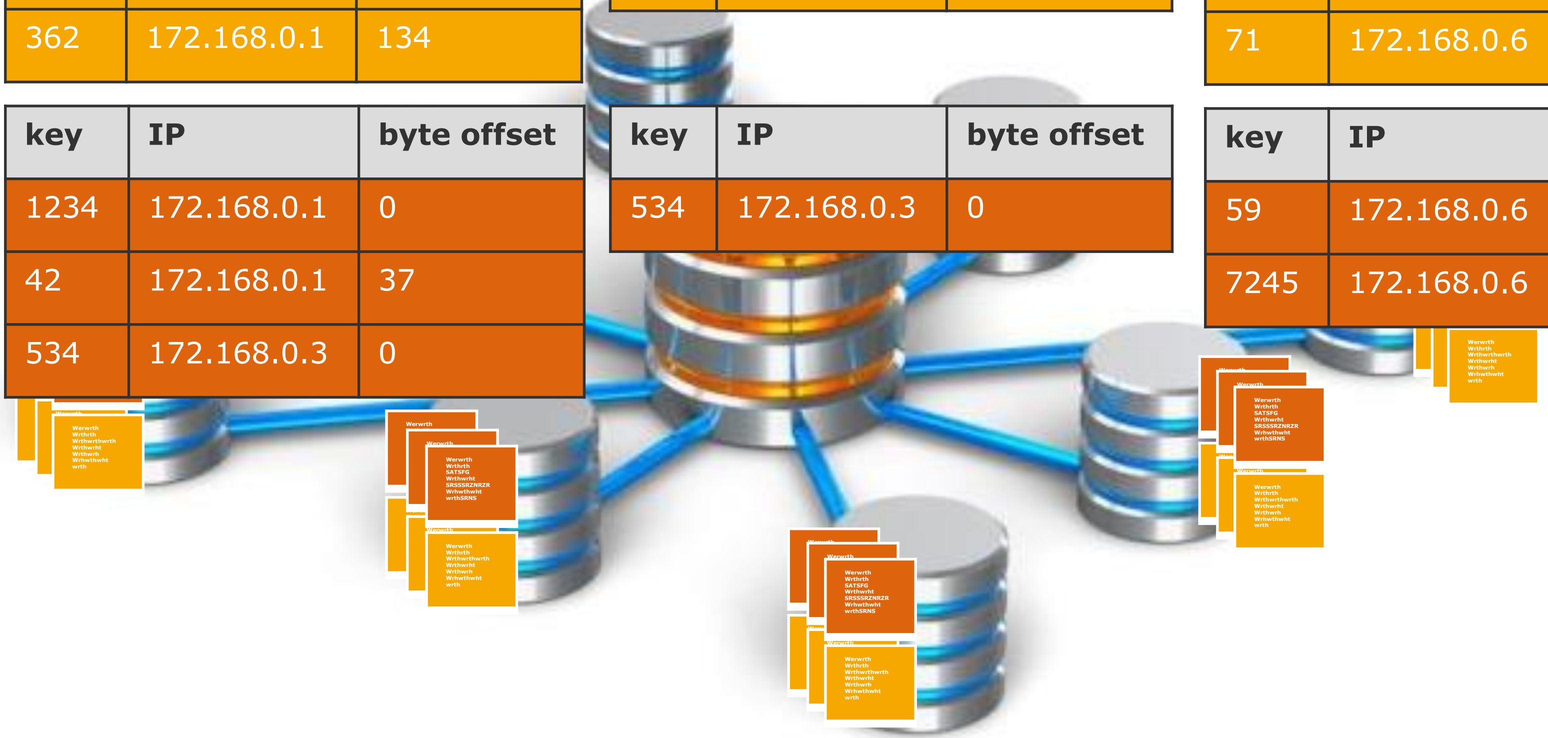
key	IP	byte offset
514	172.168.0.3	34

key	IP	byte offset
6624	172.168.0.6	256
71	172.168.0.6	325

key	IP	byte offset
1234	172.168.0.1	0
42	172.168.0.1	37
534	172.168.0.3	0

key	IP	byte offset
534	172.168.0.3	0

key	IP	byte offset
59	172.168.0.6	0
7245	172.168.0.6	52



Wait!

- If the index becomes too large and we start partitioning our index, we are back to our initial problem!
  - What is different now with segmented index data?
- Advantages:
  - Index entries are much smaller than data entries (faster compact and merge).
  - Index entries are of fixed length (will enable binary search).
  - Index could use key range partitioning while the data still uses random partitioning.
- Reality:
  - The index is usually merged with the data.

Segmentation

Index = key-value store

➤ Segmentation!

key	IP	byte offset
153	172.168.0.1	243
362	172.168.0.1	134

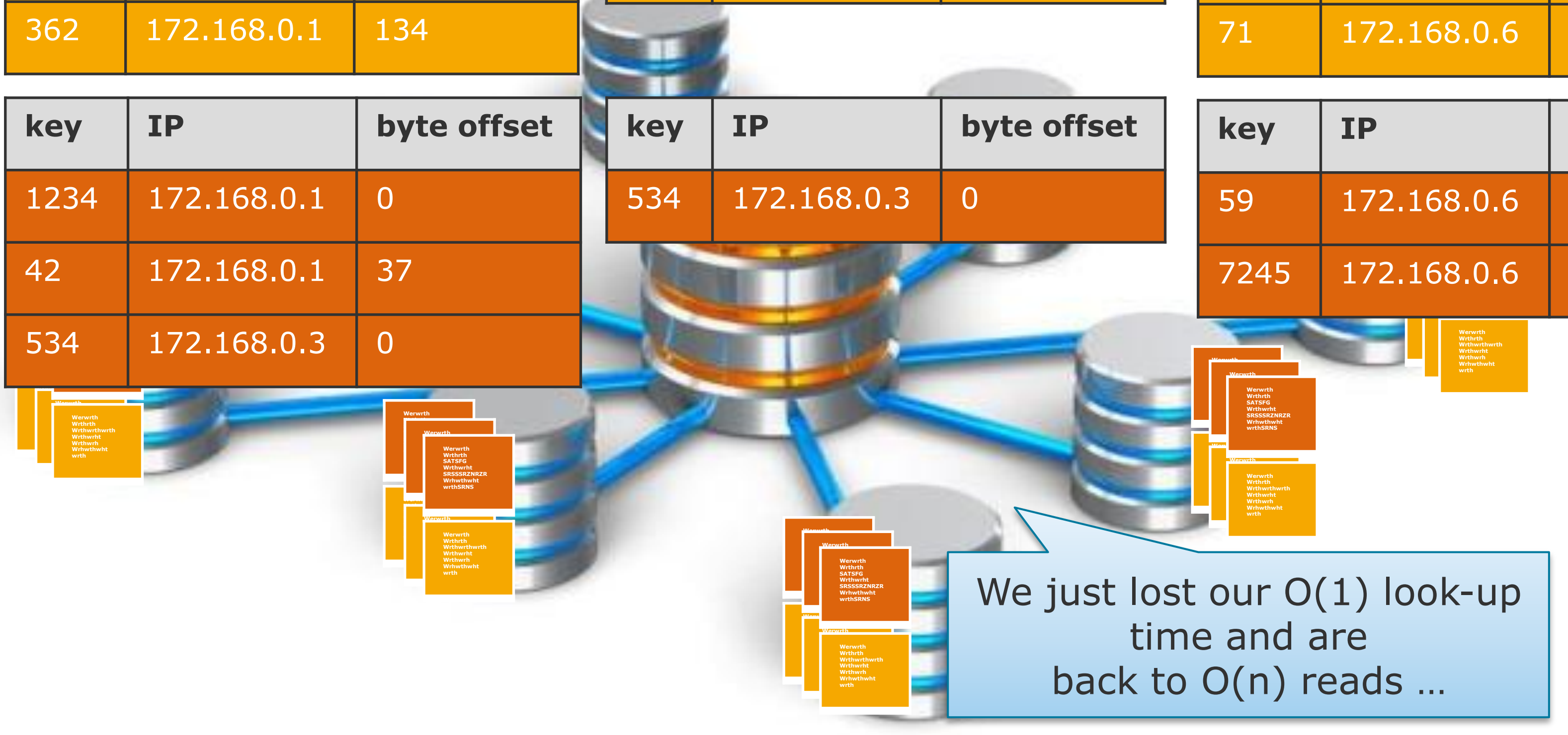
key	IP	byte offset
514	172.168.0.3	34

key	IP	byte offset
6624	172.168.0.6	256
71	172.168.0.6	325

key	IP	byte offset
1234	172.168.0.1	0
42	172.168.0.1	37
534	172.168.0.3	0

key	IP	byte offset
534	172.168.0.3	0

key	IP	byte offset
59	172.168.0.6	0
7245	172.168.0.6	52



We just lost our O(1) look-up time and are back to O(n) reads …

Overview
Objective

Design a distributed DBMS
for fast storage and retrieval
of huge and evolving datasets

Fast Retrieval DBMS

Segmentation

key	IP	byte offset
153	172.168.0.1	243
362	172.168.0.1	134

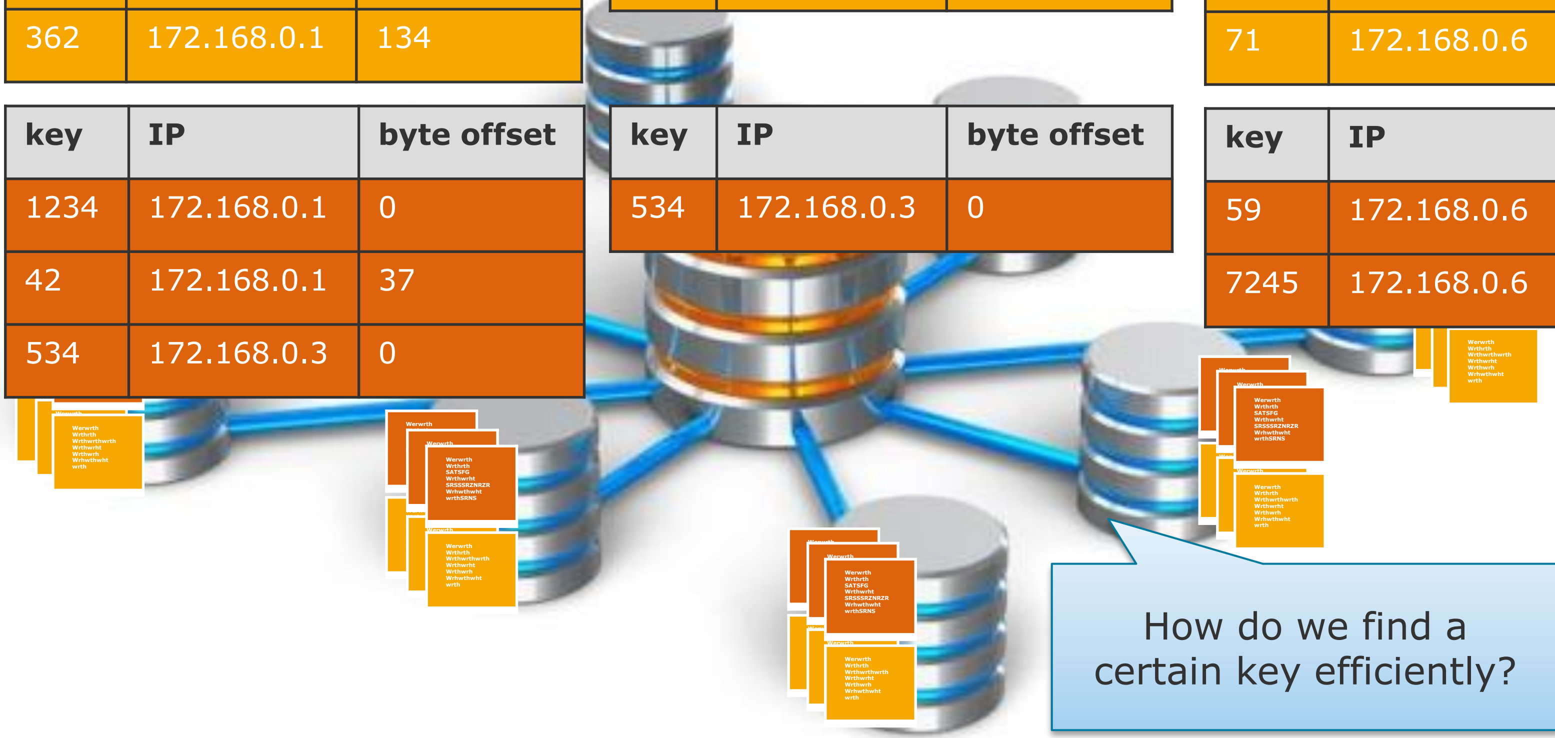
key	IP	byte offset
514	172.168.0.3	34

key	IP	byte offset
6624	172.168.0.6	256
71	172.168.0.6	325

key	IP	byte offset
1234	172.168.0.1	0
42	172.168.0.1	37
534	172.168.0.3	0

key	IP	byte offset
534	172.168.0.3	0

key	IP	byte offset
59	172.168.0.6	0
7245	172.168.0.6	52

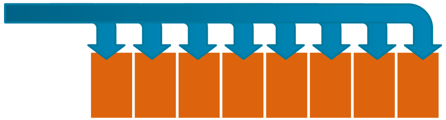


How do we find a certain key efficiently?

How do we find a certain key efficiently?

a) A dense index?

- All key-value pairs in the segment files are indexed.
- ➢ Direct look ups but index size equal to segment file size

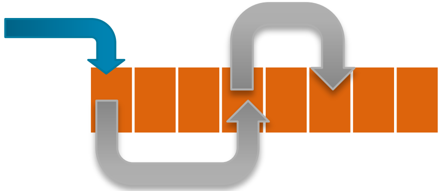


b) A sparse index?

- First key-value pair in each segment file is indexed.
- ➢ Small index but lookup still in $O(n/p)$ with p segment files

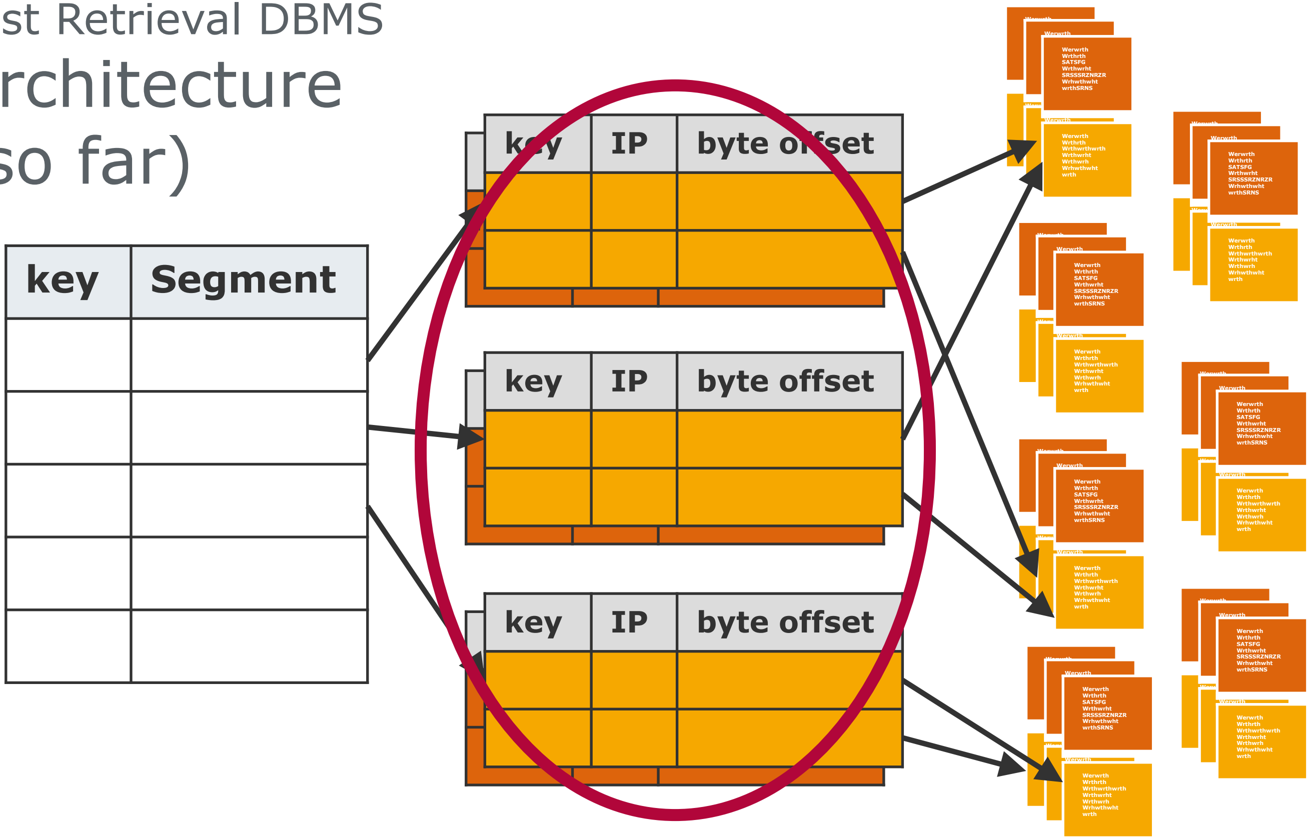
c) A sparse index + sorting?

- First key-value pair in each segment file is indexed and segment files are sorted.
- If a query key is not directly indexed:
  → find the next smaller key in the index (binary search)
  → find the segment file of the next smaller key (look up)
  → search for the query key in the block (binary search)
- ➢ Small index and lookups in $O(\log(n))$



Fast Retrieval DBMS

Architecture (so far)



Hash-index
on first key

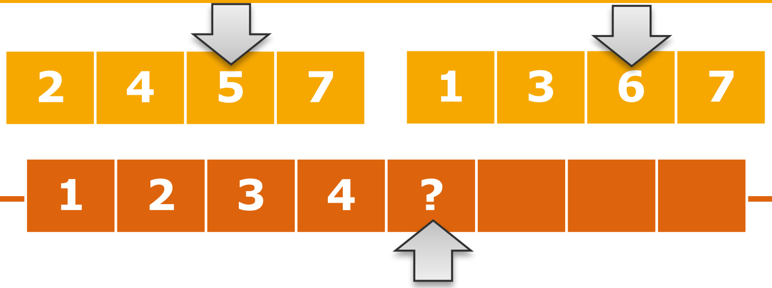
Sorted Segments
with dense pointers

Data Segments
with some partitioning and data of arbitrary length

Sorted Segments

- Current approach:
  1. All key-value pairs are sorted by their key.
     - ➢ Only pairs with larger keys can be appended:
       If a new key cannot be appended, trigger compact+merge
       with compacted segment and start a new current segment!
  2. Each key appears only once.
     - ➢ No pair with an existing key can be appended:
       If a key already exists, trigger compact+merge
       with compacted segment and start a new current segment!
  3. Key-value pairs have same length.
     - ➢ Find a key via binary search in the sorted segment
       (and its compacted sorted segment).
  - ➢ O(2 * log(n)) read performance now (with binary search),
    but we lost our O(1) write performance!

Sorted Segments



Compact + Merge

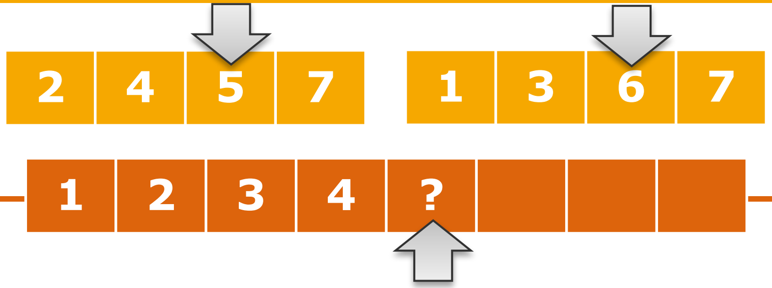
- Given two (or more) sorted segment files, their merge is calculated in linear time similarly to the merge-sort algorithm:

1. Create an empty compacted segment file.
2. Read all sorted segment files simultaneously.
3. Until all files are read entirely: Copy the smallest key with its value into the compacted segment and read the file's next key-value pair.
   - If keys are equal: Copy only the most recent key-value pair and advance both pointers.

➢ More efficient than merging general segment files,
but still too slow for random inserts of key-value pairs!

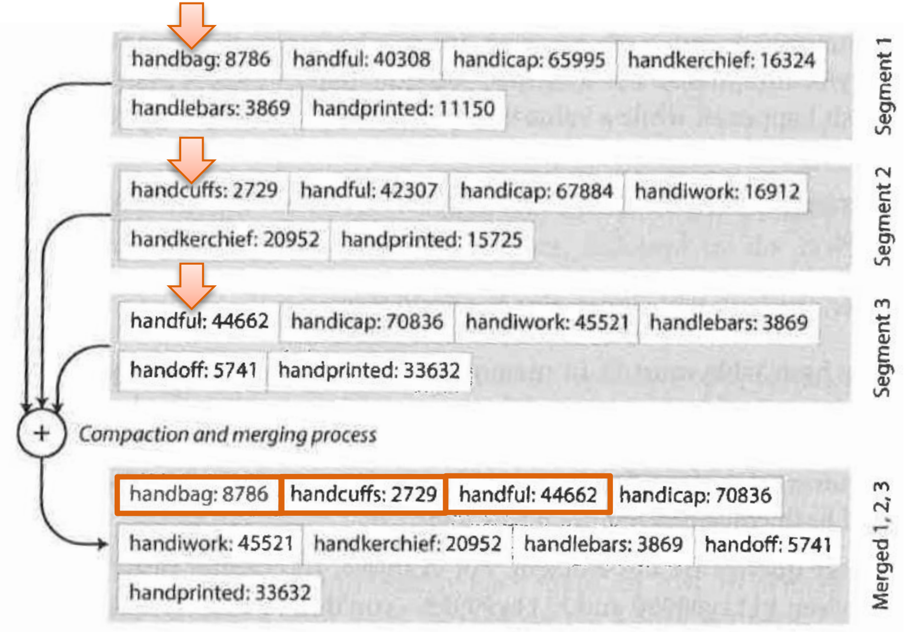
Fast Retrieval DBMS

Sorted Segments



Compact + Merge

- Given two (or more) sorted segment files, their merge is calculated in linear time similarly to the merge-sort algorithm:



SSTables

Sorted String Tables (SSTables)

- SSTables are special segment files with two properties:
  - Sorted (by their keys)
  - Immutable (hence, no appending writes)
- First introduced by Google (in BigTable and Google File System GFS).
- Divergent interpretations of this concept exist.
- Assume variable length data, i.e., no binary search!
- Structure:

Block index for the SSTable

SSTable

block1	block2	block3	block4	block5	block6	block7	block8	block9	block10
1:js 3:yy 4:pq 9:df 11:vr 12:xa	14:qd 24:ds 26:oa 30:oo	31:gw 32:yh 37:fp	39:fg 43:s m 48:ry 51:fw 52:qq	53:n m	69:ab 70:rn	71:dp 72:vg 73:as 75:yt	76:tz 77:gr 83:wt 86:rs 87:tt 89:gd 90:dy	91:us 95:ts 96:ez 98:gg 99:fj	1:1 14:2 31:3 39:4 53:5 69:6 71:7 76:8 91:9

SSTables

Sorted String Tables (SSTables)

- **Blocks**
  - Typically 64 KB
  - Are read in one disk seek operation
  - Store key-value data of any length
    (key look-up = sequential scan in memory)
- **Block index**
  - Stored in the last block of an SSTable
  - Indexes the first key of each block
  - Supports binary search

- Structure:

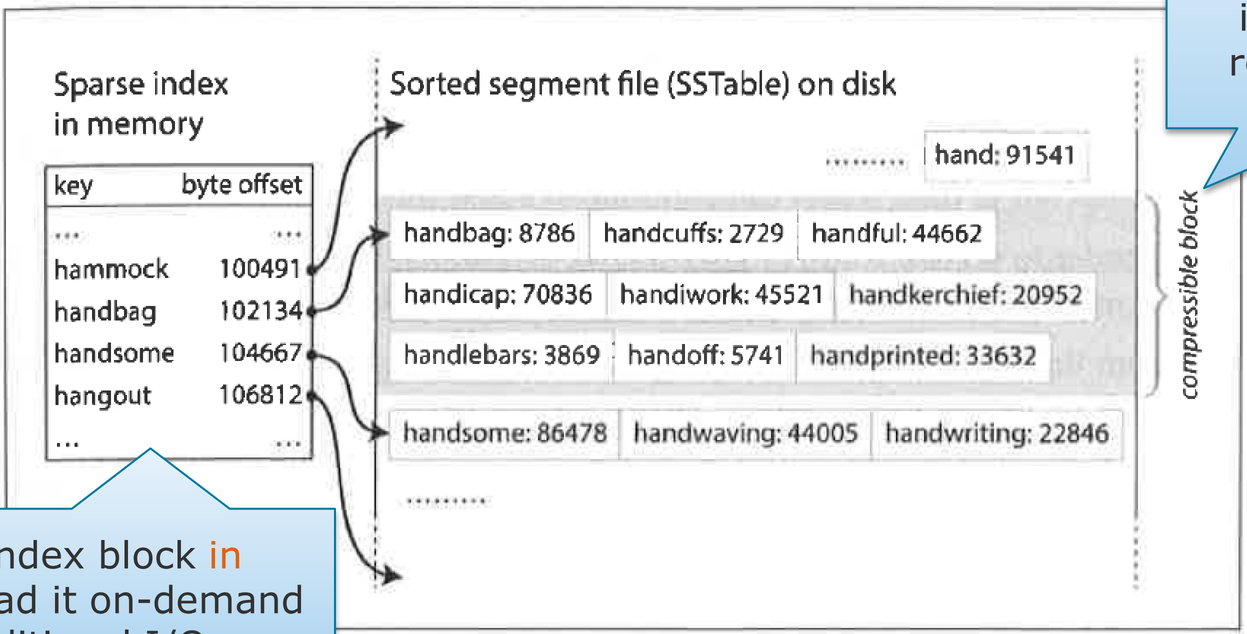
SSTable

block1	block2	block3	block4	block5	block6	block7	block8	block9	block10
1:js 3:yy 4:pq 9:df 11:vr 12:xa	14:qd 24:ds 26:oa 30:oo	31:gw 32:yh 37:fp	39:fg 43:s m 48:ry 51:fw 52:qq	53:n m	69:ab 70:rn	71:dp 72:vg 73:as 75:yt	76:tz 77:gr 83:wt 86:rs 87:tt 89:gd 90:dy	91:us 95:ts 96:ez 98:gg 99:fj	1:1 14:2 31:3 39:4 53:5 69:6 71:7 76:8 91:9

SSTables

Sorted String Tables (SSTables)

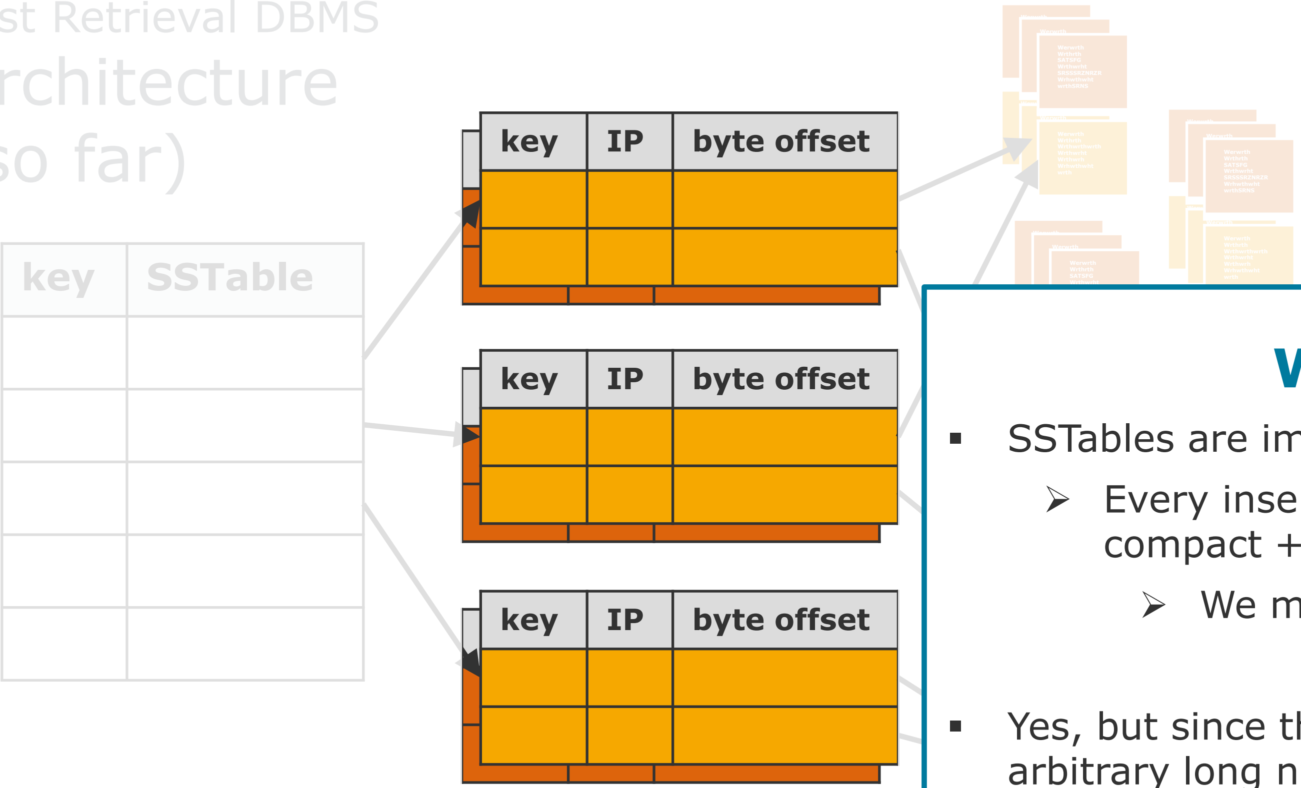
- Example:



Compressible, because blocks are immutable and read in one I/O!

Keep this index block in memory or read it on-demand in one additional I/O.

Architecture (so far)



Hash-index
on first key

SSTables
with dense pointers

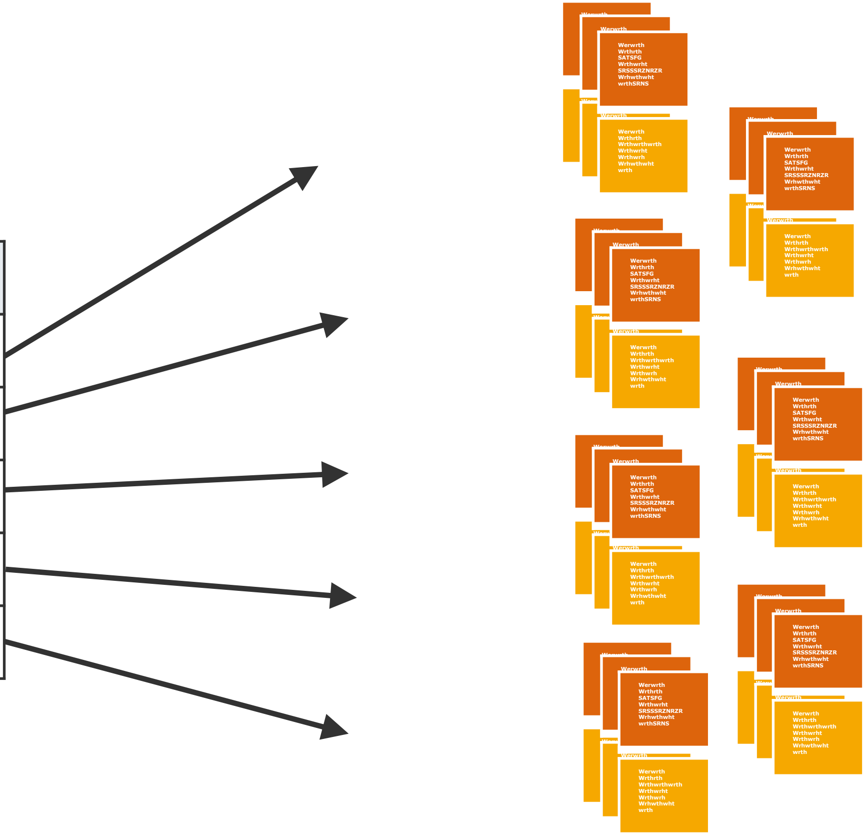
Wait!

- SSTables are immutable.
  - ➢ Every insert will trigger a compact + merge!
    - ➢ We made it worse!
- Yes, but since the values can be arbitrary long now, we can merge the Data Segments with the SSTables.
- We solve the write issue later ;-)

Fast Retrieval DBMS

Architecture (so far)

key	IP	SSTable



Hash-index
on first key

SSTables

- We solve the write issue later ;-)

Architecture (so far)

Hash-indexes are good for point queries, but can we do better for range queries?

key	IP	SSTable

Hash-index
on first key

SSTables

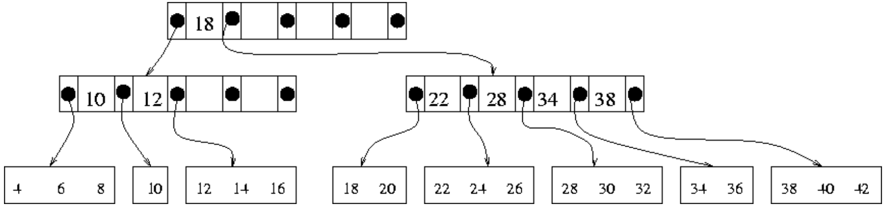
- We solve the write issue later ;-)

B-Tree

R. Bayer and E. McCreight. "Organization and maintenance of large ordered indices", In Proceedings of SIGFIDET (now SIGMOD) Workshop, pages 107-141, 1970

Definition

- A self-balancing, tree-based data structure, that stores values sorted by key and allows read/write access in logarithmic time.
- A generalization of a binary search tree as nodes can have more than two children:



Structure

- Blocks:
  - Nodes in the tree that contain key-value pairs and pointers to other blocks
  - Correspond to physical, fixed sized disk blocks/pages that are addressed and read as single units
- Pointers:
  - Edges in the tree that connect blocks in a tree structure
  - Correspond to physical block/page addresses

B-Tree

log() to a very large base;
hence, depth usually <= 3

Constraints

- Balanced:
  - Same distance from root-node to all leaf-nodes
    - ➢ Depth of the tree is in O(log(n)) (= key-look-up complexity)
    - ➢ Insert/delete procedures ensure balance
- Block-Content:
  - A block contains n keys and n+1 pointers in alternating order
  - Pointers left to a key point to blocks containing smaller keys
  - Pointers right to a key point to blocks containing larger/equal keys
    - ➢ All values in the tree are sorted by their keys!
- Block-Size:
  - Typically 4096 Byte per block; 4 Byte per key; 8 Byte per pointer
    - ➢ $4n + 8(n+1) \leq 4096 \quad => \quad n = 340$

B-Tree

Constraints

- Root Node:
  - Points to underlying nodes (and values)
  - At least 2 pointers used
- Inner Node:
  - Points to underlying nodes (and values)
  - At least $\lceil (n+1)/2 \rceil$ pointers used
- Leaf Node:
  - Points to right neighbor leaf and values
  - At least 1 neighbor pointer (if present) and $\lfloor (n+1)/2 \rfloor$ value pointers used

B-Tree vs. **B$^+$-Tree**

B-Trees store keys and values in both internal and leaf nodes;
B$^+$-Trees store values only in leaf nodes.

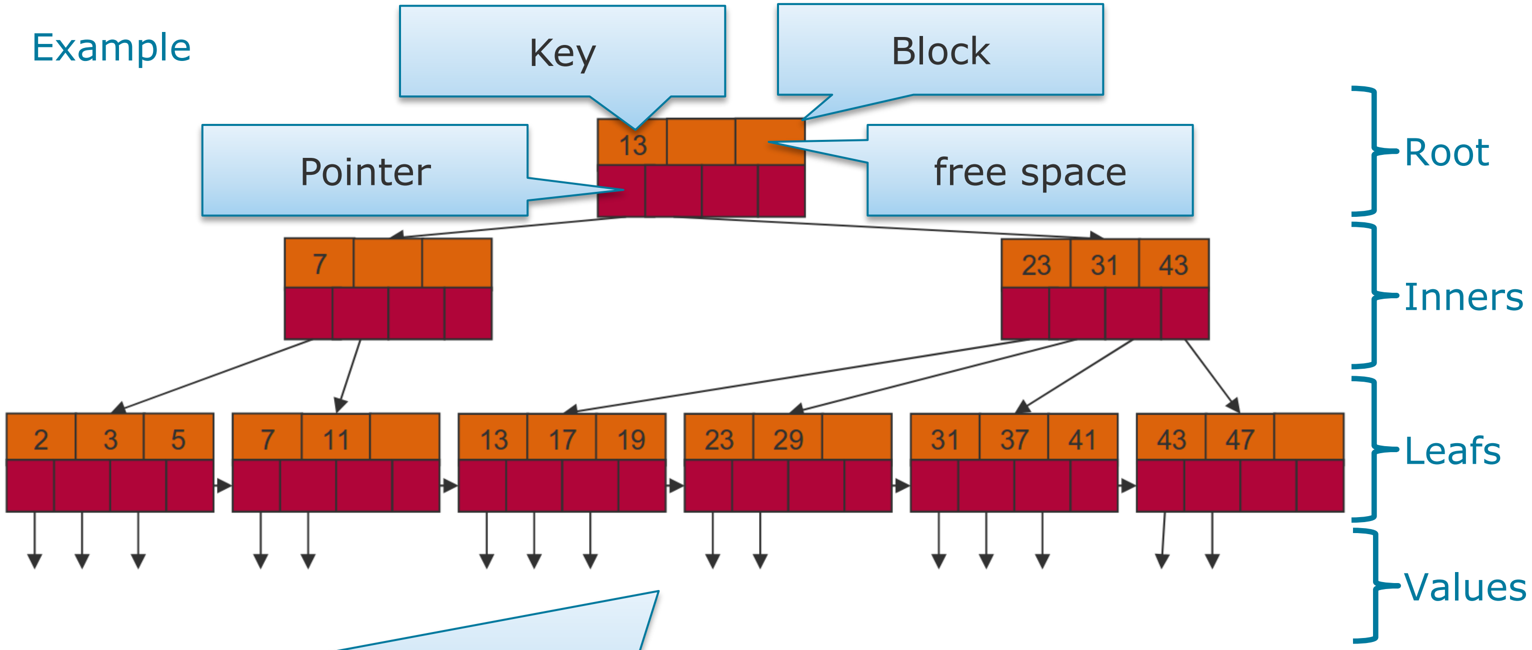
➢ The following examples show B$^+$-Trees

Uses

- Any data store: most widely used index structure for DBMSs
- Sorted, dense, and sparse indexes

B-Tree

Example

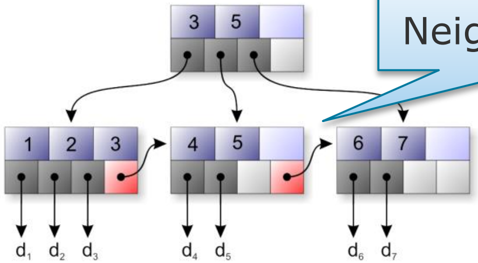


Leafs are linked!

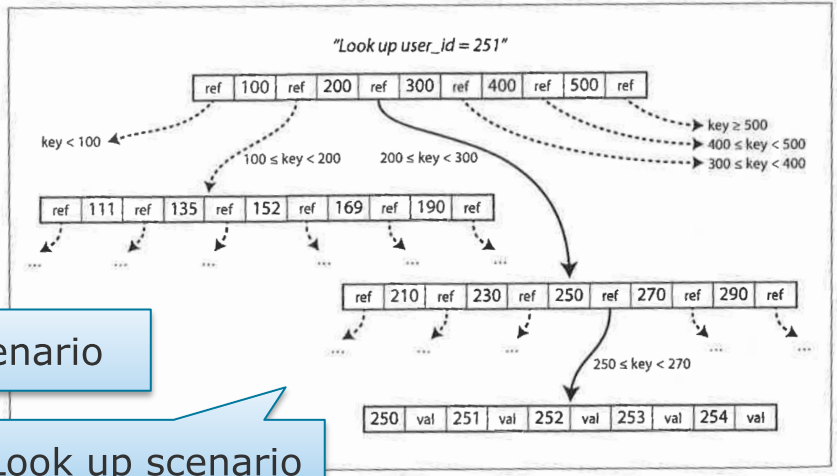
- ➢ Range query: find start of the range through the tree, then scan leafs.

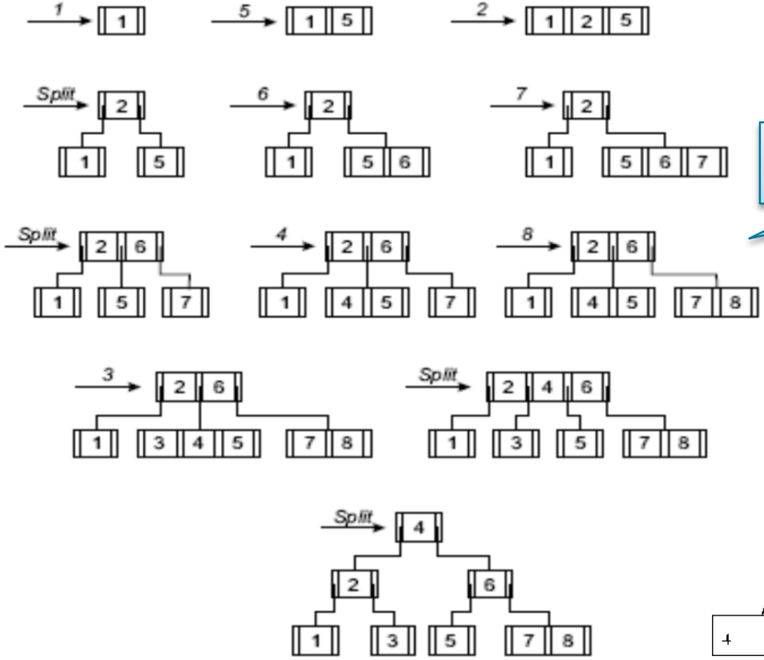
Fast Retrieval DBMS

B-Trees



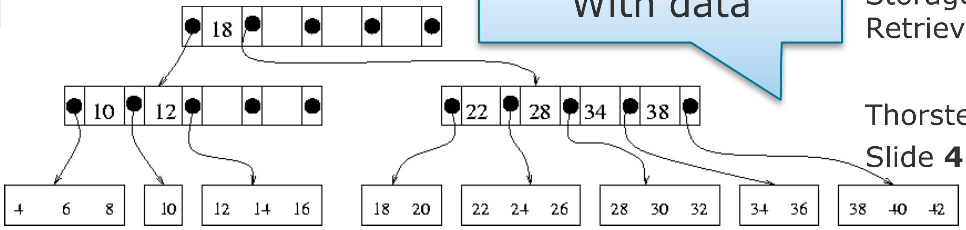
Neighbor pointer





Grow scenario

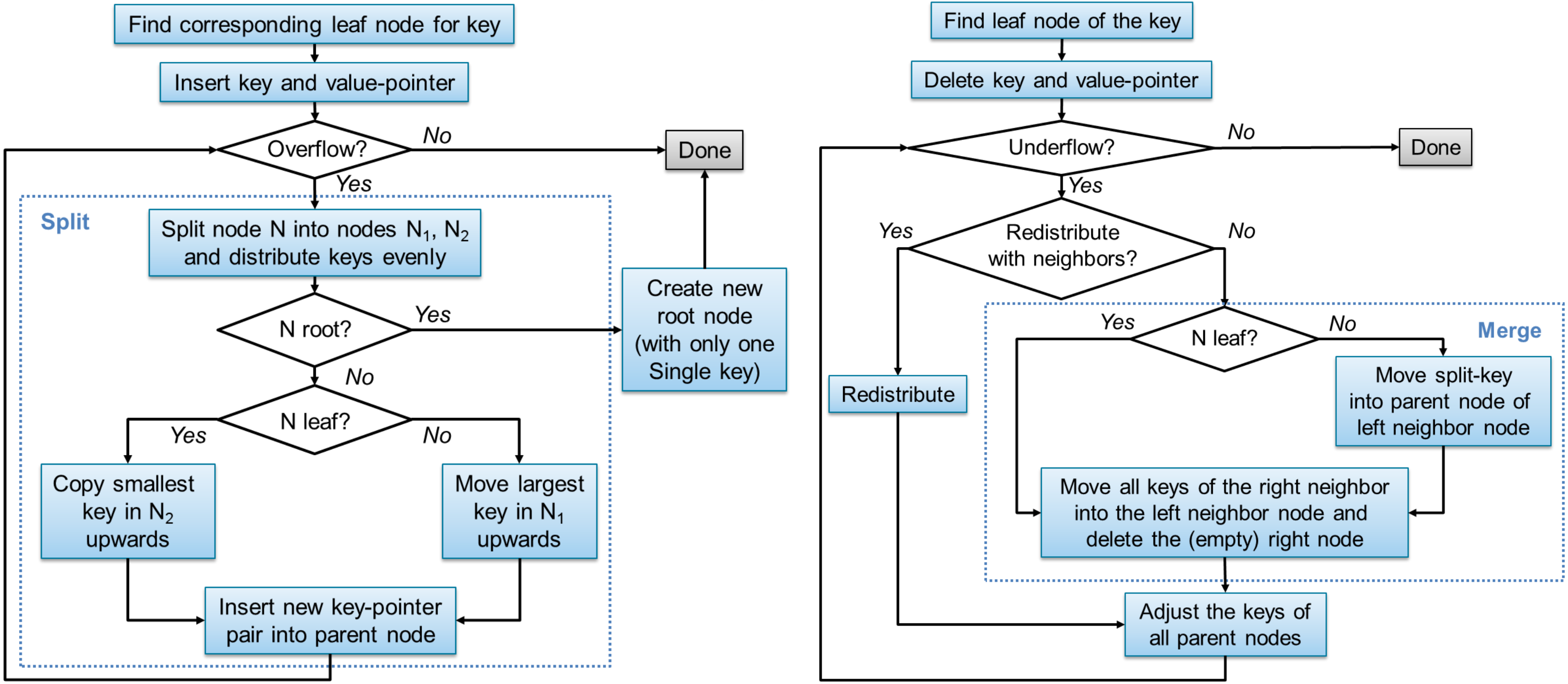
Look up scenario



With data

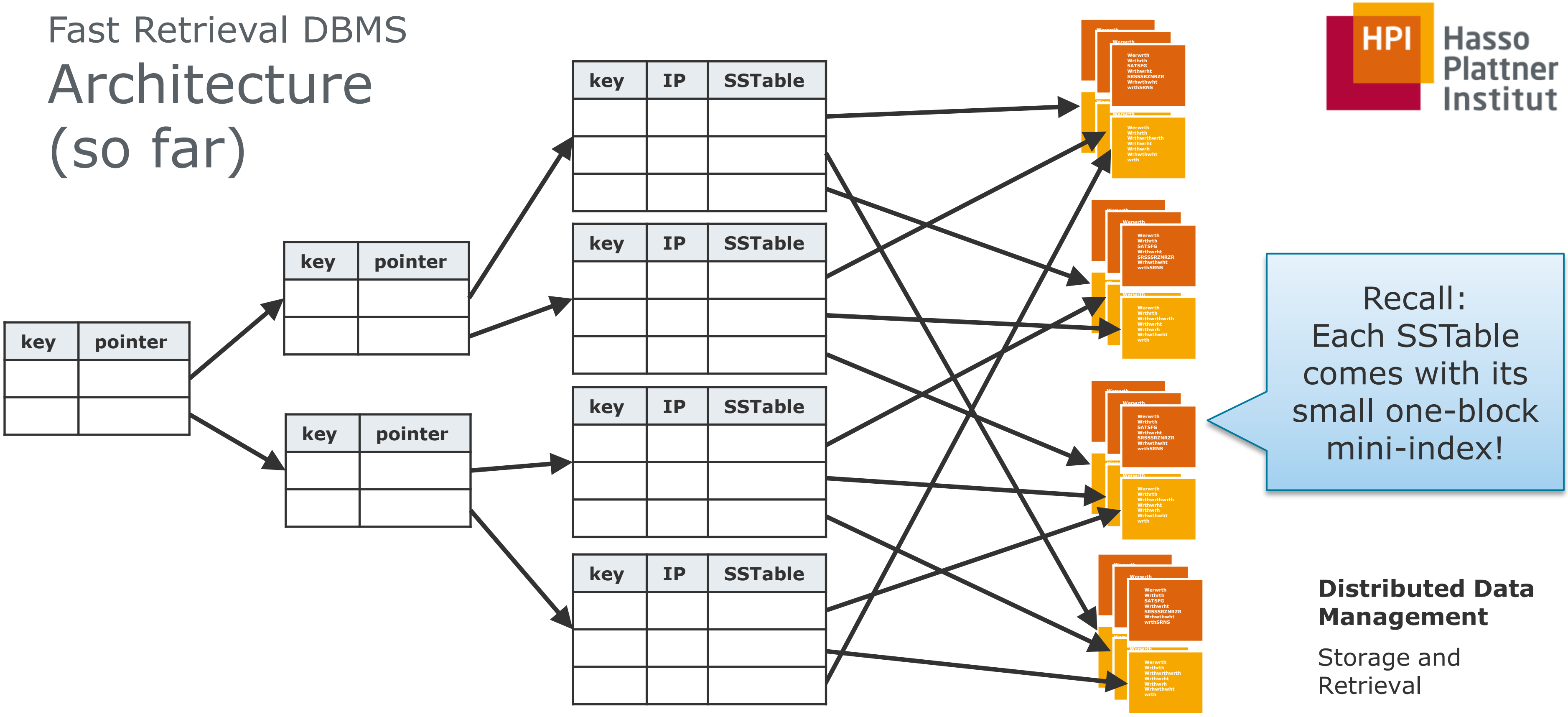
B-Tree: Insert & Delete

Split and Merge operations guarantee that the B-Tree is always balanced and the blocks are filled sufficiently.



Fast Retrieval DBMS

Architecture (so far)



B+-tree
on first key

SSTables
with data

Objective

Design a distributed DBMS
for fast storage and retrieval
of huge and evolving datasets

LSM-Trees

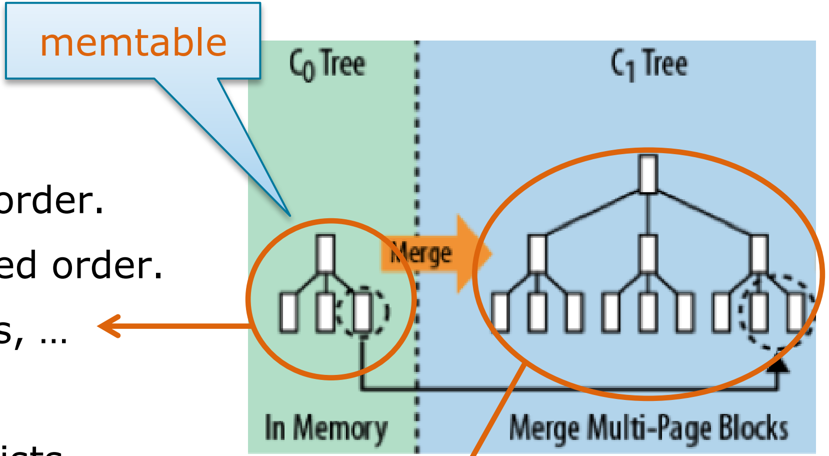
Patrick O'Neil et. al. "The log-structured merge-tree (LSM-tree)", Acta Information, volume 33, number 4, pages 351-385, 1996

Definition

- Log-Structured Merge-Trees (LSM Trees) are multilayered search trees for key-value log-data that use different data structures, each of which optimized for its underlying storage medium.

Example

- First layer (C_0 Tree): index structure that …
  1) efficiently takes new key-value pairs in any order.
  2) outputs all contained key-value pairs in sorted order.
  - B-trees, skip-lists, red-black trees, AVL trees, …
- Second layer (C_1 Tree): index structure that …
  1) is able to merge with sorted key-value pair lists.
  2) effectively compacts/compresses contained key-value pairs.
  - SSTables (+ some index structure, e.g. B-tree or block index)

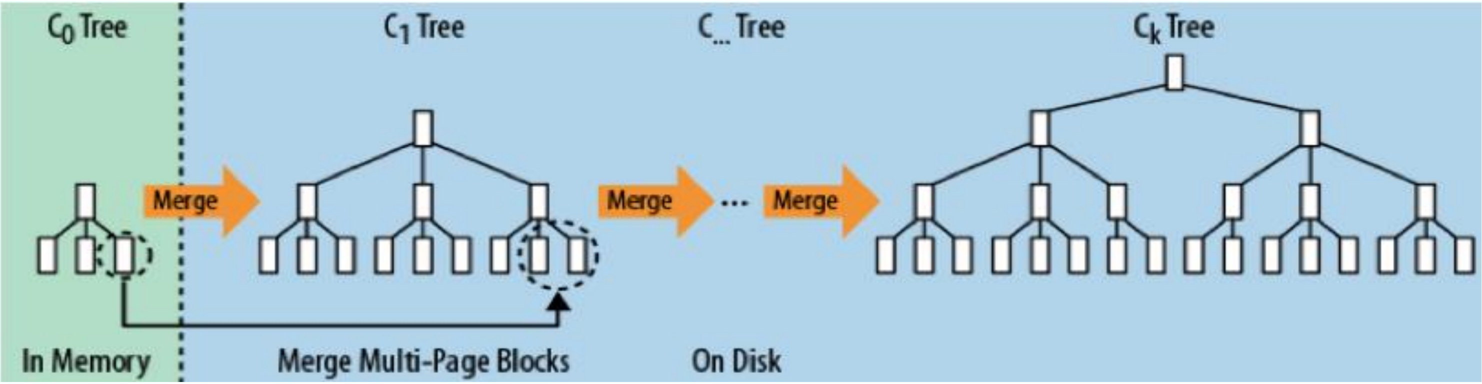


LSM-Trees

Intuition

- Sorted trees are fast in-memory indexes but they outgrow main memory.
- SSTables are indexable and compact but don't support random inserts.
  - ➢ Insert: Add new key-value pairs to C_0 Tree; frequently merge trees down the hierarchy ($C_0 \rightarrow C_1 \rightarrow C_2$ …) to free memory.
  - ➢ Read: Search the key chronologically in every layer ($C_0 \rightarrow C_1 \rightarrow C_2$ …) until the first, i.e., most recent value is found.

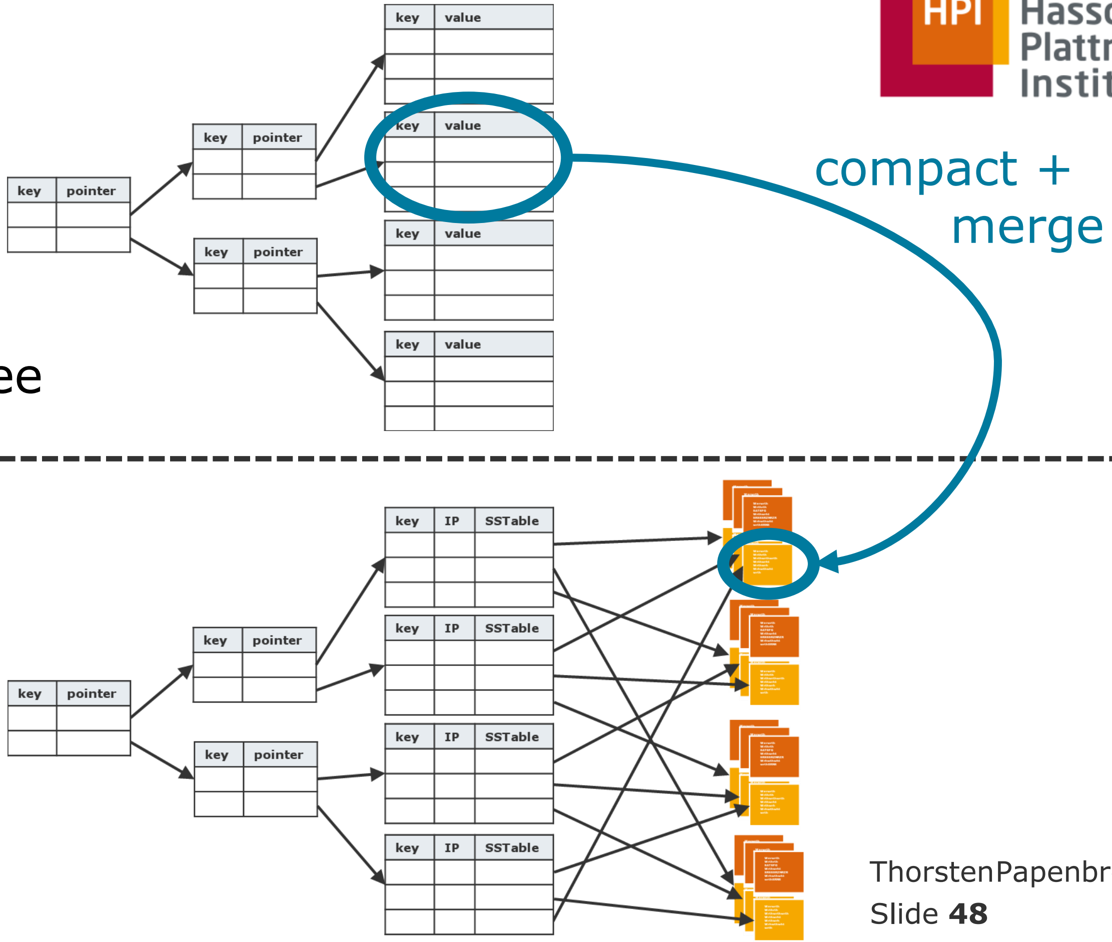
Merge is required if a block is full!



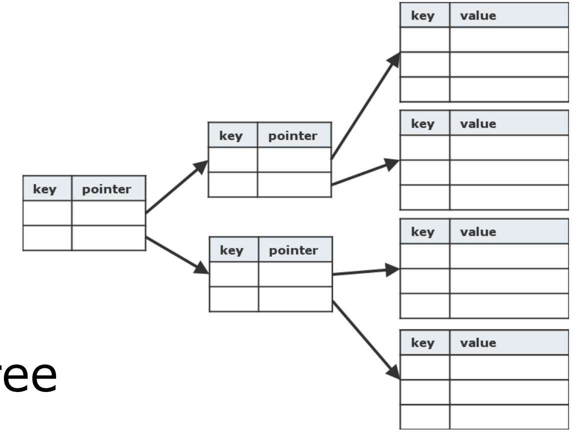
Architecture

C_0 Tree: In-memory B^+-tree

C_1 Tree: On-disk SSTables (and B^+-tree)

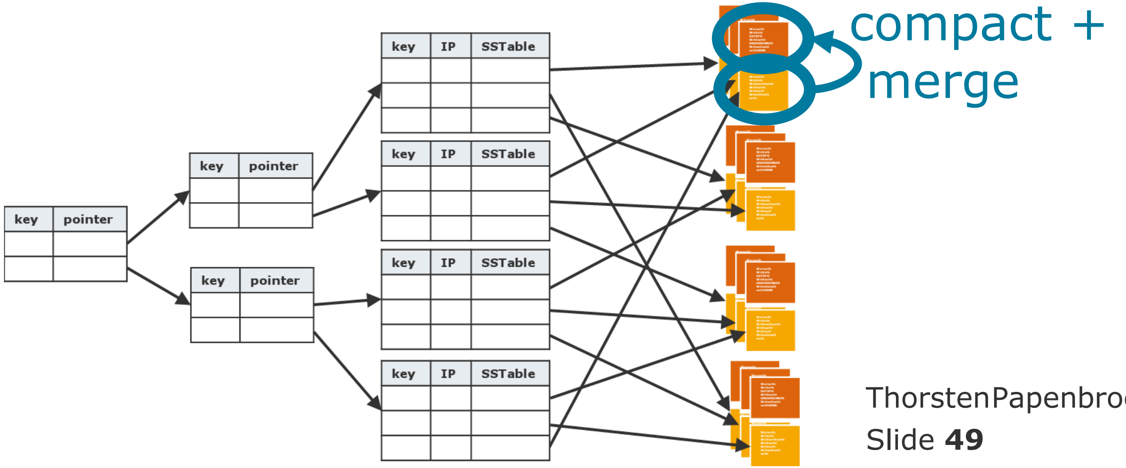


Architecture



C_0 Tree: In-memory B^+-tree

C_1 Tree: On-disk SSTables (and B^+-tree)

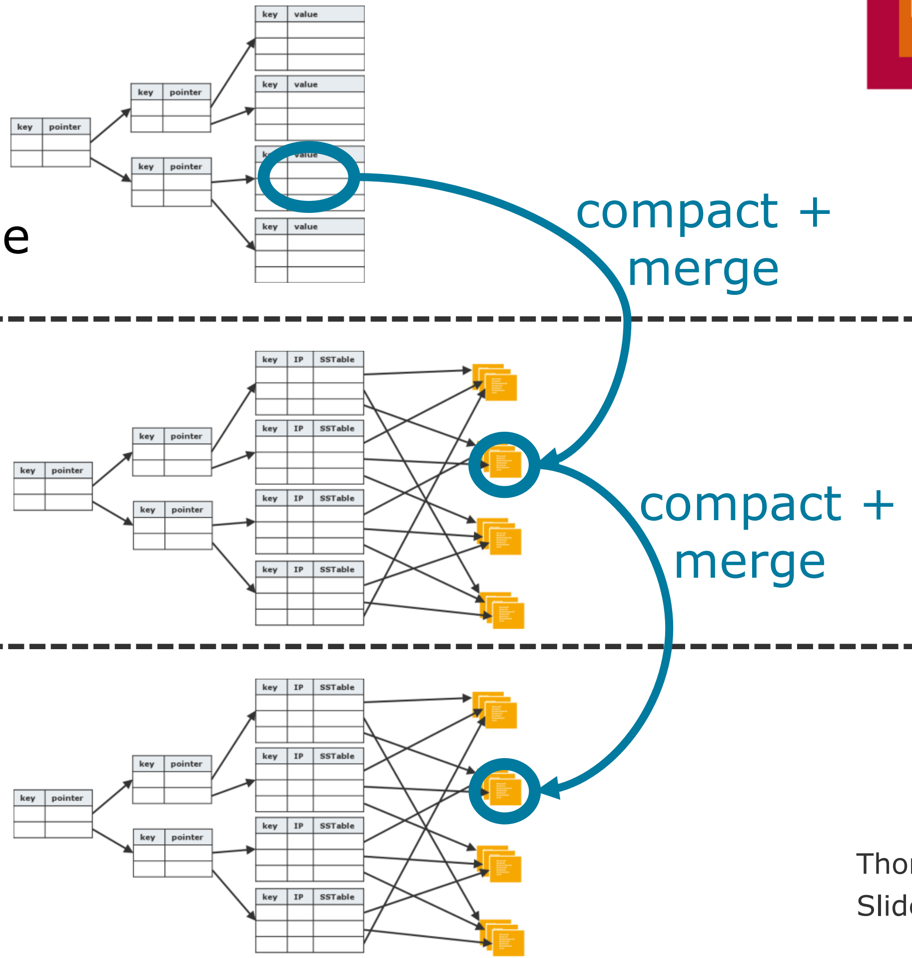


Architecture

C_0 Tree: In-memory B^+-tree

C_1 Tree: On-disk SSTables (and B^+-tree)

C_2 Tree: On-disk SSTables (and B^+-tree)

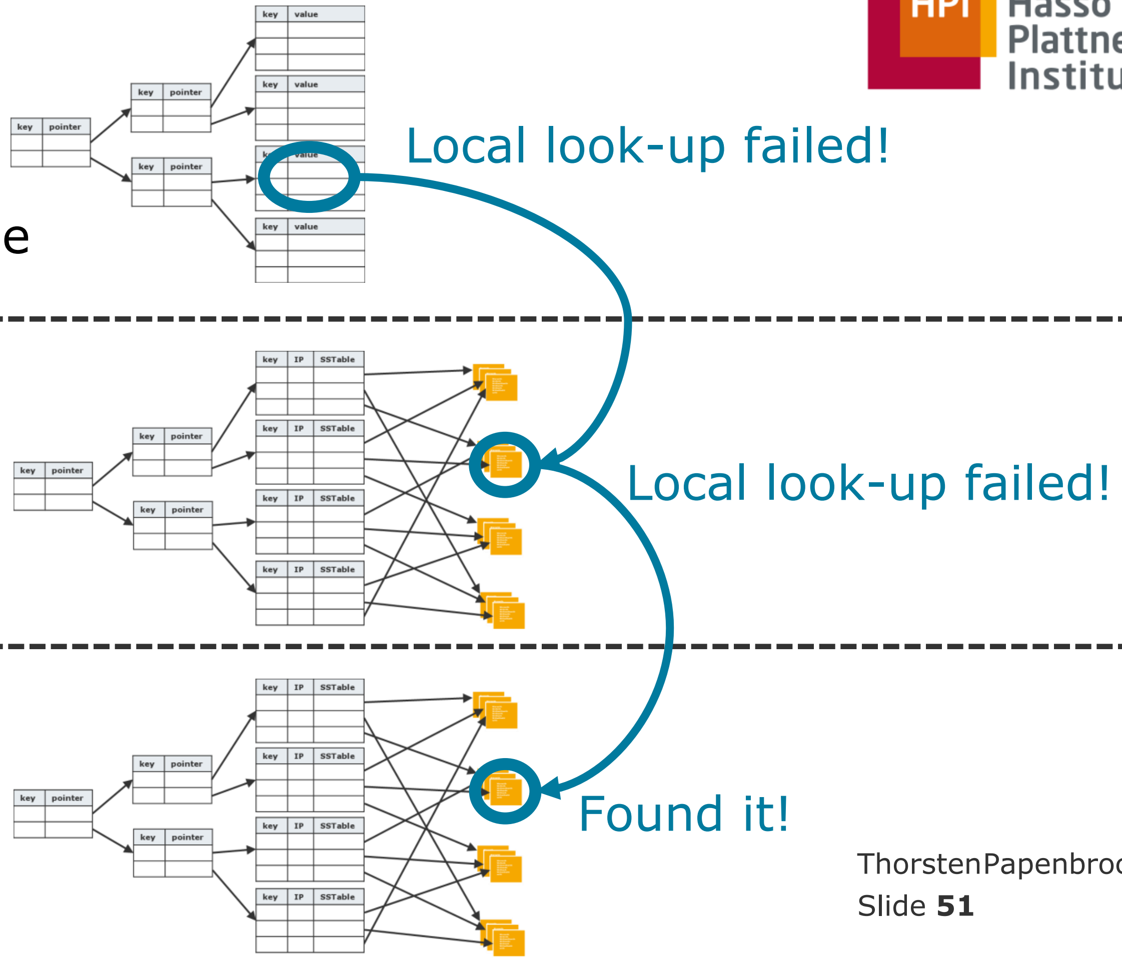


Architecture

C_0 Tree: In-memory B^+-tree

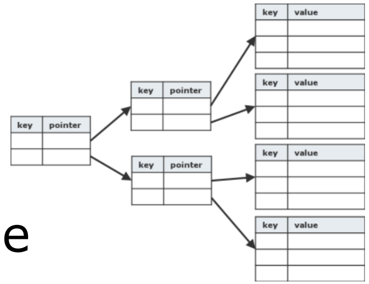
C_1 Tree: On-disk SSTables
(and B^+-tree)

C_2 Tree: On-disk SSTables
(and B^+-tree)

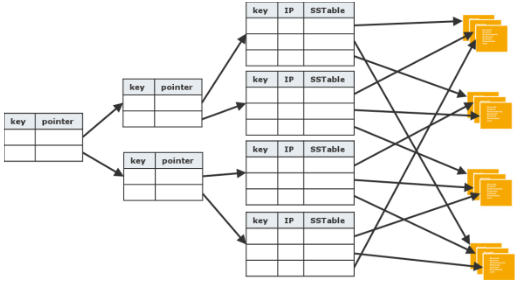


Architecture

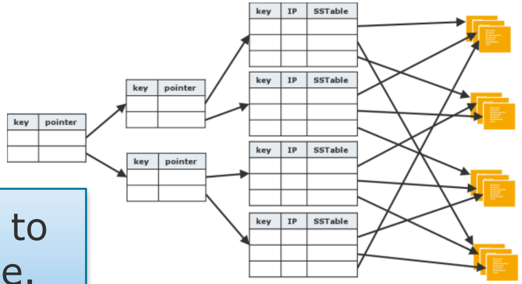
C_0 Tree: In-memory B^+-tree



C_1 Tree: On-disk SSTables (and B^+-tree)



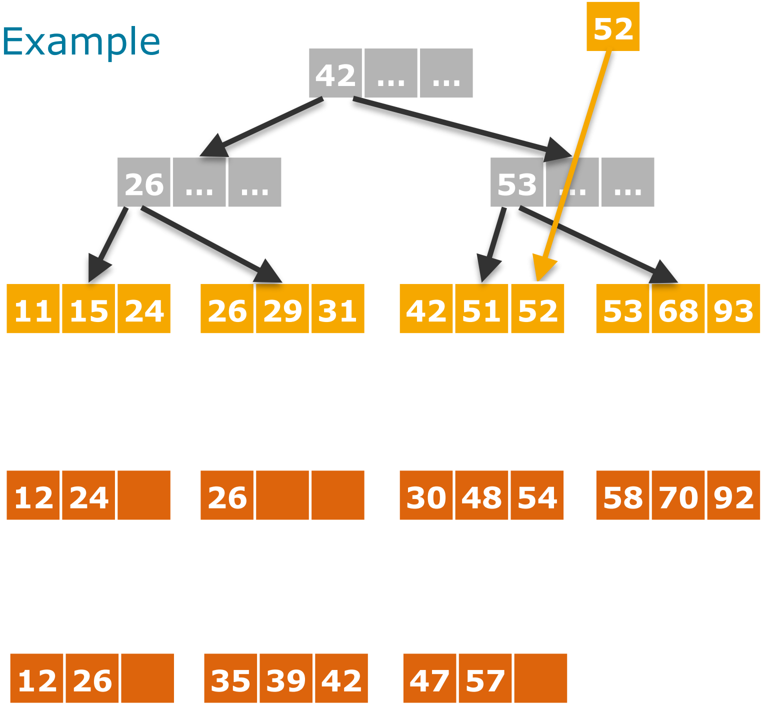
C_2 Tree: On-tape SSTables (and B^+-tree)



Can use trees for distribution and to move older data to slower storage.

LSM-Tree Example: B⁺-Tree & SSTables

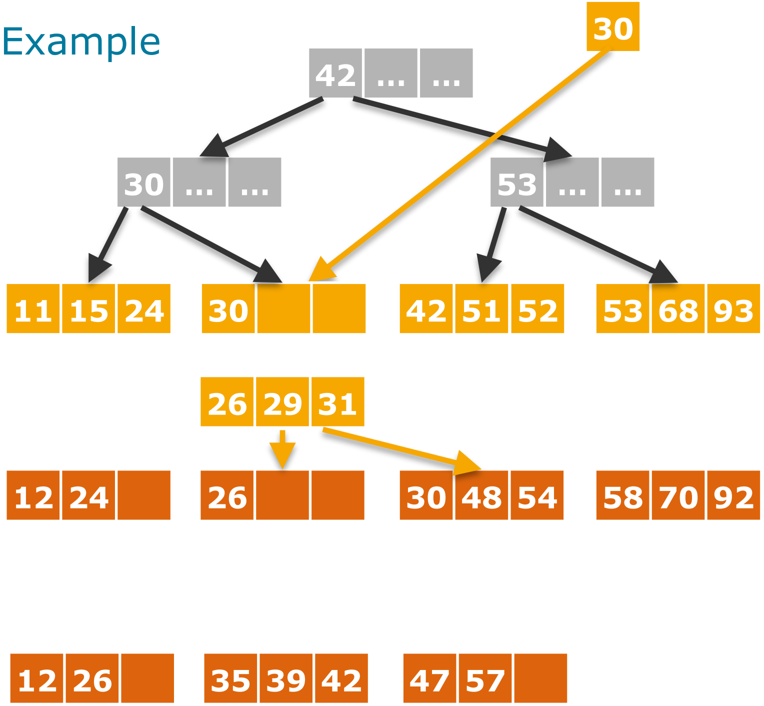
Example



- Insert everything into the B⁺-tree first.
- Depth of the tree is fix.

LSM-Tree Example: B+-Tree & SSTables

Example

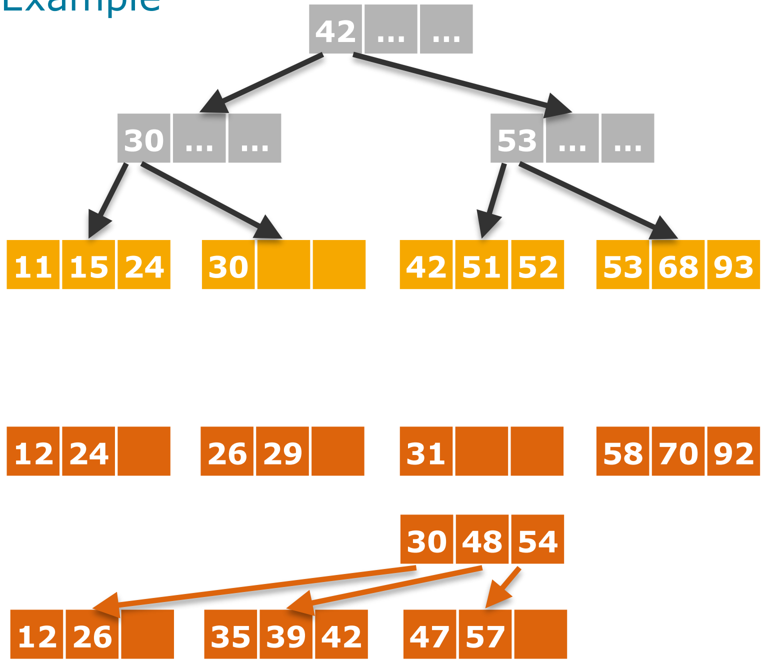


- Insert everything into the B+-tree first.
- Depth of the tree is fix.
- If leaf is full:
  1. Re-assign keys?
  2. Split without increasing depth over max?
  3. Merge leaf into C_1's SSTables.
- Merge:
  - Find SSTable that would take the first key of the leaf.
  - Start merging that SSTable with the leaf.
  - If current leaf key >= start key of next SSTable:
    - Continue merge with that SSTable.

For this example:
Assume all inner nodes are full
and no redistribution possible.

LSM-Tree Example: B⁺-Tree & SSTables

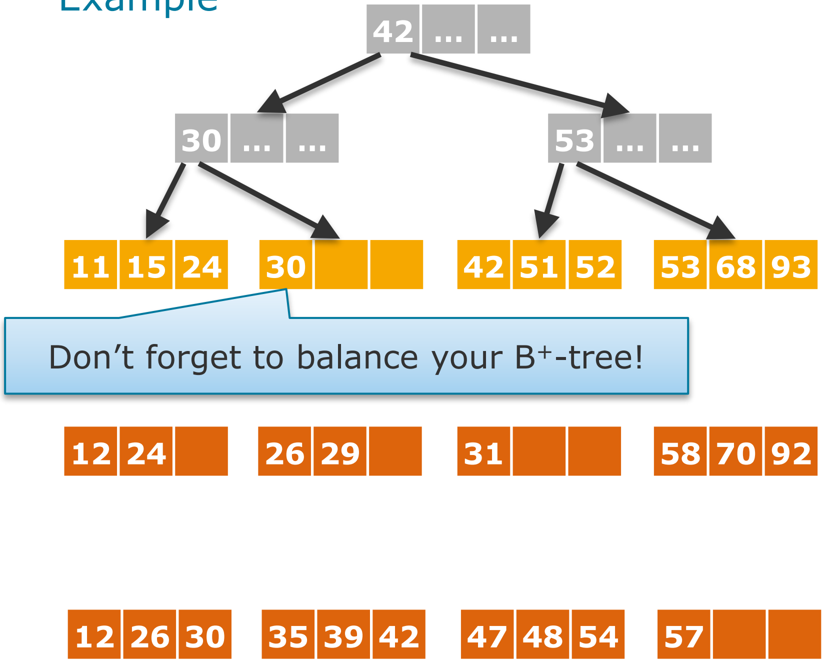
Example



- Insert everything into the B⁺-tree first.
- Depth of the tree is fix.
- If leaf is full:
  1. Re-assign keys?
  2. Split without increasing depth over max?
  3. Merge leaf into C_1's SSTables.
- Merge:
  - Find SSTable that would take the first key of the leaf.
  - Start merging that SSTable with the leaf.
  - If current leaf key >= start key of next SSTable:
    - Continue merge with that SSTable.
  - If some SSTable gets full:
    - Merge that SSTable down the hierarchy.
    - If no further level exists:
      - Split the SSTable.

LSM-Tree Example: B⁺-Tree & SSTables

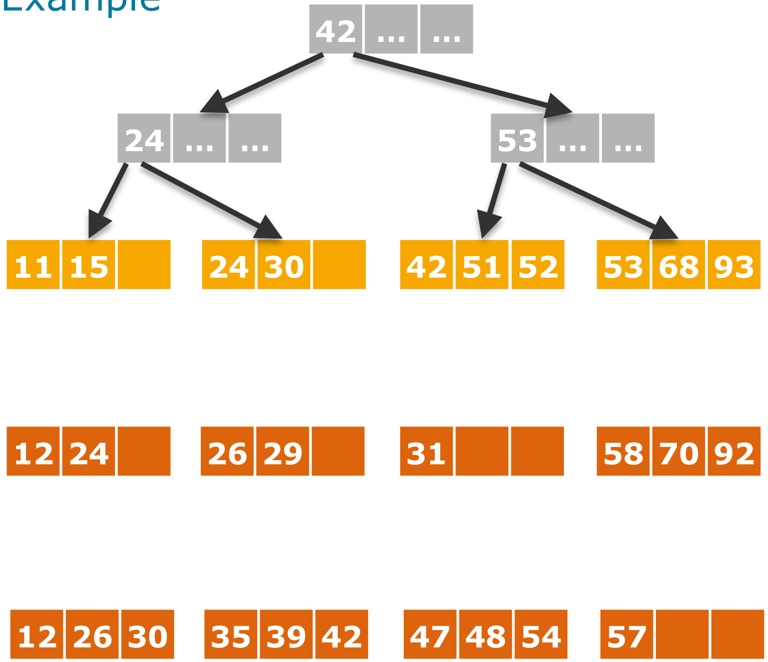
Example



- Insert everything into the B⁺-tree first.
- Depth of the tree is fix.
- If leaf is full:
  1. Re-assign keys?
  2. Split without increasing depth over max?
  3. Merge leaf into C_1's SSTables.
- Merge:
  - Find SSTable that would take the first key of the leaf.
  - Start merging that SSTable with the leaf.
  - If current leaf key >= start key of next SSTable:
    - Continue merge with that SSTable.
  - If some SSTable gets full:
    - Merge that SSTable down the hierarchy.
    - If no further level exists:
      - Split the SSTable.

LSM-Tree Example: B⁺-Tree & SSTables

Example



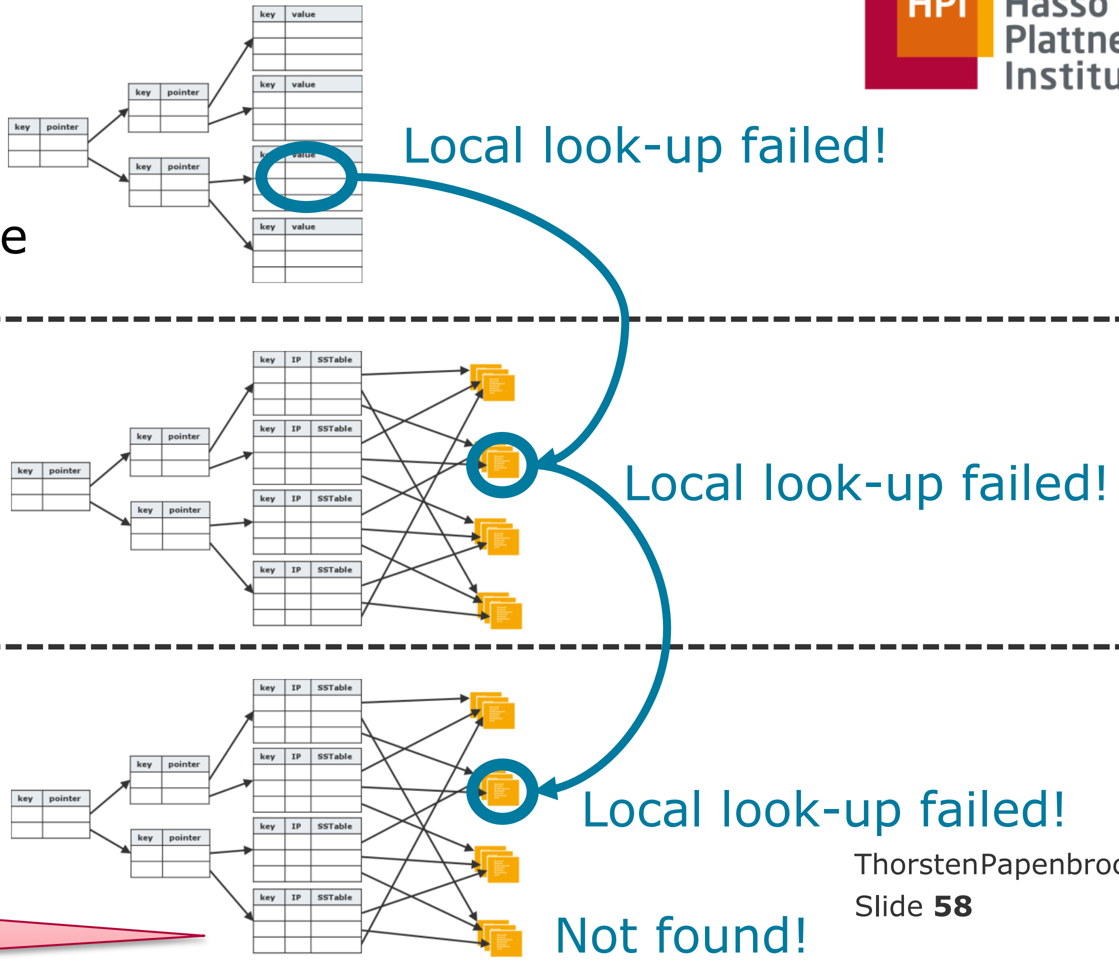
- Insert everything into the B⁺-tree first.
- Depth of the tree is fix.
- If leaf is full:
  1. Re-assign keys?
  2. Split without increasing depth over max?
  3. Merge leaf into C_1's SSTables.
- Merge:
  - Find SSTable that would take the first key of the leaf.
  - Start merging that SSTable with the leaf.
  - If current leaf key >= start key of next SSTable:
    - Continue merge with that SSTable.
  - If some SSTable gets full:
    - Merge that SSTable down the hierarchy.
    - If no further level exists:
      - Split the SSTable.

Architecture

C_0 Tree: In-memory B^+-tree

C_1 Tree: On-disk SSTables (and B^+-tree)

C_2 Tree: On-disk SSTables (and B^+-tree)



Looking up non existing keys is super expensive!

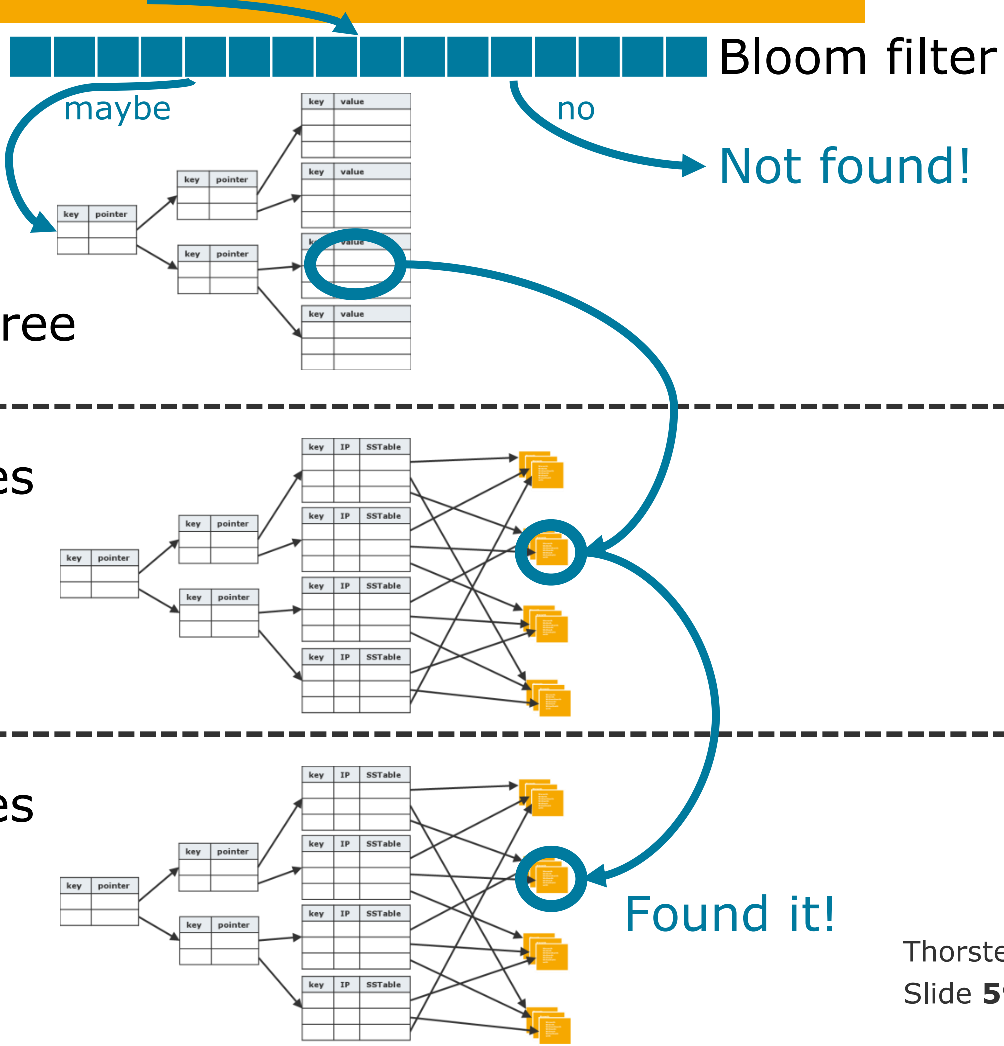
Fast Storage and Retrieval

Architecture

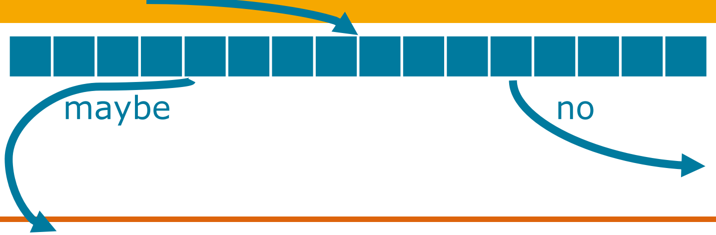
C_0 Tree: In-memory B^+-tree

C_1 Tree: On-disk SSTables (and B^+-tree)

C_2 Tree: On-disk SSTables (and B^+-tree)



Bloom filter



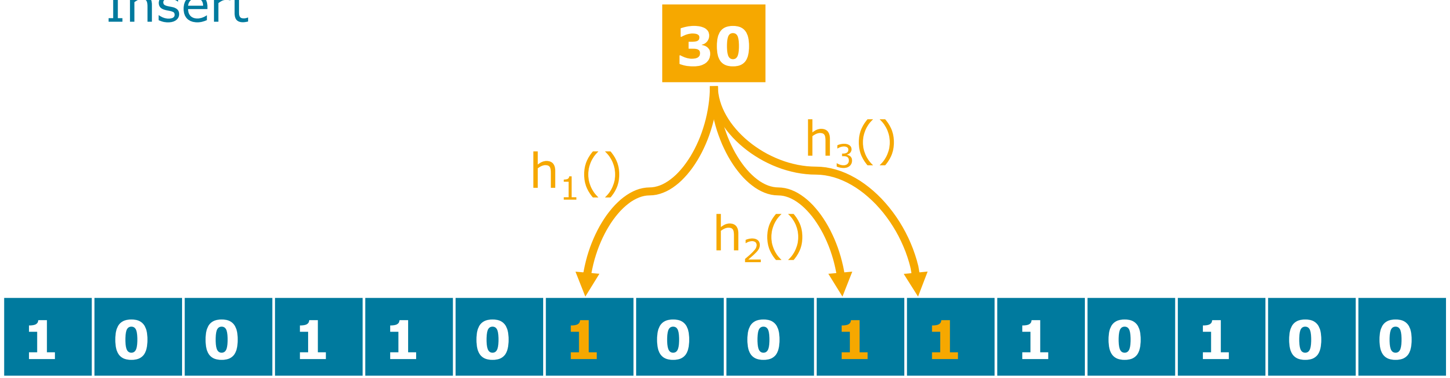
A Bloom filter is a probabilistic data structure that answers set containment questions in constant time and with constant memory consumption.

- "Does element X appear in the set?"
- Answer "no" is guaranteed to be correct.
- Answer "yes" has a certain probability to be wrong (hence, "maybe").
  - ➢ But then the concrete look-up will just fail.
  - ➢ Very nice property that allows the use of Bloom filters in exact systems.
- Structure
  - Bitset of fixed size (typically a long array)
  - One (or more) hash functions

Burton H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", Communications of the ACM, volume 13, number 7, pages 422-426, 1970

Bloom filter

Insert

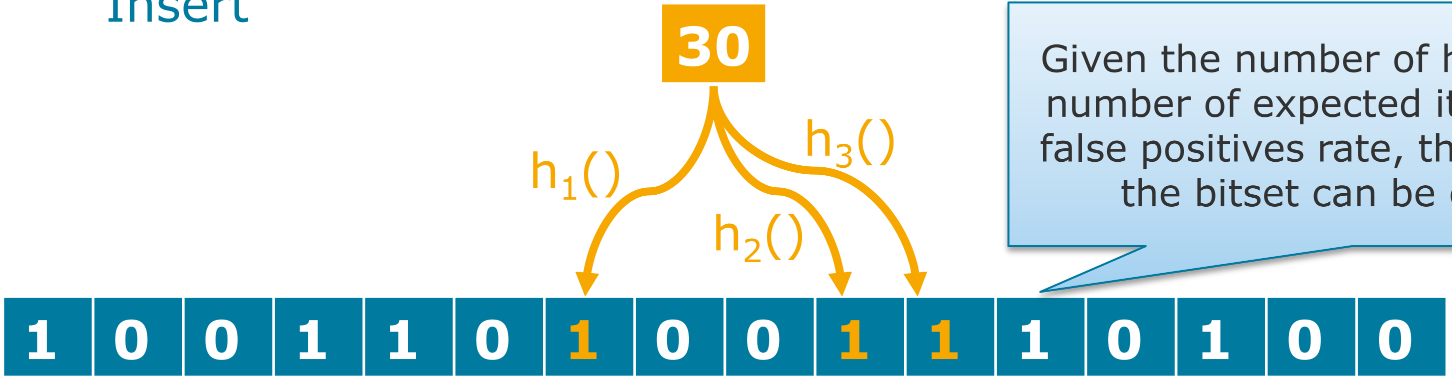


Hash functions: $h_1()$, $h_2()$, $h_3()$

- A hash function hashes the key to one bit in the bitset.
- The Bloom filter implementation can use one or multiple functions.
  - Trade-off: More functions reduce the probability of hash collisions but they also exhaust the bitset faster, which produces more collisions later.

Bloom filter

Insert



Given the number of hash functions, the number of expected items, and a target false positives rate, the minimum size of the bitset can be calculated [1].

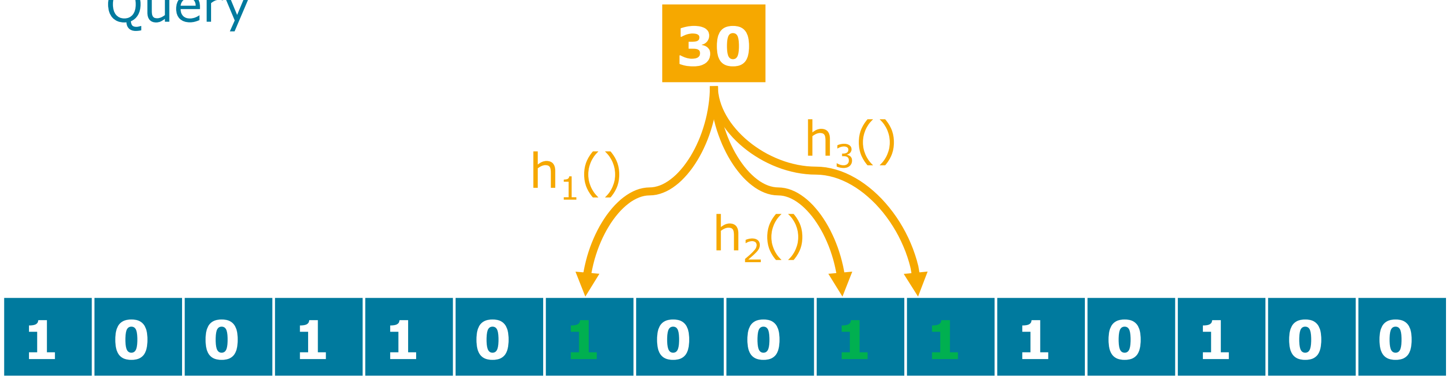
Bitset

- Fixed array of bits.
- Increasing the array size decreases the probability of hash collisions especially when multiple hash functions are used.

[1] https://en.wikipedia.org/wiki/Bloom_filter#CITEREFBloom1970

Bloom filter

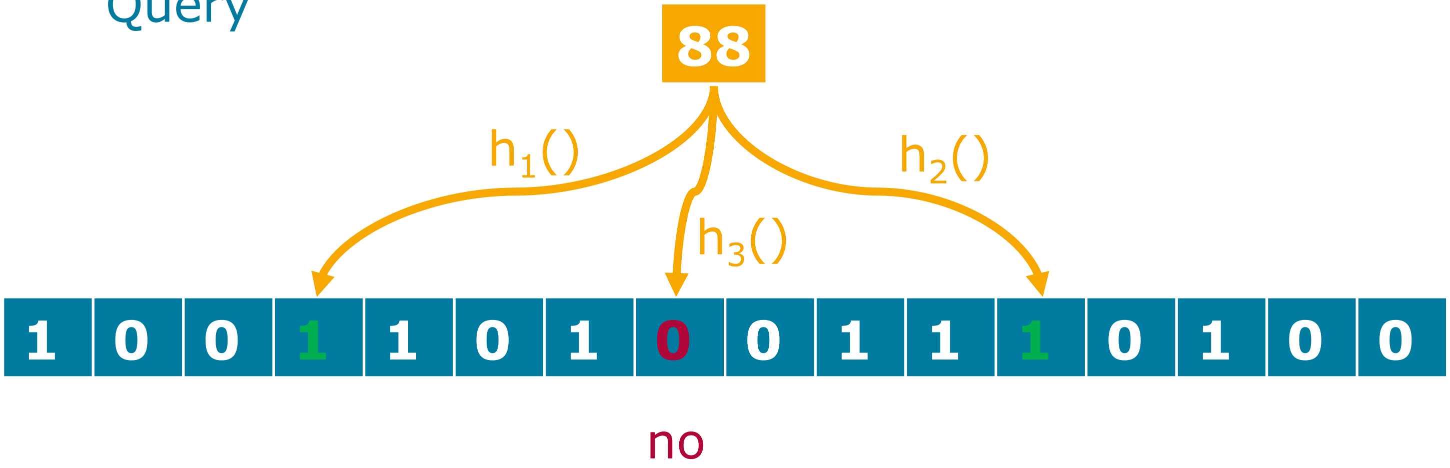
Query



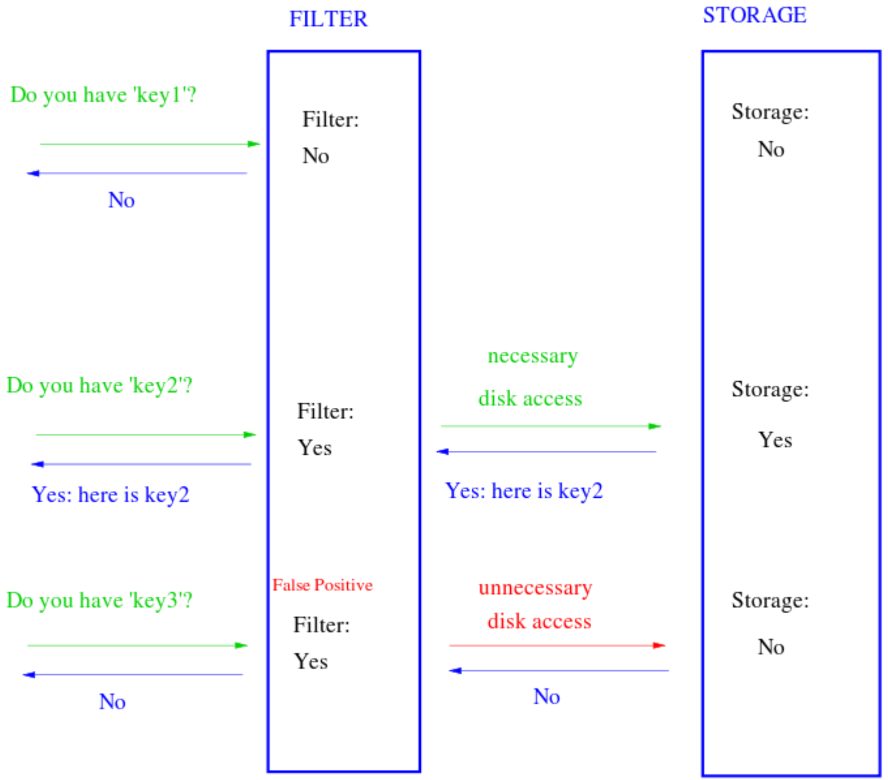
maybe

Bloom filter

Query



Bloom filter



https://en.wikipedia.org/wiki/Bloom_filter#CITEREFBloom1970

LSM-Trees

Optimizations

Querying non-existend values is expensive (check all layers)

➢ Catch most of these queries with a Bloomfilter

Size-tiered compaction

➢ Merge newer, smaller SSTables successively into older, larger SSTables

Objective

Design a distributed DBMS
for fast storage and retrieval
of huge and evolving datasets

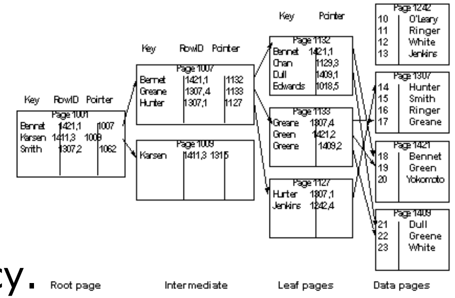
Some further indexing-techniques ...

Excursus

Alternative Index Types

Clustered Index with Data (see LSM-Trees)

- Stores indexed data or parts of it within the index (plus/instead of pointers to data)
- Example: An index on attribute `delivery_status` allows to count pending deliveries without data access.
  - ➢ Improves the performance of certain queries.
  - ➢ Might reduce write performance and require additional storage.
  - ➢ Redundant values (in data and index) complicate data consistency.

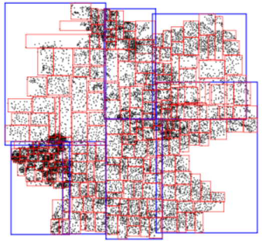


Multi-Column Index

a) Concatenated index: Merge keys into one key.

b) Multi-dimensional index: Split multi-dim. key domain into multi-dim. shapes.

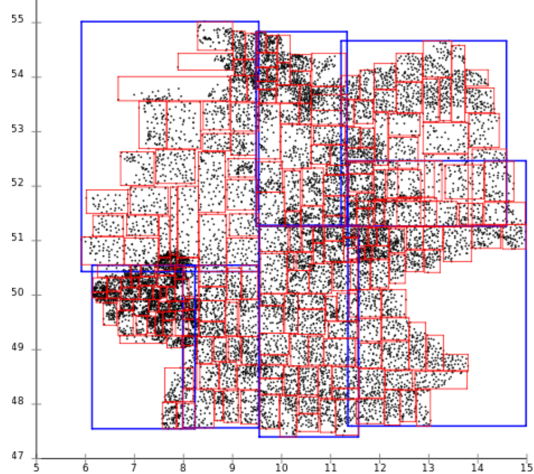
- Example: An index on two-dim. geo locations (`longitude`, `latitude`) to answer intersection, containment, and nearest neighbor queries.
- Most common implementation: R-Trees

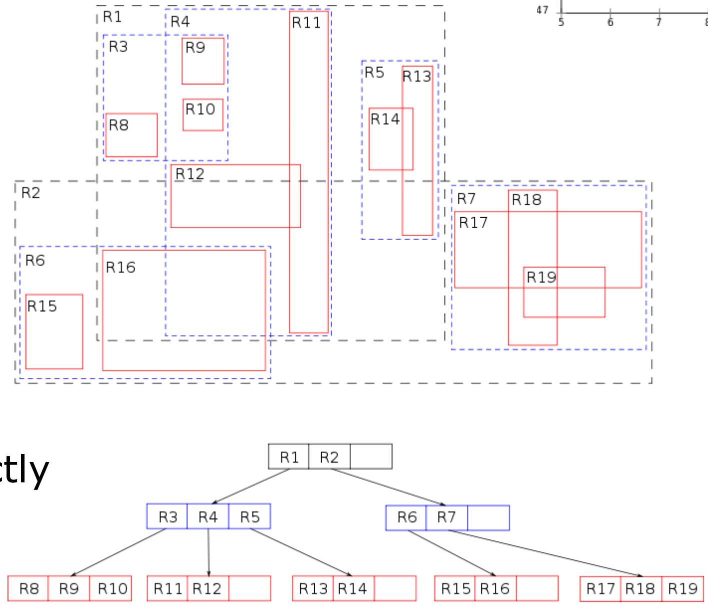


R-Tree

R-Tree

- A variation of a B-Tree that uses a hierarchy of rectangles as keys
- Also: balanced and block-sized nodes
- Indexed points …
  - are clustered into leaf nodes.
  - might occur in multiple clusters.
- Insertion:
  - into appropriate clusters
  - split cluster if too large
  - find smallest cluster extension via heuristic if no cluster fits directly

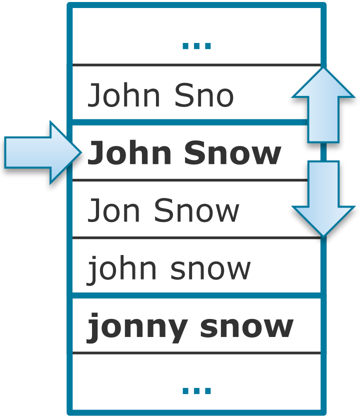




Alternative Index Types

Fuzzy Index

- Index on terms/keys that allows for value misspellings, synonyms, variations, …
- Idea: sparse, sorted index (e.g. SSTable or B-Tree) with similarity look up
- Example: An index on attribute `firstname` where names might be misspelled.
  1. Look up most similar key.
  2. Scan the (sorted!) neighborhood of that key's value for similar values.



Check yourself

- Given these two SSTable segments from 16/11/2018 and 17/11/2018, calculate their compacted merge.

16/11/2018

ambition	62
area	71
argument	59
assumption	87
atmosphere	40
attitude	53

17/11/2018

accident	63
ambition	14
ambition	27
anxiety	78
area	56
argument	85
argument	79
assistance	50

- Specify the order in which the elements are accessed.

Chapter 3. Storage and Retrieval