Storage and Retrieval: Memory, Search and Indexing

Reza Rawassizadeh

Outline

- Java Memory Structure
- Algorithm Complexity
- Search Improvement Methods
- Read and Write Optimization (SSTables, LSM-Tree and Bloom Filter)
- Lucene

Outline

- **Java Memory Structure**
- Algorithm Complexity
- Search Improvement Methods
- Read and Write Optimization (SSTables, LSM-Tree and Bloom Filter)
- Lucene

Java Memory Management

- Java has automatic memory management. It automatically performs **garbage collection**.

- By garbage we mean class objects which are instantiated, but not in use anymore. They occupy the memory. They are called garbage and the compiler should remove them.

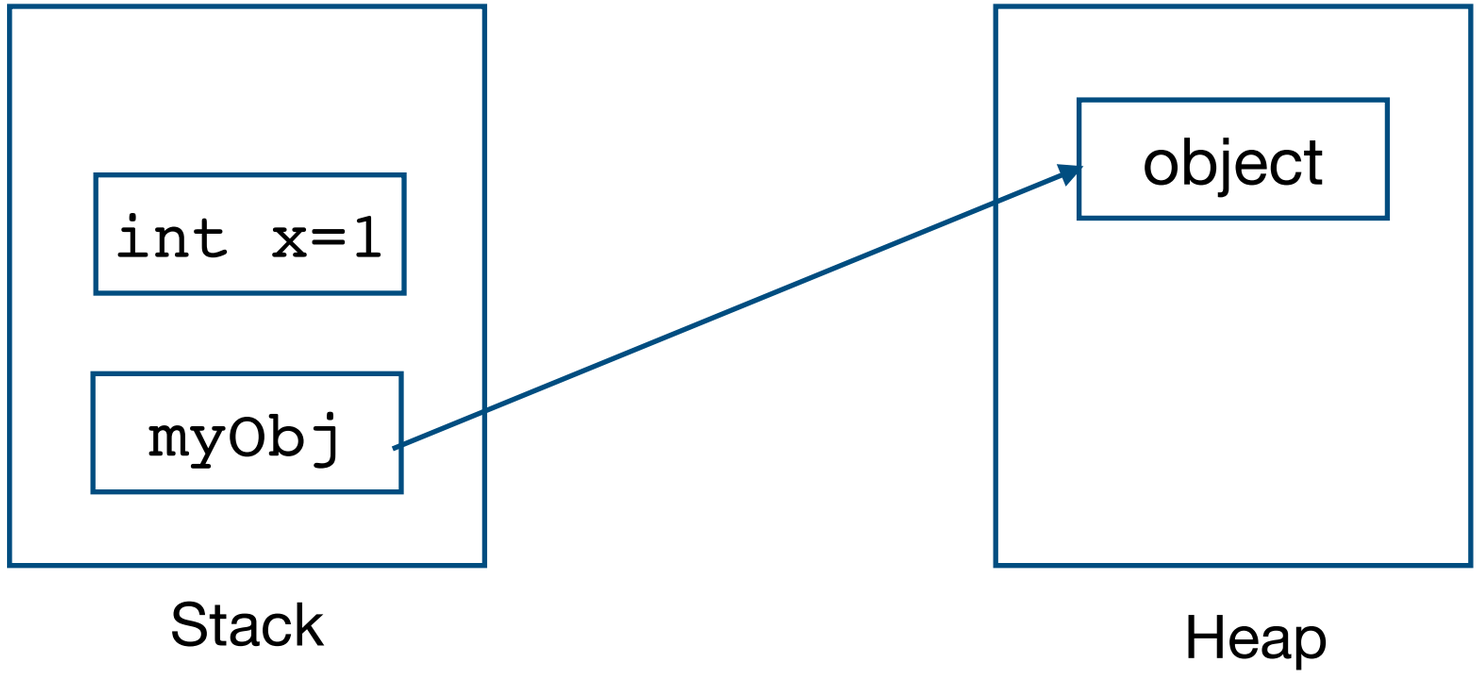
- There is no guarantee that automatic garbage collection of java does not cause our code to raise `OutofMemoryError`.

Memory Structure in Java

- Java divides memory into two parts, **stack** and **heap**.
- **Stack:** stores **Java primitive types (`int`, `char`, ...)** and **holds a reference to heap objects**.
- Java primitive types do not have reference to an object, their value directly stayed in stack memory.
- **Heap:** It is a part of the memory that **actual object will be stored**. In other words, the objects that their reference is located inside stack, hold their value inside the heap.

Memory Structure in Java

```
TestClass myObj = new TestClass()
```



Stack vs Heap

Stack

- The **size of the stack is flexible** and can changes as methods and functions create and delete local variables.
- **Memory allocation** (assignment and freeing) is **automatic**, without a need for code.
- Stack has **size limits**, which can vary according to the underlying operating system that hosts JVM.
- Variables inside the stack **remain there as long as the function that created them is running**.
- If stack space is full, Java throws **java.lang.StackOverFlowError**

Heap

- It is **not managed automatically** and we need to **free allocated memory by code**, when memory blocks are no longer needed.
- The heap is prone to **memory leak** problems, where memory is allocated to unused objects and will not be available to processes other than that.
- There is **no size limit** in the heap.
- Compared to stack, objects in the heap are much slower to access. It is also slower to write to the memory on the heap.
- If heap space is full, Java throws **java.lang.OutOfMemoryError**

Java Memory Structure

- We have only one heap memory for each JVM process. Therefore, heap is a shared part of memory regardless of how many threads are running inside the Java.

- Usually we do not configure maximum stack and heap size. However, Java enables us to define them as follows:

- `-Xms<size>` set initial Java heap size
- `-Xmx<size>` set maximum Java heap size
- `-Xss<size>` set java thread stack size
- `java -Xmx6g myApplication`

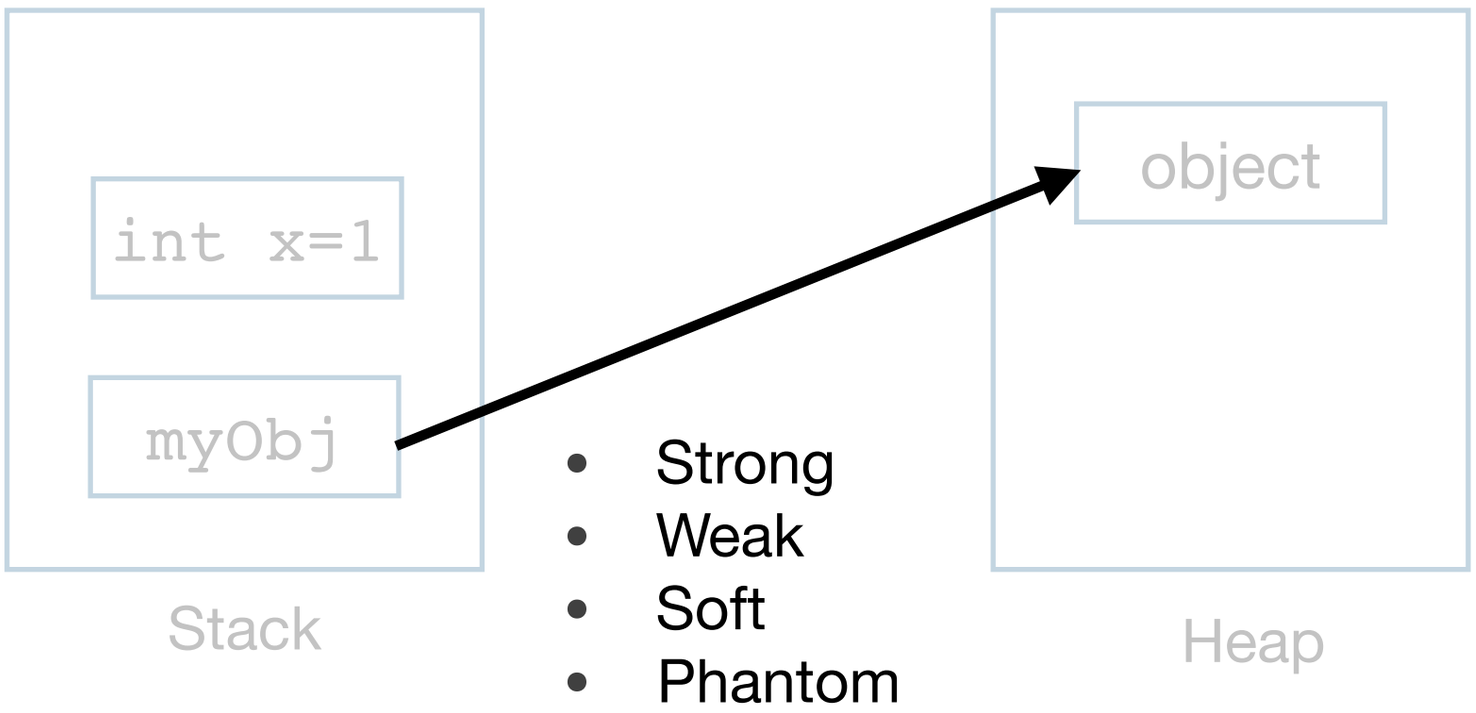
Memory Size Manipulation in Eclipse IDE

- Run —> Run Configurations —> Arguments —> VM arguments
- Enter `-Xmx6g and -Xmx256m`

`Test the following code:`

```
package edu.bu.met622.memory;

import java.util.Vector;

public class MemoryTest {

    public static void main(String[] args){
        Vector v = new Vector();
        while (true)
        {
          byte b[] = new byte[1048576];
          v.add(b);
          Runtime rt = Runtime.getRuntime();
          System.out.println( "free memory: " + rt.freeMemory() );
        }
      }
}
```

Reference Types in Java



source: https://www.onlinetutorialspoint.com/java/reference-types-in-java-strong-soft-weak-phantom.html

Memory References

- **Strong:** When there is a direct reference to an object and it is not eligible for garbage collection.

```
Test obj = new Test()
```

- **Weak:** When an object has neither strong nor soft reference, it is a weak reference. Weak references should be explicitly specified while referencing them.

```
// Strong Reference
Test a = new Test();
a.x();
// Creating Weak Reference to Test object to which ‘a’is also pointing.
WeakReference<Test> weakref = new WeakReference<Test>(a);
```

- **Soft:** When an object is free for garbage collection, but it is not yet garbage collected, until JVM is running low on memory and looking for objects to remove.

```
// Marked for gc, but JVM might not do it immediately.
 g = null;
```

- **Phantom:** Objects are eligible for GC, but JVM puts them in a queue called “**ReferenceQueue**”. We don’t know a phantom referenced object is dead or alive.

Outline

- Java Memory Structure
- **Algorithm Complexity**
- Search Improvement Methods
- Read and Write Optimization (SSTables, LSM-Tree and Bloom Filter)
- Lucene

Big O notation

Big O notation specifies the order of magnitude of performance rather than the exact performance. It also assumes the worst-case response time. If we write an algorithm that could take a while or be instantaneous, big O uses the longer one. It uses an *n* to reflect the number of operations or size we have.

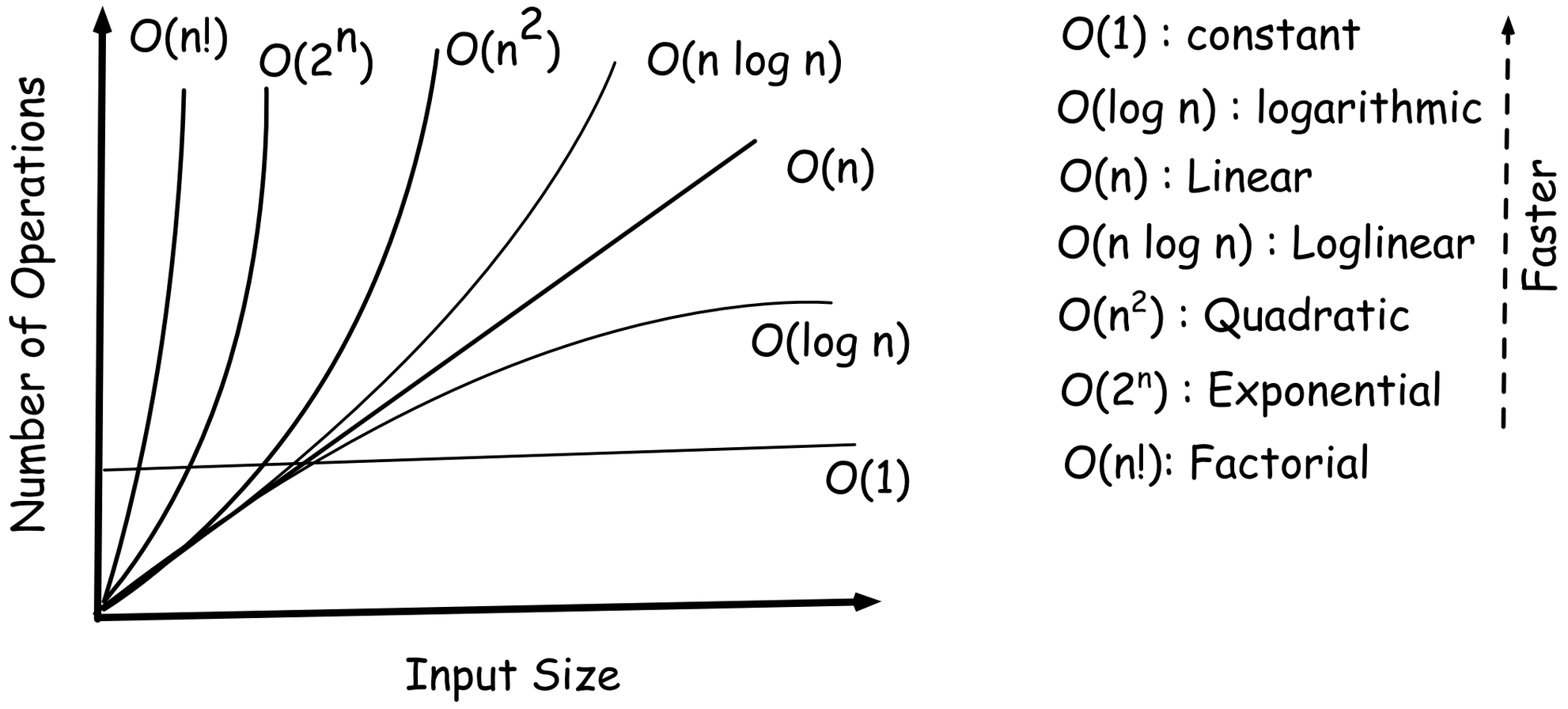
***O(1)*—constant time:** It doesn't matter how large the data is, the answer will always take the same time to return.

***O(n)*—linear time:** The performance will grow linearly with respect to the size of the collection. Looping through a list and returning the number of elements matching "Panda" will take linear time.

***O(n log n)*—logarithmic time:** A logarithm is a mathematical function that grows much more slowly than the data size. Binary search runs in logarithmic time because it doesn't look at the majority of the elements for large collections.

***O(n^2)*—n squared time:** Code that has nested loops where each loop goes through the data takes *n* squared time. An example would be putting every pair of pandas together to see if they'll share an exhibit.

Big O notation



Outline

- Java Memory Structure
- Algorithm Complexity
- **Search Improvement Methods**
- Read and Write Optimization (SSTables, LSM-Tree and Bloom Filter)
- Lucene

Search Improvement Methods

- Hash Tables
- Tree Structures
- Bit Manipulations & Compression
- Sliding Window
- MinHashing
- Bloom Filter

Search Improvement Methods

- Suppose that we have a dataset composed of 1000 transactions and each transaction has 100 member (itemset).

- For the first transaction the algorithm should compare followings items together: $\{i_1,i_2\}$, $\{i_1,i_3\}$...$\{i_1,i_{100}\}$, $\{i_2,i_3\}$, $\{i_2,i_4\}$,... $\{i_2,i_{100}\}$, ... Therefore, for single transaction we need comparisons. Assuming we have 1000 transactions, then this number of comparison will be extremely large. This method called **brute force**.

transaction	items
1	$i_1, i_2, \ldots, i_{100}$
2	.
3	.
⋮	
1000	.

Search Improvement Methods

- **Hash Tables**
- Tree Structures
- Bit Manipulations & Compression
- Sliding Window
- Minhashing

Hash Table

A hash function is a function that maps the original data (value) to a smaller size data (key). Usually a hash function will be applied on a set of data (2D data) and as a result we get a 2D table, which is hash table.

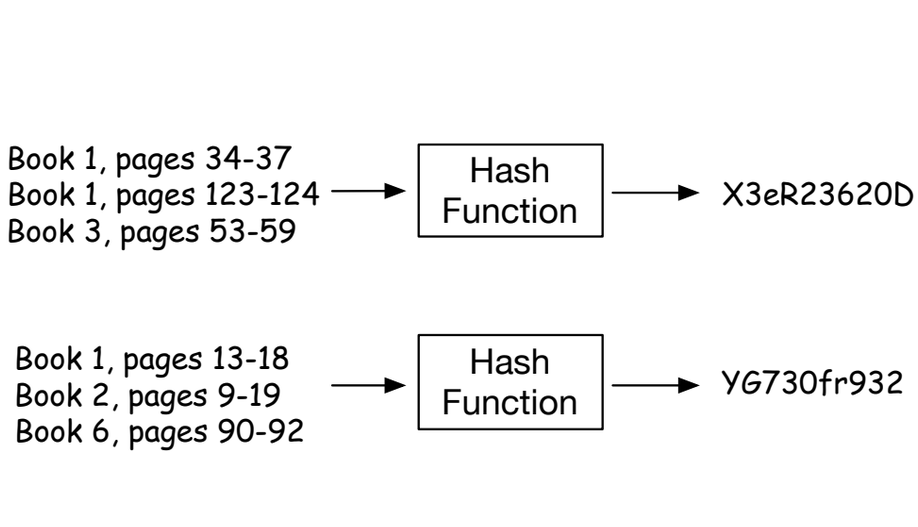
- A gym coaching system.

- Reading books to coach us and answering our questions.

- To search books in a reasonable time it needs **index**.

Keyword	Text Location
weight lose	Book 1, pages 34-37 Book 1, pages 123-124 Book 3, pages 53-59
cardio	Book 1, pages 13-18 Book 2, pages 9-19 Book 6, pages 90-92
bench press	Book 3, pages 101-102
. . .	. . .

Hash Function



Hash value (Keywords)	Hash Key (new value for Text Location)
weight lose	X3eR23620D
cardio	YG730fr932
bench press	F46yu19Wst
. . .	. . .

Hash function usually converting the large strings into smaller strings with equal sizes.

You can say in this example we use hash function to compress the data. We could not say your argument is wrong, but it is more than compression, it also enables the system to rapidly access the data.

Hash table is a data structure that associates a key to a value to enable the algorithm very quickly lookup for the original data [McDowell '16].

Reading a record from hash table, adding a new record into hash table, or updating an existing record in the table all have *O(1)* complexity, in worst cases, when our hash function is not good, it has *O(n)* complexity.

BitCask Hash Table Example

- Bitcask is an implementation that supports hash index. It offers high-performance read and write.

- All information will be stored inside memory, but if there is a need to have a read from the disk, with one single disk seek it can read the required data from disk into memory.

- Bitcask is useful when we have lots of updates in our key-value pairs.

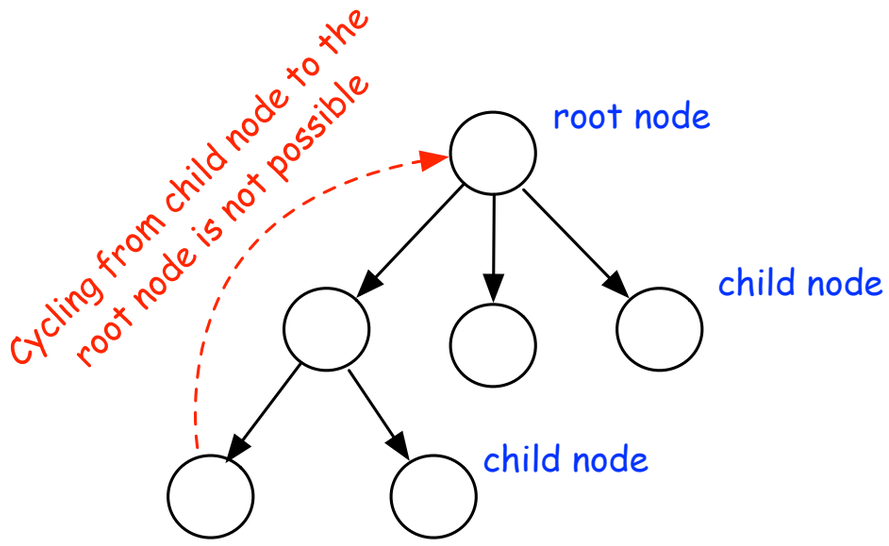
- You can look more about Bitcask here: https://docs.riak.com/riak/kv/2.1.4/setup/planning/backend/bitcask/

Search Improvement Methods

- Hash Tables
- **Tree Structures**
- Bit Manipulations & Compression
- Sliding Window
- Minhashing

Tree Structure

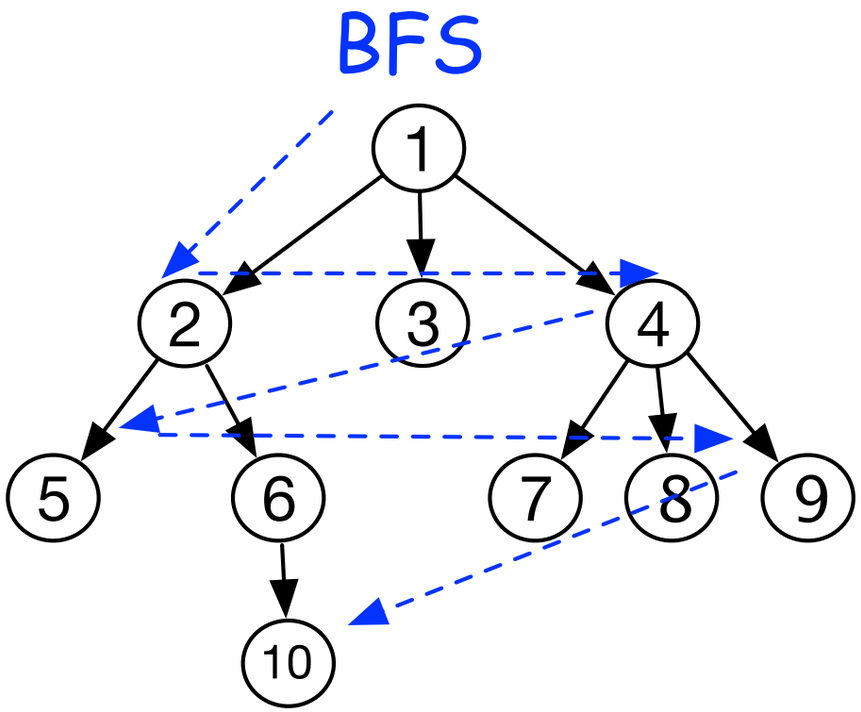
Tree is a sub type a graph data structure, which has a root and child nodes (but without cycle)



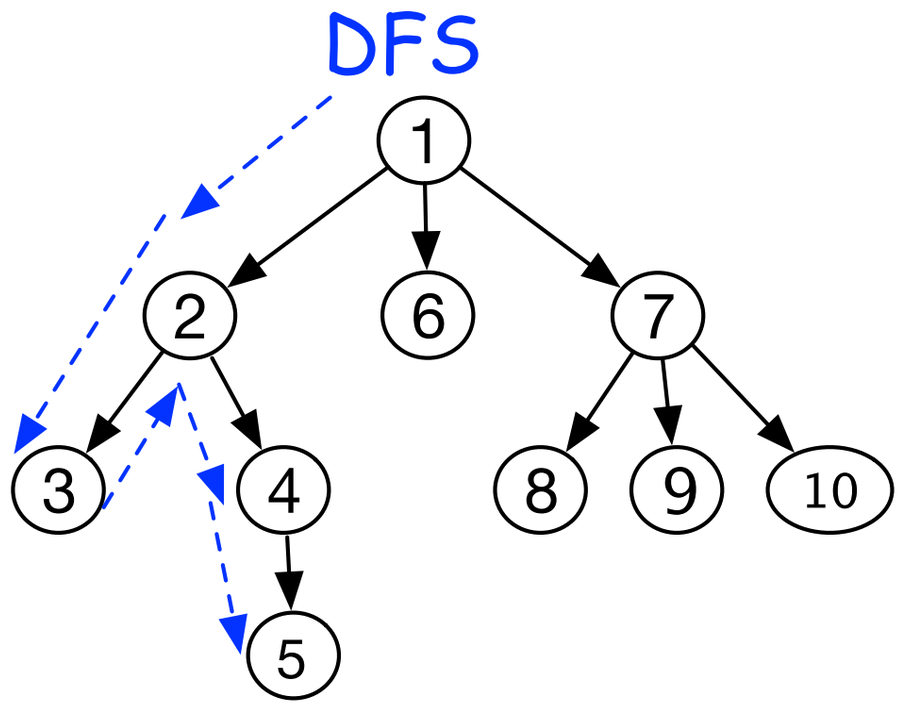
Tree structure and its elements. Note that, unlike graph, cycling is not possible in tree.

Scanning the Tree

Breadth first search



Depth first search

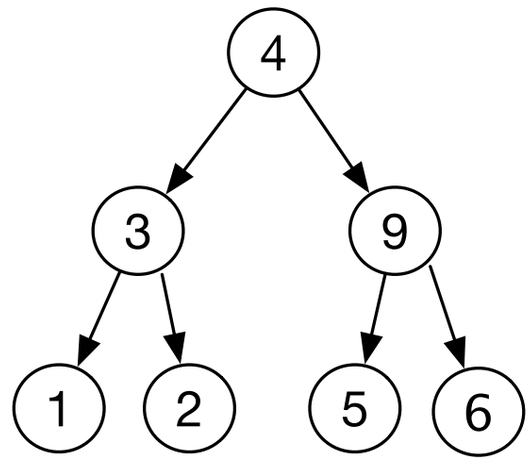


Binary Tree

*When each root node in the tree has only two child nodes, it is called **binary tree**.*

Binary search tree (BST) is a type of binary tree in which all nodes on its left size should be smaller that the nodes on the right side, i.e., all left nodes < all right nodes. Besides, all nodes of BST are **stored as key/value** pairs, and keys are presented as values on nodes.

BSTs are useful when the data objects (or keys) are having a **sort of order** inside the tree, because, when the algorithm intends to compare a new data object with the content of the tree, it checks **whether the new data object is smaller or larger than the root**. If it is **smaller** then, it goes and search the **left side of the tree**, otherwise, if it is greater it goes and search the right side of the tree.

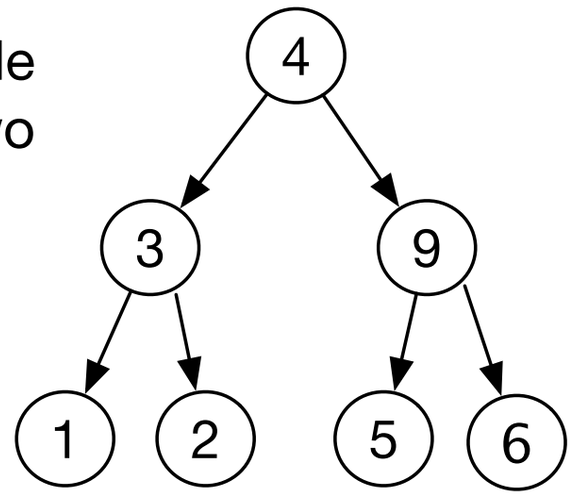


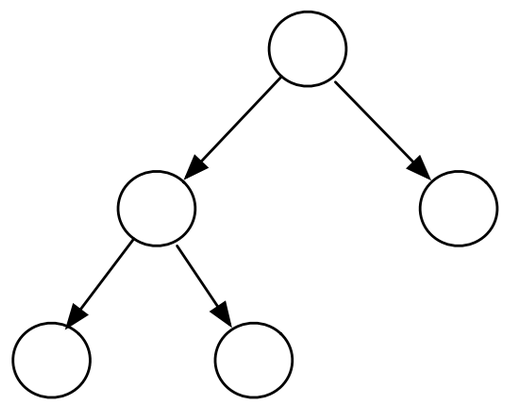
An example of balanced binary tree, which is also balanced.

Binary Tree

Searching an information in binary tree usually takes ***O (log n)***. **Inserting** a new key (data object) inside a binary tree is not extremely efficient, i.e. ***O(2n)***, and **when the table is not empty it is *O(n^2),*** because it the algorithm should navigate through a correct branch to fit the new key into its correct position or perhaps update a node and its children structure. Therefore, it is good to consider using binary table **when the structure of data is static** and not changing.

A **full binary tree**, means that each node of the table is either zero or have two children (both figures are full binary tree)





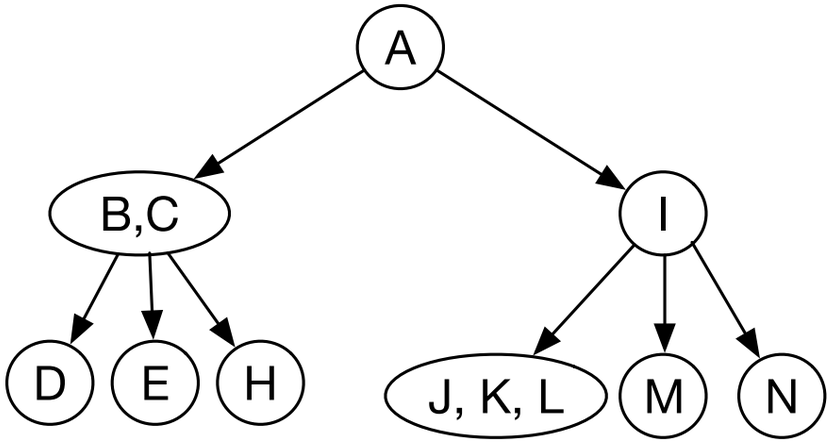
Unbalanced binary tree.

The advantage of **balanced binary tree** over BST is that their search cost and insert cost is guaranteed to be logarithmic ***O(n log n)***

2-3 Search Tree

It is not trivial to construct a balance tree. Therefore, in some implementation nodes could hold more than one keys. These types of trees are called **2-3 search trees.** When there is *n* keys in a node, it always has *n+1* children.

They are not binary and balanced anymore but still keep tree nodes on balanced depth. Besides, keys are ordered.



2-3 search tree, which some node hosts more than one key.

The good thing about 2-3 search tree is that search and insert cost both guaranteed to be *O(log n)*

B-Tree

- B-Tree (Balancing Factor Tree) is a type of a 2-3 search tree, in which a node can have more than two child.

- All of B-Tree leaves must be at the same depth.

- B-Tree has a *B* parameter which is **branching factor** and following two rules are always applicable in B-trees:

  - (i) *B* <= children per node < *2B*

  - (ii) *B-1* <= keys per node < *2B -1*

Red-Black Tree

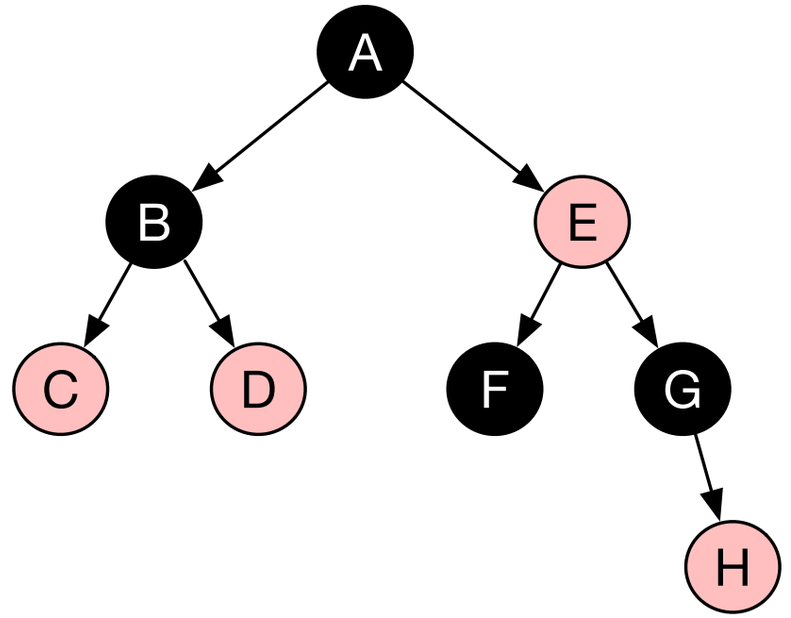
Red-Black tree is an approximately balanced tree, which has four properties:

(i) tree nodes have a color, either black or red (a bit can be used to set the color of node),

(ii) always the **root** of the tree is **black**,

(iii) it is not possible to have a **parent and child both in red** (in other words **every red node must have two or null black child**),

(iv) every **path form root to a leaf nodes**, must have **equal number of black nodes**



The insert and search in red-black tree is also *O(log n)*, but it requires less memory to balance the tree. Red-black tree is useful, where we need to insert and remove data into the tree frequently.

Trie, Prefix Tree, Radius Tree

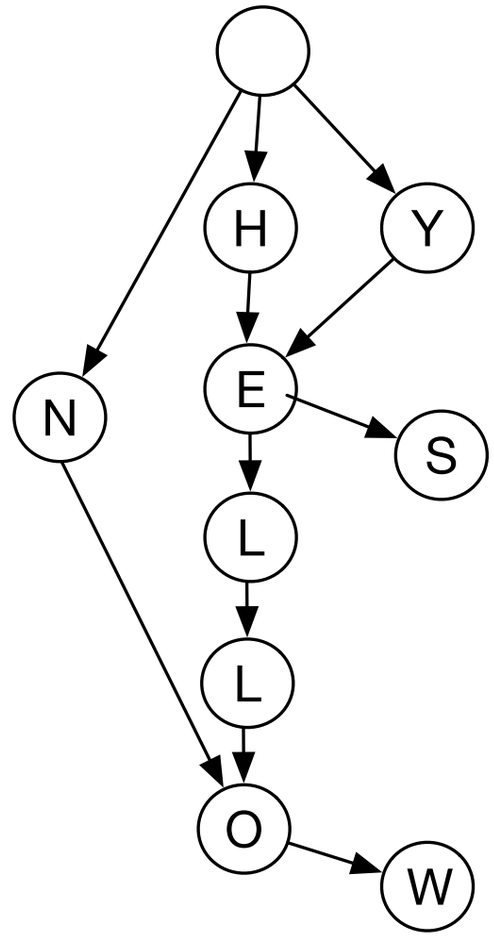
- Trie nodes are assumed to be **ordered from the root** and the root node of the trie is always **empty**.

- Nodes in trie unlike other trees do not hold a key, they hold a **partial key**.

- Trie can be used to store words, the path from the root to the last child presents a word and each node stores a character.

- When we add more words (or information) into a trie the more its size grows, but it requires less time to find the right place for the newly inserted data object, because, as the tree is getting bigger, the tree holds more words.

- A more advanced form of tire is called **patricia tree**, where a node can have more than one keys (more than one character)



Trie can find the word in a dictionary in *O(n)* time, assuming *n* is the length of words (characters of words). Besides, it can insert word in *O(n)* time.

Search Improvement Methods

- Hash Tables
- Tree Structures
- **Bit Manipulations & Compression**
- Sliding Window
- Minhashing

Data Compression and Bit Manipulation

The **bit** is the smallest unit of information in computer science and it can be either 0 or 1. Eight bit constitute a larger data called byte. 1024 bytes called kilo byte, 1024 kilo byte called mega byte, 1024 mega byte called giga byte and 1024 giga byte called tera byte,…

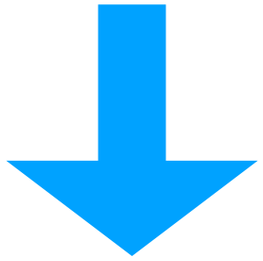
Hi = 0100100001101001, H= 01001000 and i= 01101001

- The process of **data compression** is transforming the data from one format into another format that is *smaller => occupies less space => reduces the latency (because search execution time will be reduced)*.

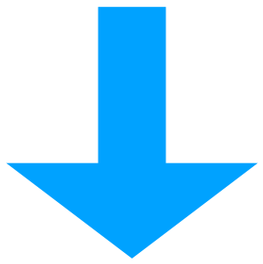
- Usually, this process is done on the **bit level,** and the data is converted into a **binary format**, some bit manipulation operations will be done on the data, its size is reduced, and then it gets smaller**.** Either the information will lose some of its data (**lossy compression**) or not lose any of its data (**lossless compression**).

Data Compression Example

do not ask what XYZ can do for you, ask what can you do for XYZ?



An algorithm could create a dictionary and assign number to each word as follows:
do=1, not=2, ask=3, what=4, XYZ=5, can=6, for =7, you=8



{1 2 3 4 5 1 6 7 8, 3 4 6 8 1 7 5?}

Lempel-Ziv-Welch compression [Ziv ’78].

Byte Pair Encoding

One of the challenges is tokenizing words in languages that do not have space such as Chinese, Japanese and Korean. To tokenize those texts we use could use **subword tokenization** algorithms.

A compression algorithm for this approach is Byte Pair Encoding. It is a simple recursive algorithm that scans the string of characters and substitutes the repeated pair of characters (at least two characters appeared together more than once) with one character. Then performs the scan again and substitute the repeated pair of characters again with one character. This process continues all character strings are repeated only once in the text string.

Byte Pair Encoding

To better understand this algorithm, consider an example that we have a short text written as:
veeeerrrrry gggood arry.
The first scan identifies that the pair 'rr' occurs most often, and the algorithm replaces it with some character such as '!' , we use a sign for the sake of understanding. The result will be as follows:
veeeer!!y gggood a!y
Again it scans the new string to identify the most repeated pair, which is 'ee' and '!y'. It substitutes them with one character, let's say 'ee' with '@' and '!y' with '#'. Thus, as a result, we will have the following string:
v@@r!# gggood a#.
This process continues until there is no duplicate left and we have a string as follows.
v*r!# %g^d a#

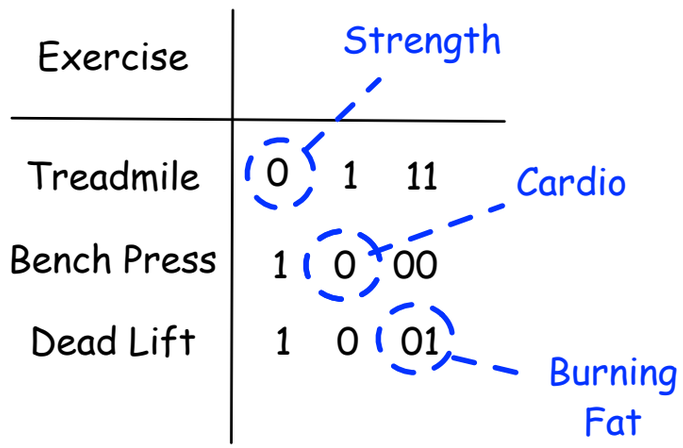
Similar to all other compression algorithms there is a catalog/dictionary that includes all variable mappings as well. In this example, the catalog is as follows:
rr = !, ee = @, !y = #, @@ = *, gg=%, oo=^

Bitmap Index

- One useful concept with bitwise operation or compression is the use of **Bitmap index**.

- Bitmap indexes are useful for data with low cardinality. By cardinality here we mean the diversity of data is limited, for instance, gender could be either ‘male’ or ‘female’.

Exercise	Strength	Cardio	Burning Fat
Treadmile	no	yes	high
Bench Press	yes	no	Low
Dead Lift	yes	no	Medium



(left) Different gym exercise training and this impact, (right) converted exercise data to bitmap.

Huffman Encoding

- Huffman coding is a popular lossless compression technique.
- input: $\{a, b, c, d, a, d, c, b, c, e, e, c, b, b, a, c, e\}$
- by default, computers use ASCII codes (8 bits) to store each character, which means here we need $16 \times 8 = 128$ bits to store this string of characters.
- Instead of using ASCII codes, we can use three bits to present all four characters, something like following, with occupies $16 \times 3 = 48$
  $a = 000$, $b = 001$, $c = 010$, $d = 100$, $e = 101$.
- Therefore, it is smaller than the original string. Here, we have only four unique characters; in a real-world case, we have also lots of repetitive information, like words inside the text document, pixels inside a picture, etc. Huffman encoding can compress much better than this simple approach (not using ASCII format).

Huffman Encoding

- 1- frequencies of words/terms/characters,… are written in-front of them as follows: $\{a = 12, b = 15, c = 11, d = 24, e = 20\}$

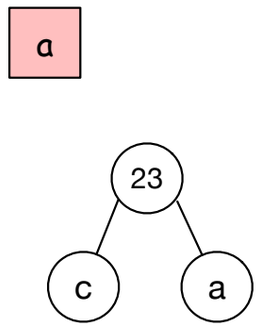
- Meaning, to store this text in ASCII code we will need 640 bit to store this text, because:
  $12 \times 8 + 15 \times 8 + 11 \times 8 + 24 \times 8 + 20 \times 8 = 640$, but if we store this text with our three bit compression we will need
  $12 \times 3 + 15 \times 3 + 11 \times 3 + 24 \times 3 + 20 \times 3 = 240$ bits. It is smaller than 640, but still we could make it also smaller (by using Huffman coding).

- Huffman encoding operates based on creating a binary tree, and assign values to each character based on their position inside the tree.

Huffman Encoding

- First the algorithm orders characters based on their frequency, which will be as follows: $\{c = 11, a = 12, b = 15, e = 20, d = 24\}$. Next, it takes the two data objects which have the lowest frequency and construct the two right most leaf nodes.

- The root node value for these two nodes are sum of the frequency of these two nodes, $11 + 12 = 23$. (Fig a)

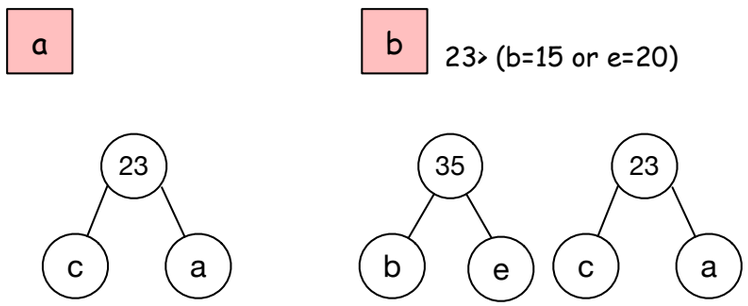


$$\{c = 11, a = 12, b = 15, e = 20, d = 24\}$$

Huffman Encoding

- Then we look for the next node which has the lowest frequency, i.e. $b = 15$, and adds it to the tree as another node, but since it is a binary tree it adds it into a new branch and the new result root node will be $15 + 23 = 38$ (Fig b.). On the other hand combining *b* and *e* results in smaller number so the algorithm choose *b* and *e* and makes a new branch.

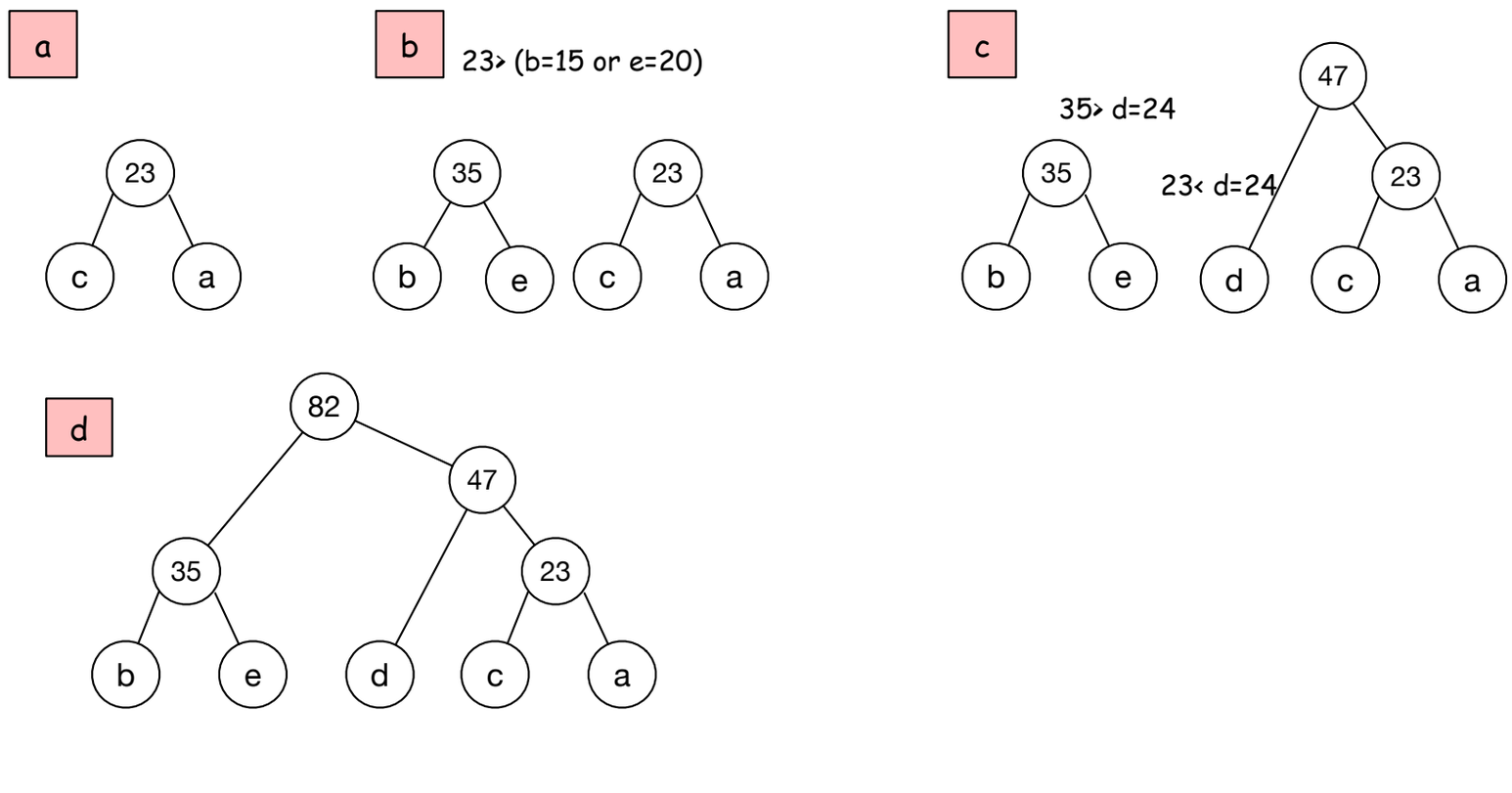
$$\{c = 11, a = 12, b = 15, e = 20, d = 24\}$$



Huffman Encoding

- Now, we should process d and 35 is larger than the 23. Therefore, the algorithm assigns it to the branch with lower frequency number and the new parent node will have value of 23+24= 47. There is no more data objects left to be added into the tree. Therefore, the algorithm connects both branches with the root value of 47+35 = 82 (Fig d).

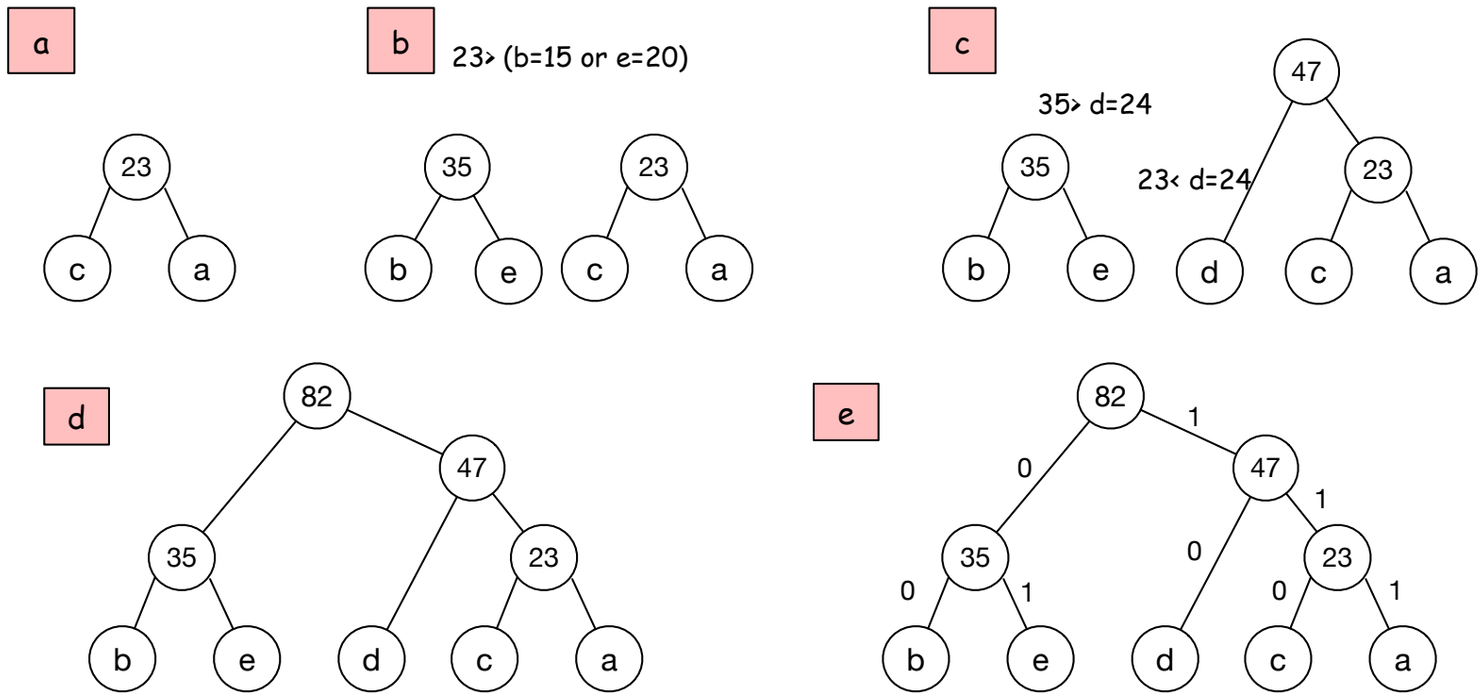
$$\{c = 11, a = 12, b = 15, e = 20, d = 24\}$$



Huffman Encoding

- When the tree is constructed the algorithm assigns 0 to all left branches and 1 to all right branches (Fig e). Now every character can be represented with a bit string started from root, $\{b = 00, e = 01, d = 10, c = 110, a = 111\}$

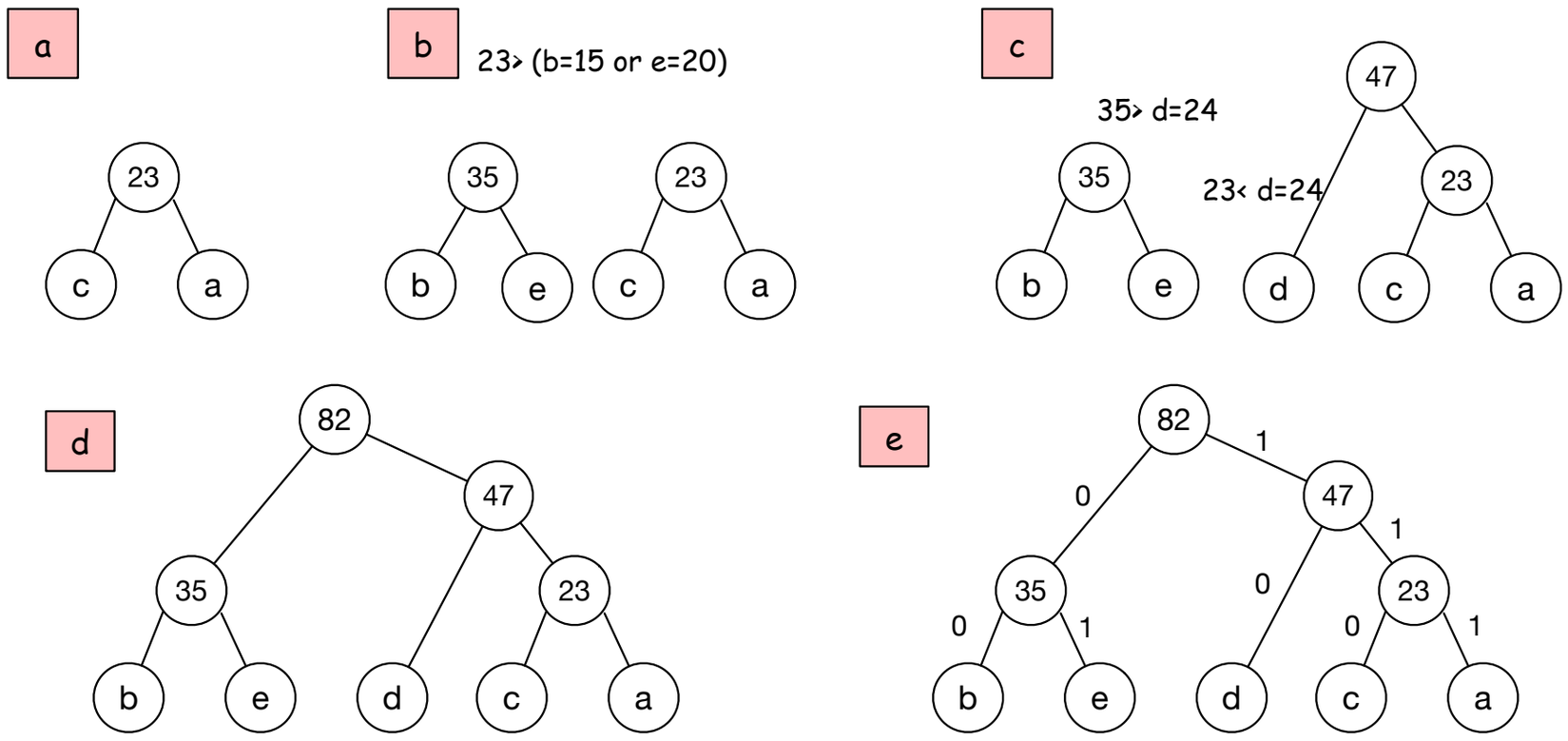
$$\{c = 11, a = 12, b = 15, e = 20, d = 24\}$$



Huffman Encoding

- You can realize that with this representation characters that are more frequent have smaller number of bits to present them and characters that are less frequent have larger number of bits.

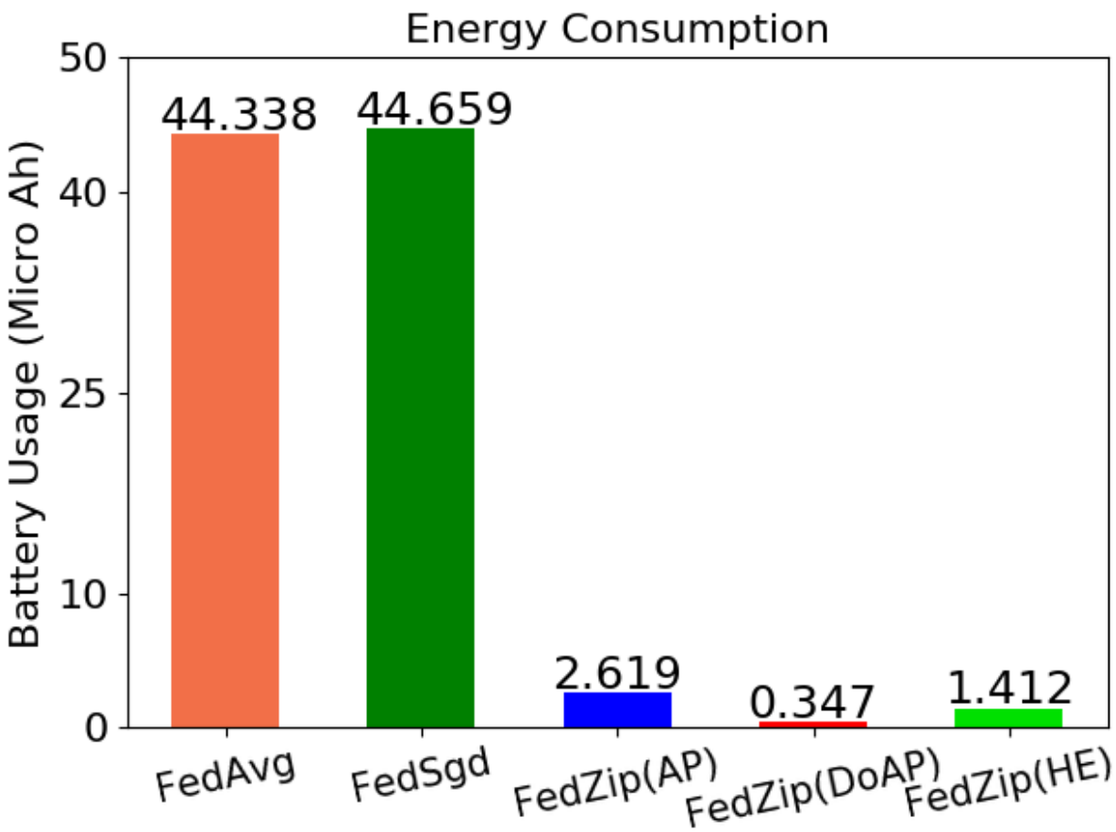
$$\{c = 11, a = 12, b = 15, e = 20, d = 24\}$$

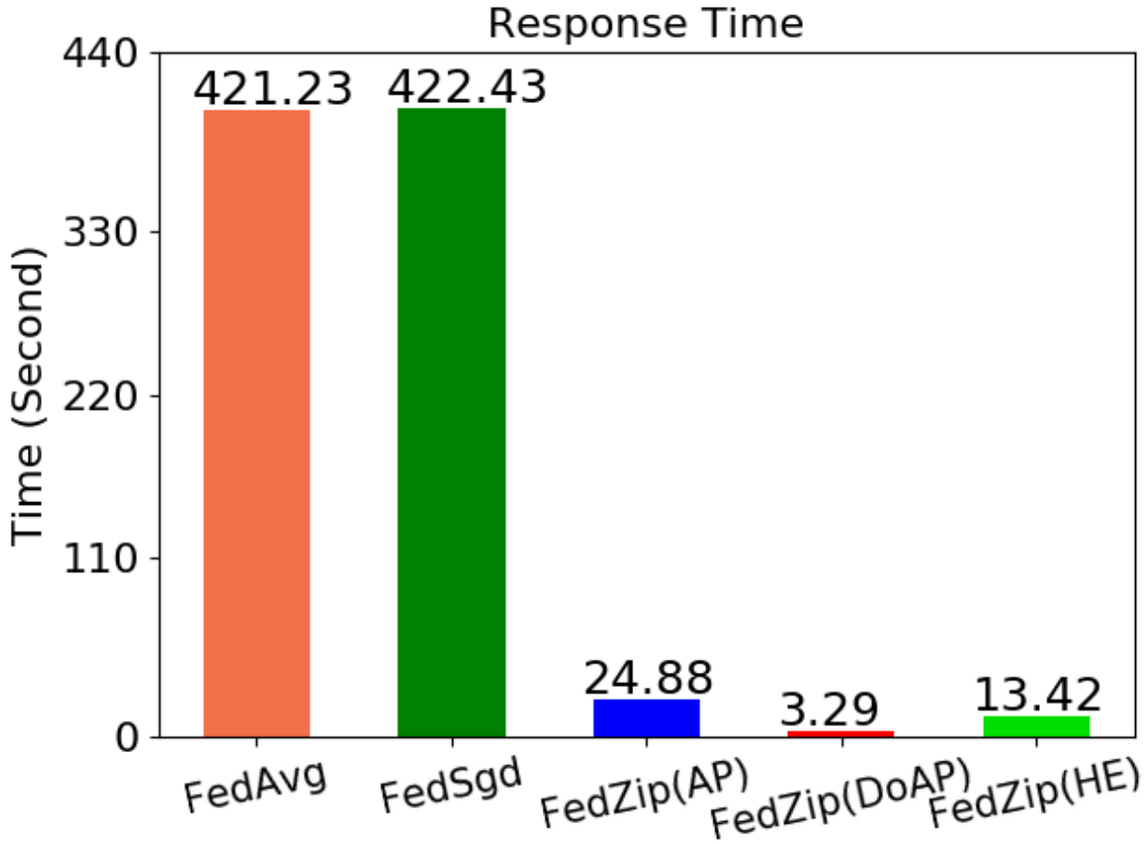


$$12 \times 3(111 = a) + 15 \times 2(00 = b) + 11 \times 3(110 = c) + 20 \times 2(01 = e) + 24 \times 2(10 = d) = 185$$

Since 185 is smaller than 240, it is more compressed. Besides, we need a small space to store the tree as well, which is too small and usually neglect able.

How Huffman Encoding can make a revolution?





Malekijoo, A., Fadaeieslam, M. J., Malekijou, H., Homayounfar, M., Alizadeh-Shabdiz, F., & Rawassizadeh, R. (2021). FEDZIP: A Compression Framework for Communication-Efficient Federated Learning. *arXiv preprint arXiv:2102.01593*.

Java Examples

- A simple implementation: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

- More detailed implementation: https://www.techiedelight.com/huffman-coding/

- In other programming languages: https://rosettacode.org/wiki/Huffman_coding

Compression Report

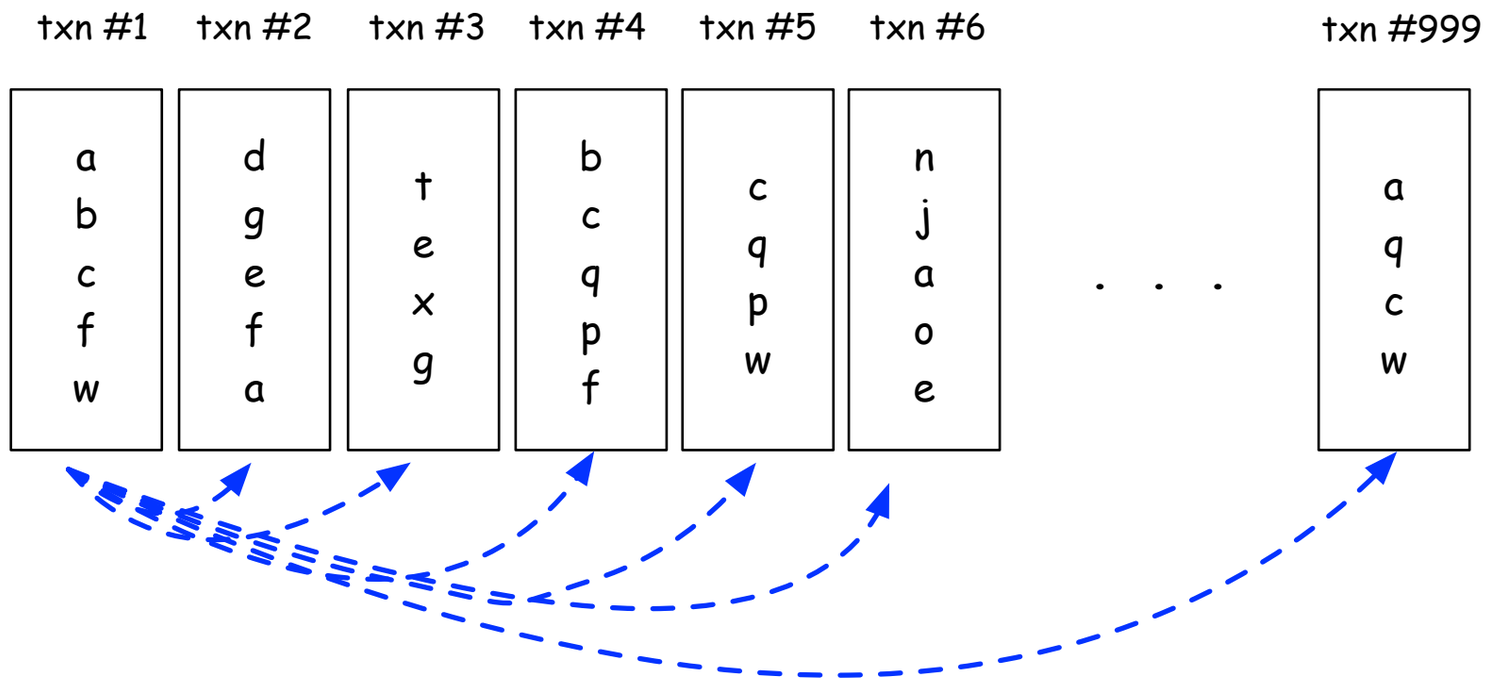
Algorithm	Compression Ratio (%)	Average Encoding Time (s)	Average Decoding Time (s)	Total Time of Encoding and Decoding (s)
LZW Encoding	**397.34**	1.3074	**0.0236**	1.3310
Arithmetic Encoding	187.55	0.0662	0.0714	0.1376
Shanon-Fano Encoding	180.57	0.4704	0.4918	0.9622
Run Length Encoding	24.24	**0.0548**	0.0538	0.1086
Huffman Encoding	187.17	0.0674	0.0338	**0.1012**

Table 4. The results of applying five compression algorithms on the synthetic dataset.

Search Improvement Methods

- Hash Tables
- Tree Structures
- Bit Manipulations & Compression
- **Sliding Window**
- Minhashing

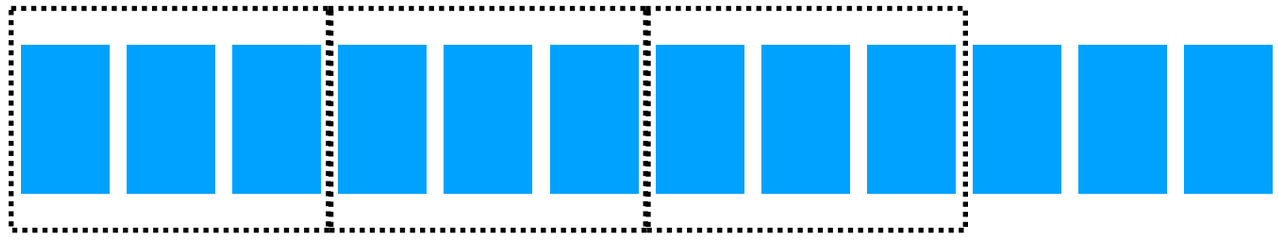
Sliding Window



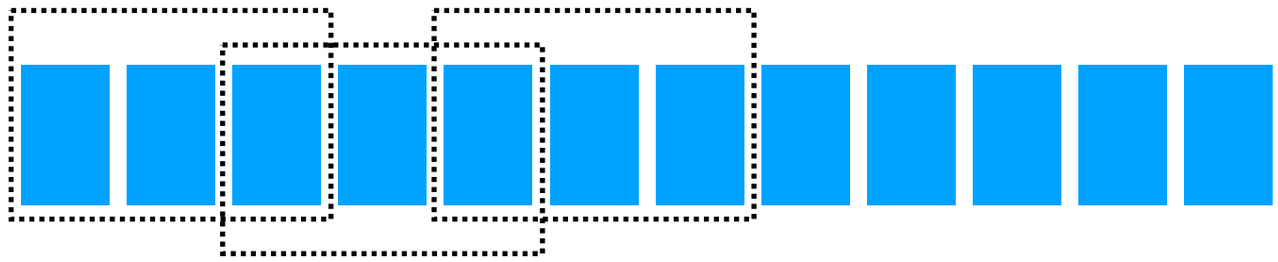
Comparing events of txn #1 with all other transactions to identifies its frequent itemsets. This is brute force comparison and it is not feasible.

Sliding Windows

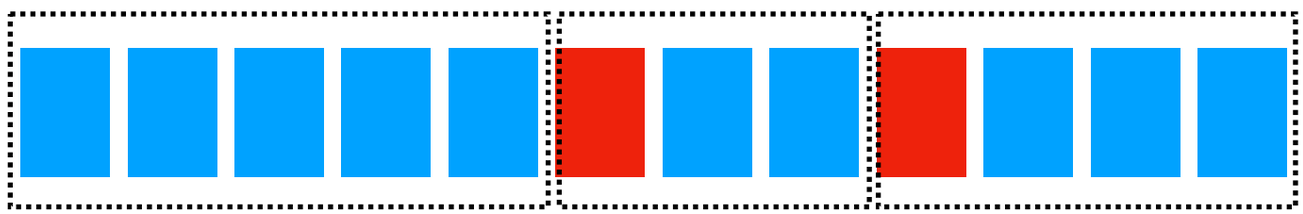
- Disjoint Sliding Windows (windows are not overlapping)



- Overlapping Sliding Windows(windows are overlapping)

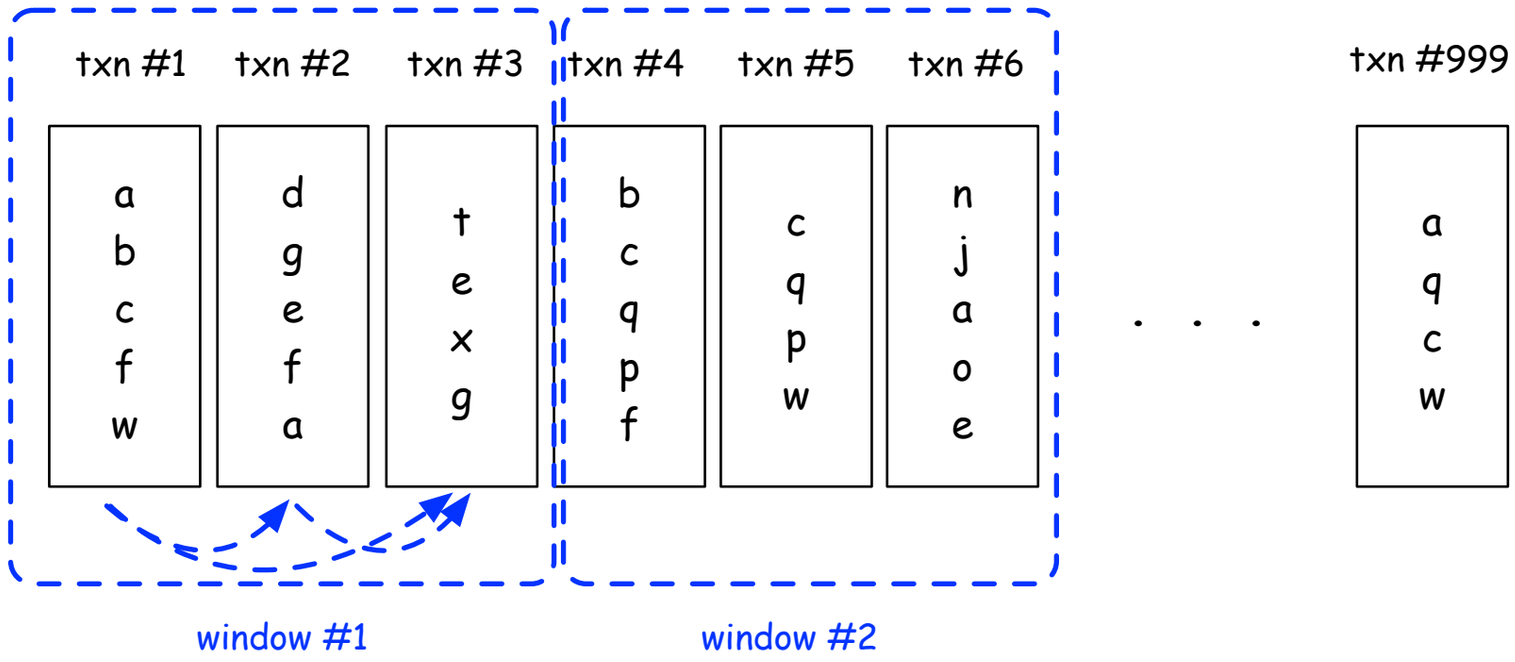


- LandMark Sliding Windows (size of the window depends on a encountering a landmark event)

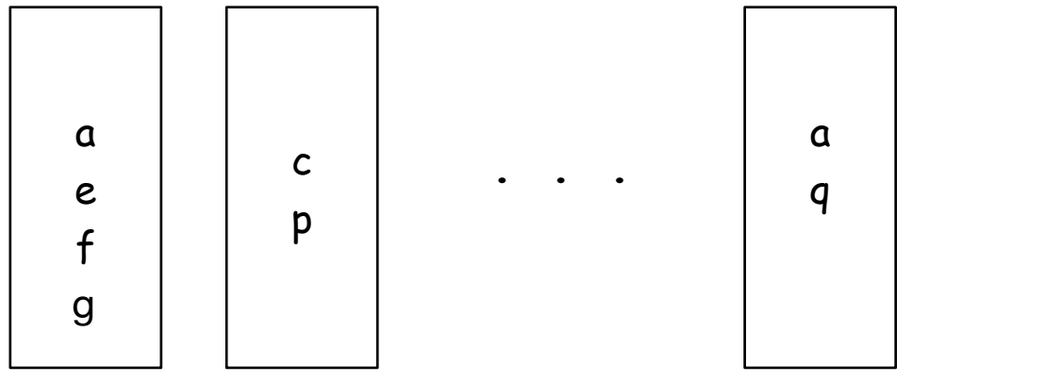


Sliding Window Example

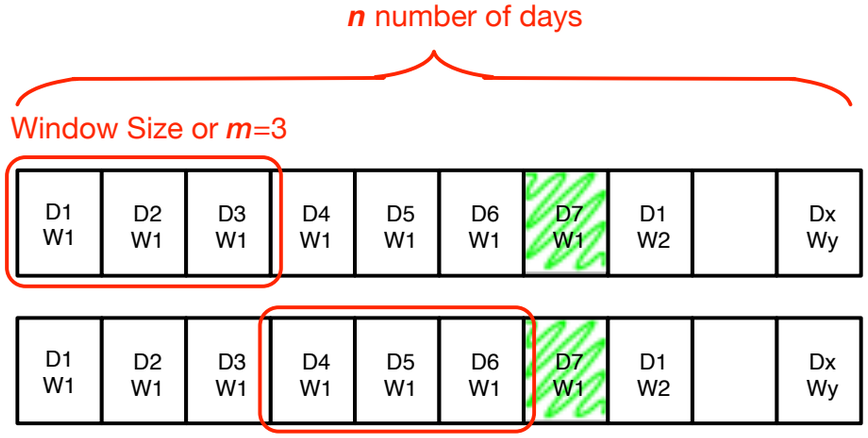
Phase 1:

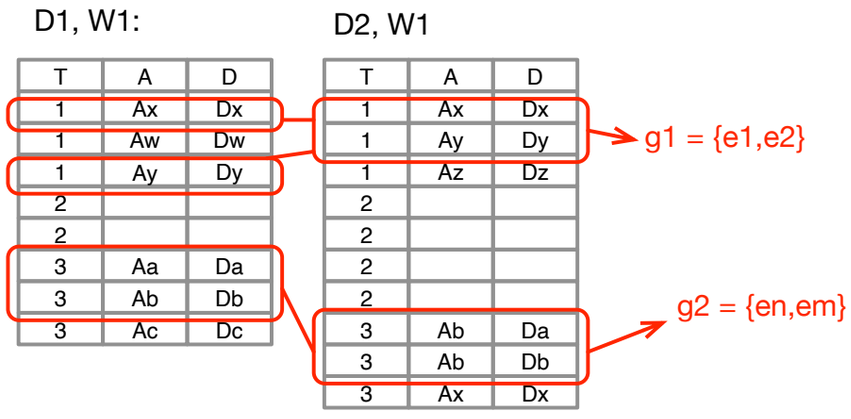


Phase 2:



Sliding Window Impact





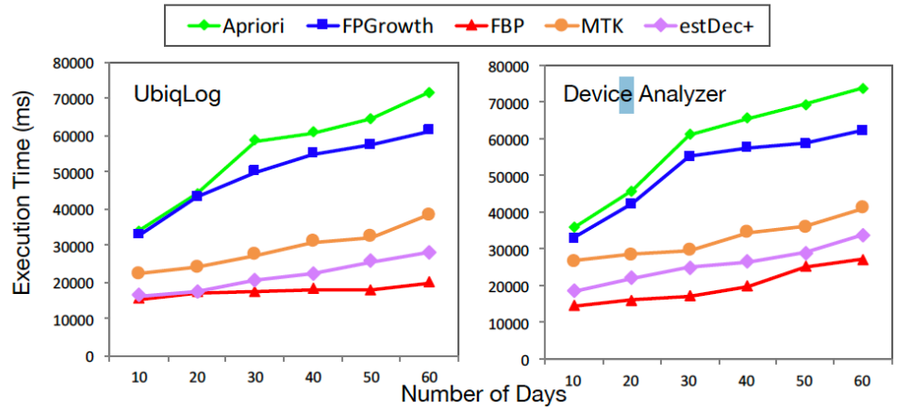


Fig. 6: Execution time comparision between FBP and other algorithms on the smartphone.

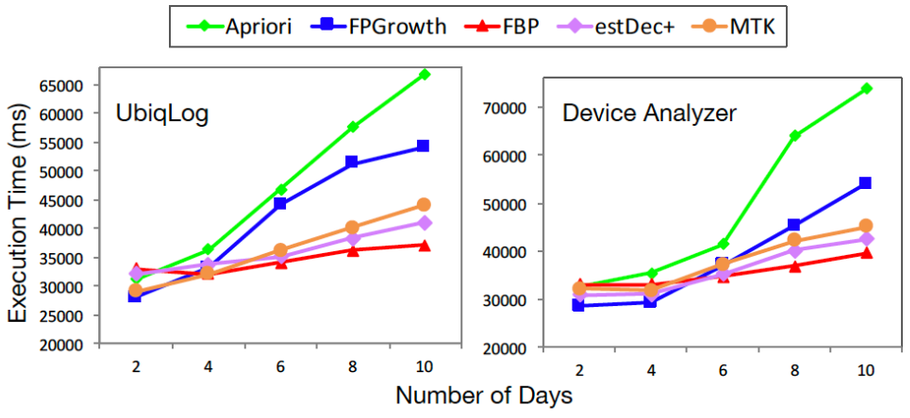


Fig. 7: Execution time comparision between FBP and other algorithms on the smartwatch.

Rawassizadeh, R., Momeni, E., Dobbins, C., Gharibshah, J. Pazzani, M. (2016) Scalable Mining of Daily Behavioral Patterns from Multivariate Temporal Data. In *IEEE Transactions on Knowledge and Data Engineering*. 28(11), 3098-3112. ACM Computing Review Notable Article of 2016.

Search Improvement Methods

- Hash Tables
- Tree Structures
- Bit Manipulations & Compression
- Sliding Window
- **Minhashing**

MinHashing

- MinHash, which is widely in use to efficiently estimate similar sets of information objects. The information object could be image, time series (audio signal, heart rate) or anything. We can convert *any information object into a set or vector* and use Minhashing method to identify their similarities.

- Jaccard similarity is used to quantify the similarity between two sets of information and it is written as:

$$J(S_1, S_2) = \frac{S_1 \cap S_2}{S_1 \cup S_2}$$

- Minhash is using Jaccard. It maps each set of information into **signature vectors** which are vectors of fixed length. This enables the Jaccard to identify how similar are two sets *without enumerating* each of their elements.

MinHashing

Gym Member	Exercises
Member 1	bench press, leg press, squat
Member 2	bicep curls, over head press
Member 3	bicep curls, tricep curls, over head press
Member 4	bench press, over head press, squat

	Member 1	Member 2	Member 3	Member 4
bench press	1	0	0	1
leg press	1	0	0	0
squat	1	0	0	1
bicep curls	0	1	1	0
tricep curls	0	0	1	0
over head press	0	1	1	1

- **Step 0:** The step before the first step, is converting our dataset into a boolean (0 or 1) matrix, and any matrix that includes lots of zeros, is called a sparse matrix.

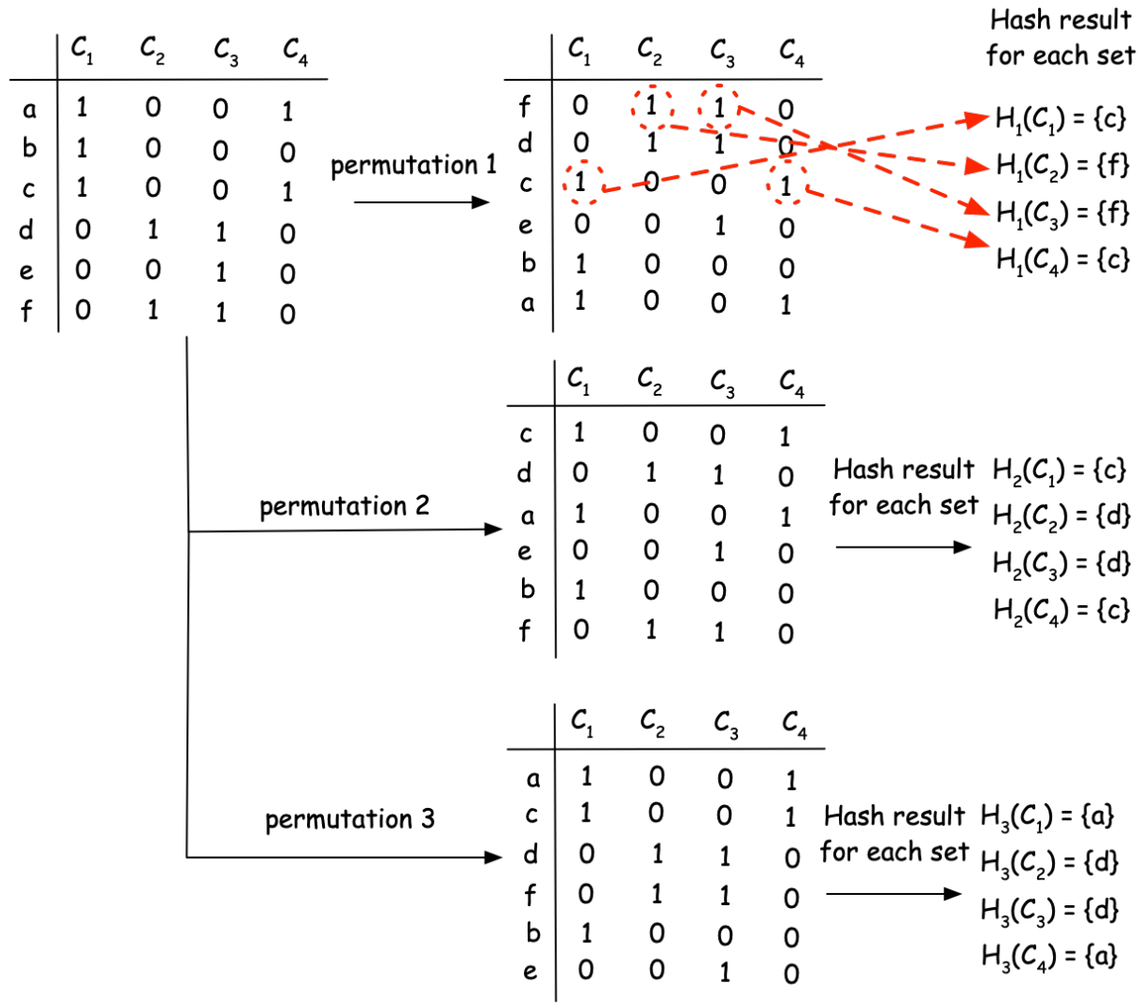
MinHashing

Step 1: For the sake of simplicity, we rename columns as *C*s and rows as alphabets (*a,b,c,…*) in the matrix presented in the Table. Therefore, the result will look like the left side of Figure.

Next, the MinHash algorithm randomly permutes rows, and as a result, we will have a matrix on the right side of Figure.

In this example, we permuted the table three times (tree has been chosen arbitrarily), but based on our signature vector size and dataset size, we can create more permutations.

The MinHash of each *C* (gym member) is the number of the row (fitness activity) with the first non-zero (i.e. 1) in the permuted order.



MinHashing

Step 2: Based on the hash values for three permutations in the Figure, we can write a signature $S(\ )$, for each set member as a set of its hash values, such as following:

$S(C_1) = \{H_1(C_1), H_2(C_1), H_3(C_1)\} = \{c, c, a\}$
$S(C_2) = \{H_1(C_2), H_2(C_2), H_3(C_2)\} = \{f, d, d\}$
$S(C_3) = \{H_1(C_3), H_2(C_3), H_3(C_3)\} = \{f, d, d\}$
$S(C_4) = \{H_1(C_4), H_2(C_4), H_3(C_4)\} = \{c, c, a\}$

In other words, we have used a hash function $H()$ that converts each set gym member into a signature (here is a single row name or fitness activity), in each permutation. The signature is the set of hash values for each permutation, but it is smaller than the original data and thus signatures fit into the memory better.

MinHashing

Step 3: The Jaccard function can compare two sets to each other but through their signatures only. It means that the Jaccard similarity uses the signature, instead of original data. For example:

$$J(C_1, C_2) = \frac{\{c, c, a\} \cap \{f, d, d\}}{\{c, c, a\} \cup \{f, d, d\}} = Null \text{ or}$$

$$J(C_2, C_3) = \frac{\{f, d, d\} \cap \{f, d, d\}}{\{f, d, d\} \cup \{f, d, d\}} = 1$$

We can conclude that member 1 (C_1) and member 2 (C_2) have no similarity and member 2 (C_2) and member 3 (C_3) are very similar.

In this example, the number of data is too small, but in real-world scenarios each set has many data objects and if we intend to compare the contents instead of the signature it is not memory efficient.

Java Examples of MinHashing

- https://mymagnadata.wordpress.com/2011/01/04/minhash-java-implementation/

- https://massivealgorithms.blogspot.com/2014/12/matts-blog-minhash-for-dummies.html

Outline

- Java Memory Structure
- Algorithm Complexity
- Search Improvement Methods
- **Read and Write Optimization (SSTables, LSM-Tree and Bloom Filter)**
- Lucene

Types of Search Engines

1. Log Structured Search Engines

2. Page Structured Search Engines

Log Structured Search Engines

- Many databases are using log structures internally. Lucene is an example of log structure search engine.

- Logs are append only.

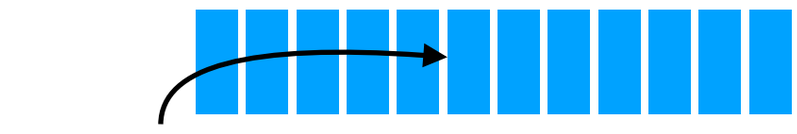
- To read data from logs, the search engine should start from the beginning and search the entire dataset completely.

- Therefore, an index is very helpful to retrieve information from such a data structure.

Log Structure Data

- Writing

- Searching



Why Log Structures are very useful?

- **Concurrency** and **crash recovery** are very easy, because all data is always available.

- Just supporting sequential write operation makes the process very fast and there is **no need to use update lock** on the data.

Compaction in Log Structured Search

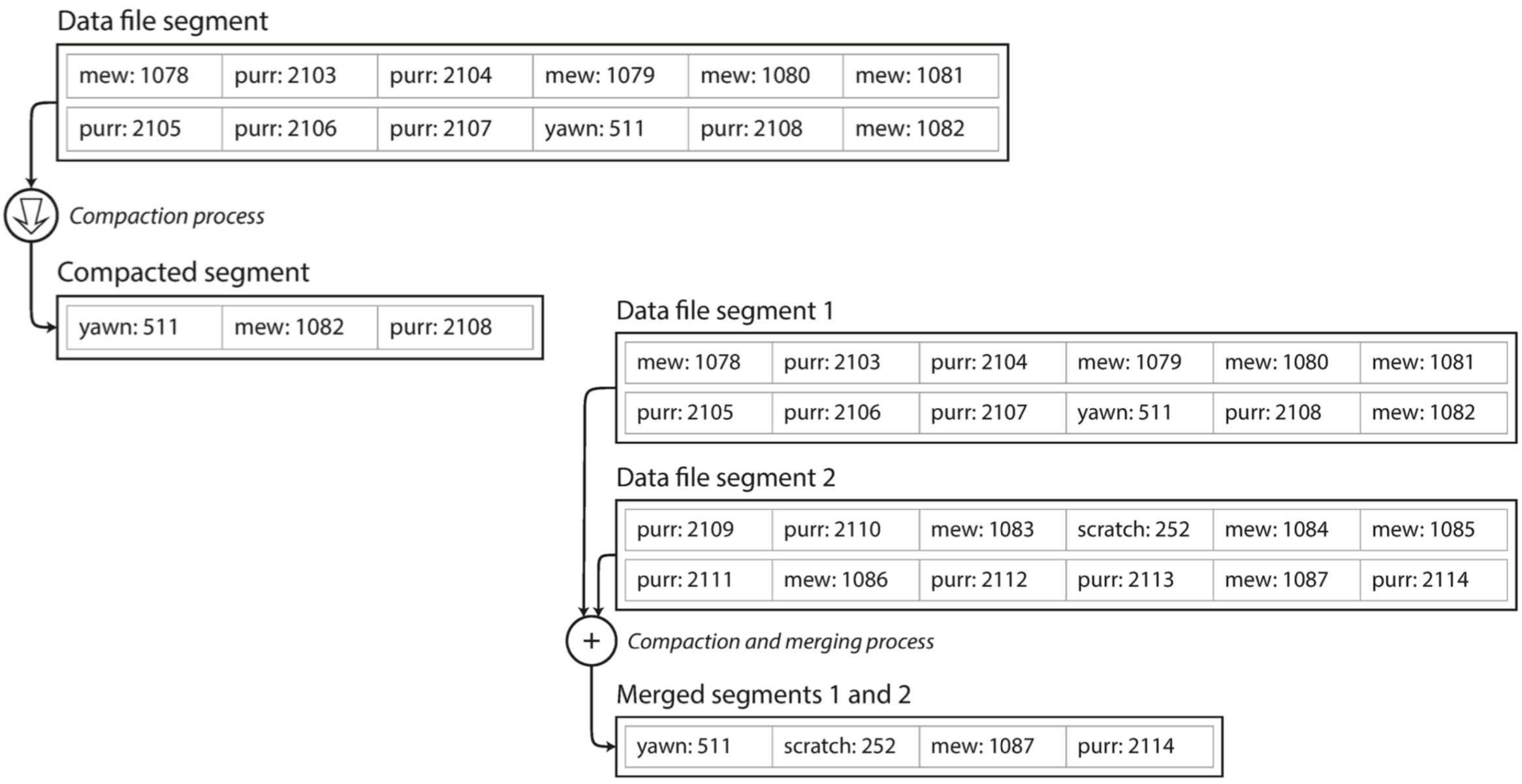
- The challenges of log files are that they are append only, and **update or even searches are too time consuming**, which is inefficient. Therefore, logs are usually just added data (no remove). Just adding could make the dataset very large.

- One approach to mitigate the disk space limitation is to **segment the log into smaller log chunks**. In other words, we break logs into segments and close each segment when it reaches a certain threshold.

- Then, we can perform compaction on each segment. **Compaction** (in this context) means removing duplicates and keeping the most recent updates only.

- We can **merge** compacted log files to save more space.

Compaction & Merging Example



Source: Designing Data Intensive Applications (Martin Kleppman ‘18)

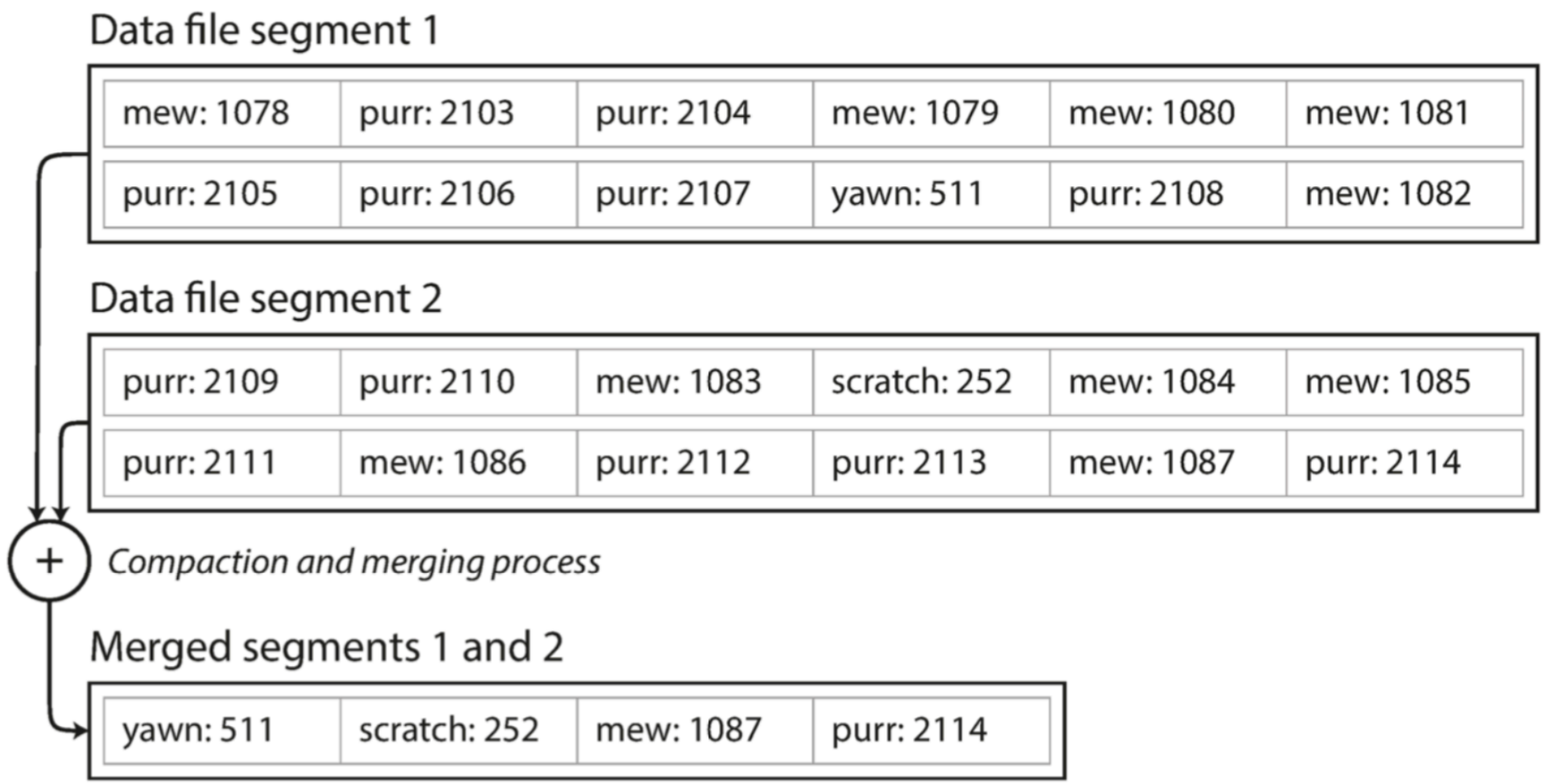
SSTables

- Each logs in a log structured table is written in a sequential order.
- If we sort sequences based on a key, we call it **Sorted String Table** or **SSTable**.

*SSTable is a **sorted list of <key,value>** pairs, where the keys do not have to be unique.*

Advantages of SSTable

1. Merging segments is a much simpler and efficient process, even if segment files are large to load into the memory.



Source: Designing Data Intensive Applications (Martin Kleppman ‘18)

Advantages of SSTable

2. To find a particular key-value pair we **do not need to keep the data in the memory**. Since we have the data sorted in the disk*, we know the approximate placement of the data* and look for its relevant segment.

   However, still, we need to store indexes inside memory, but only indexes to data segments and not the entire dataset.

3. Since data are sorted, we can compress them before preparing them for storage. Therefore, less I/O operation will be needed.

Log Structured Merging Tree (LSM-Tree)

- LSM is implementing a search as a log-shaped, but on trees.

- In other words, it appends to the tree, similar to the append that is being done in log structures data.

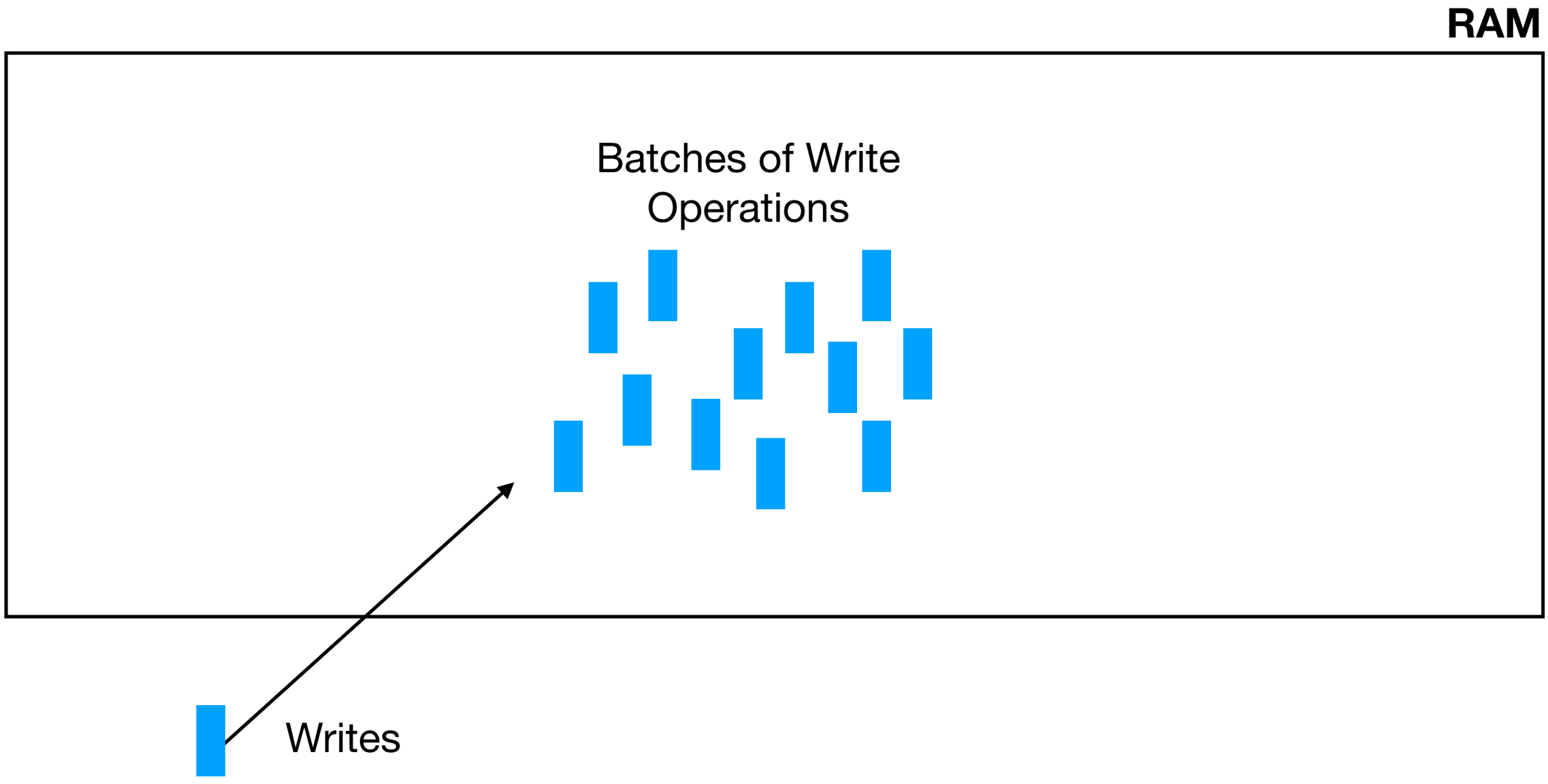
- First, it **batches write operations into the memory**.

- Second, it **sorts them** and puts them into a buffer.

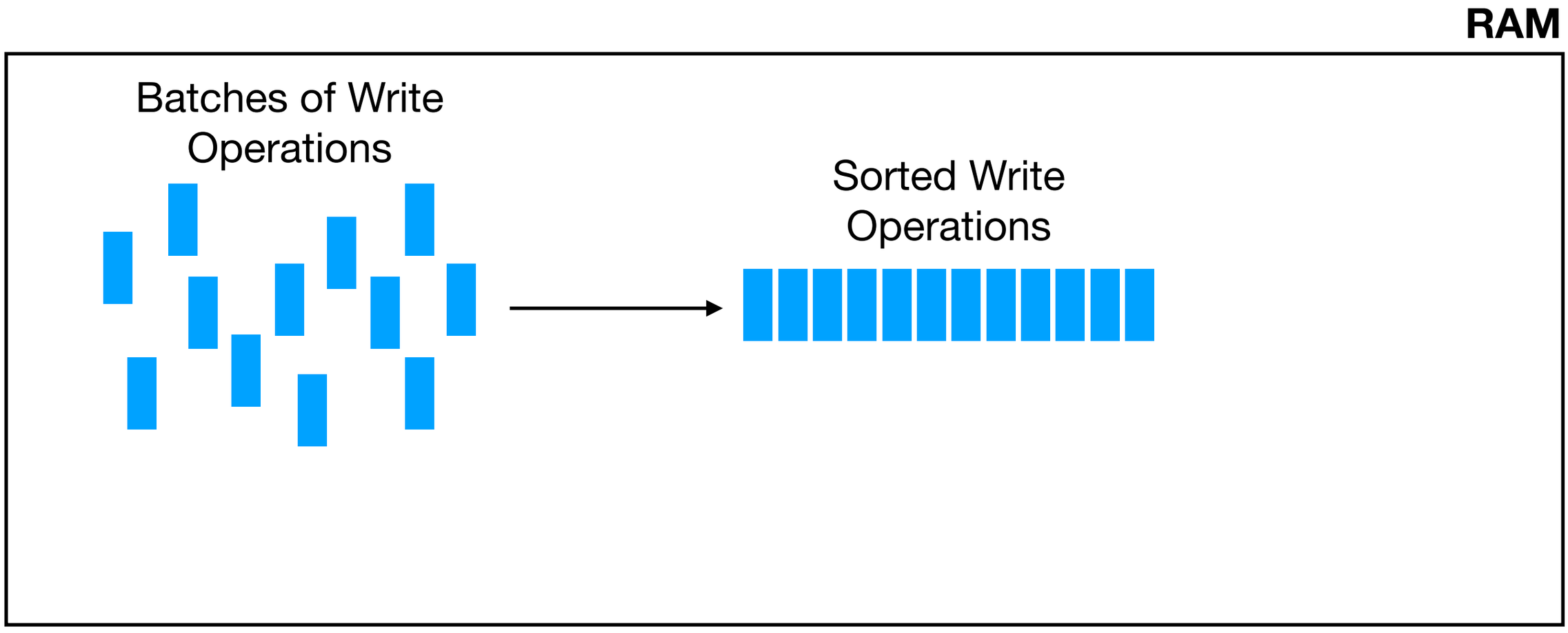
- Third, once we have **collected all required write batches**, then it **persists these batches into the disk**.

- It is naturally **optimized for a write operation and not a read operation**.

Log Structured Merging Tree (LSM-Tree)

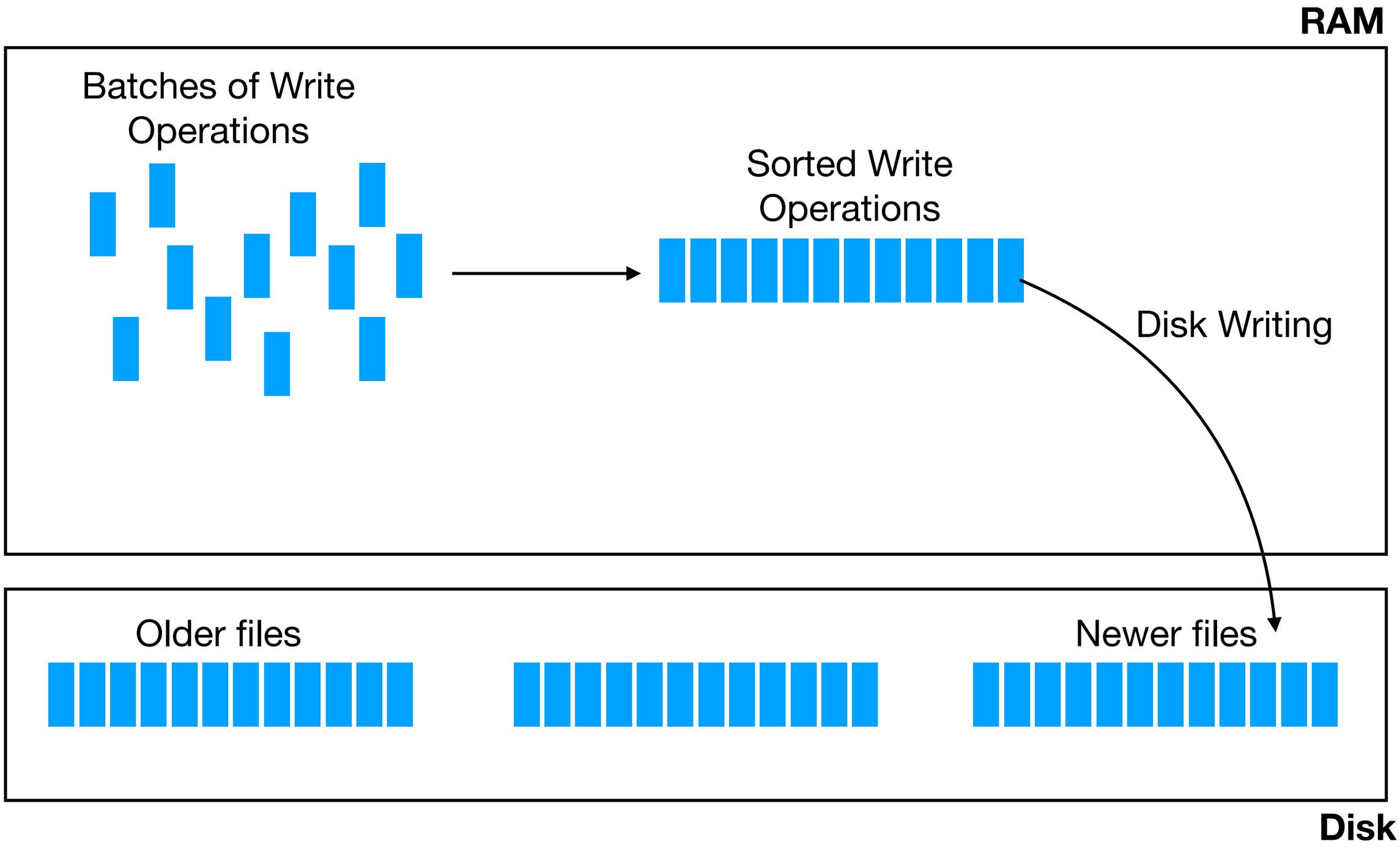


Log Structured Merging Tree (LSM-Tree)



Sorted Write Tables = Sorted **Memtables**

Log Structured Merging Tree (LSM-Tree)

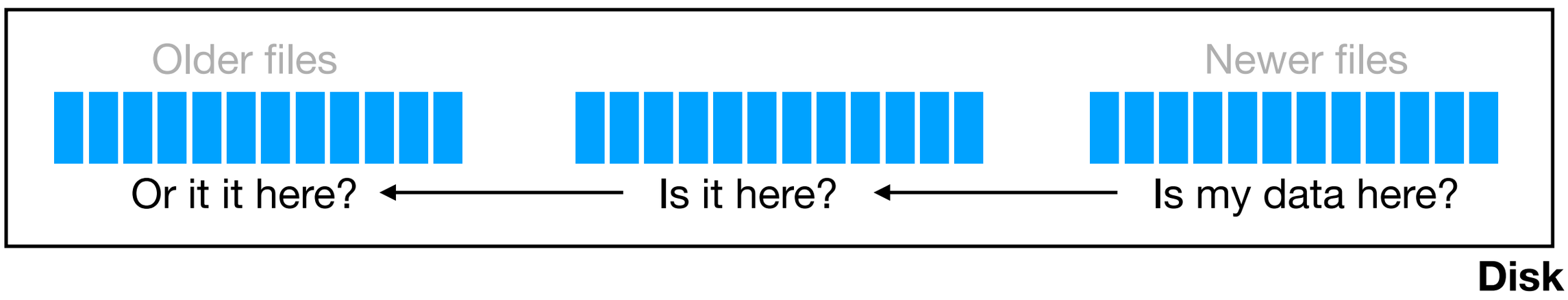


Some useful hints

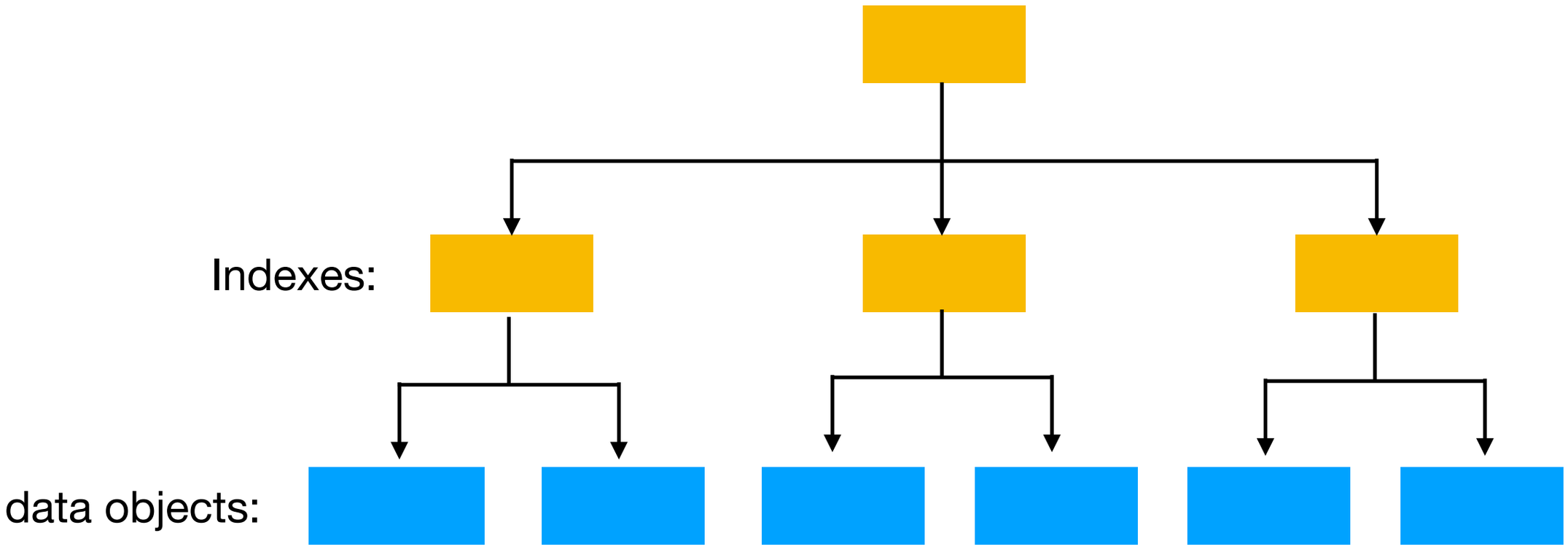
- In general, optimizing the reading process is usually easier because we have lots of freedom to choose the method we need.

- When it comes to writing, we don’t have many options to optimize the writing process.

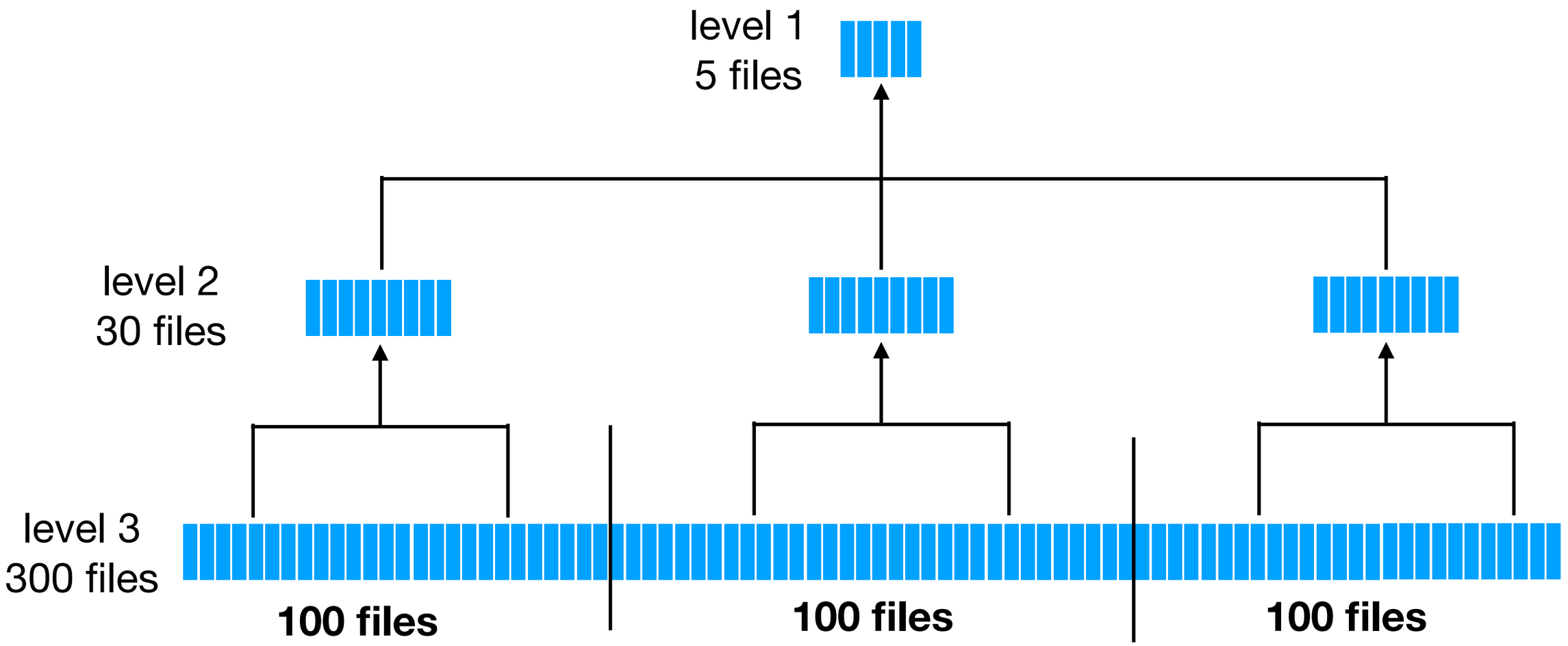
- The LSM-Tree is not very optimized for the read operation, but it is optimized for writing, to improve it for reading, we can use bloom filter or Tree based indexes.



Tree based Indexes



Leveled Compaction



Bloom Filter

- It is an efficient data structure that tell us **if a particular data existed in a set or not.** Its output can tell us (i) the target data object does not exist in the set, or (ii) the target data object might exist in the set.

- It is memory efficient and fast, which is useful for many different types of application.

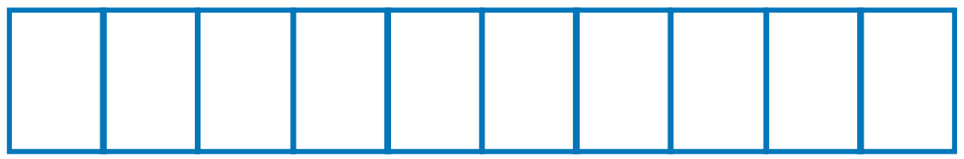
- Its base structure is a **bit vector** (each cell is a bit).

Bloom Filter

- When we add a data into the bit vector, it transfers the original data from a hash function into a bit vector and set some bits in its bit array. Then, later we can check whether the data existed inside Bloom filter, by checking the bits which are set to 1.

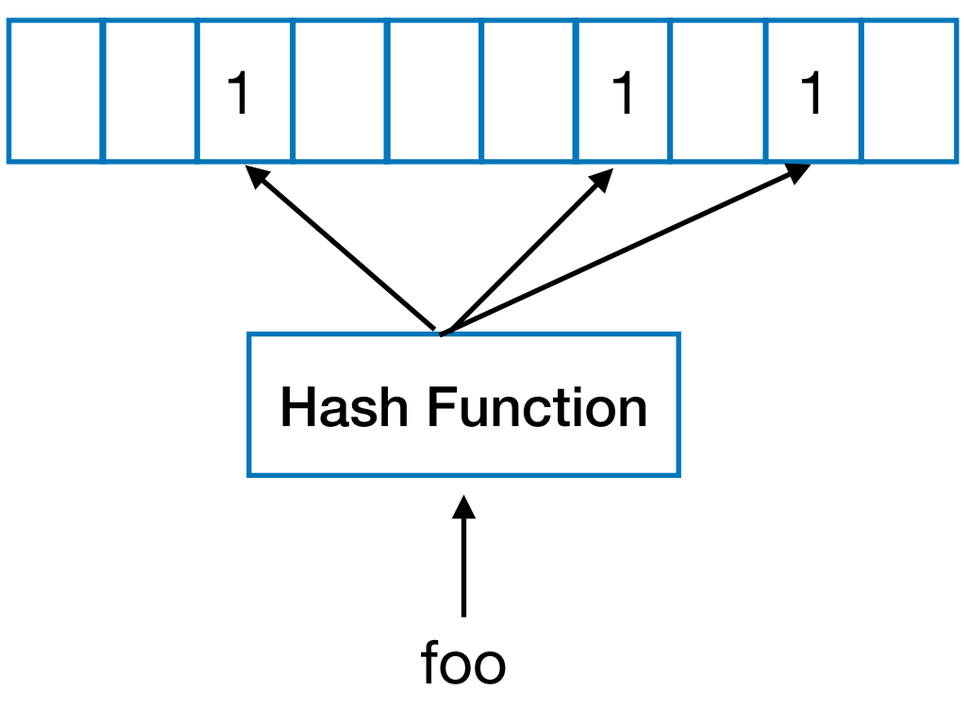
Bloom Filter

1. Empty bit vector



Database: —

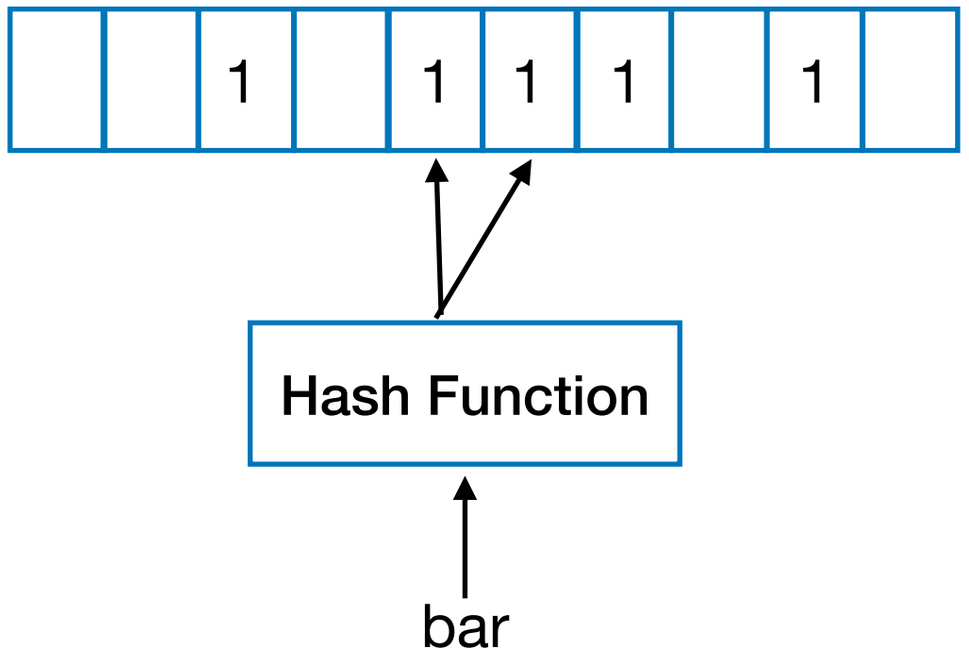
2. We add a string, e.g. “foo” into the bloom filter



Database: foo

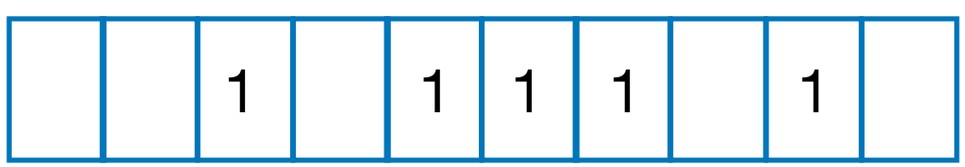
Bloom Filter

3. We add another string, e.g. “bar” into the bloom filter



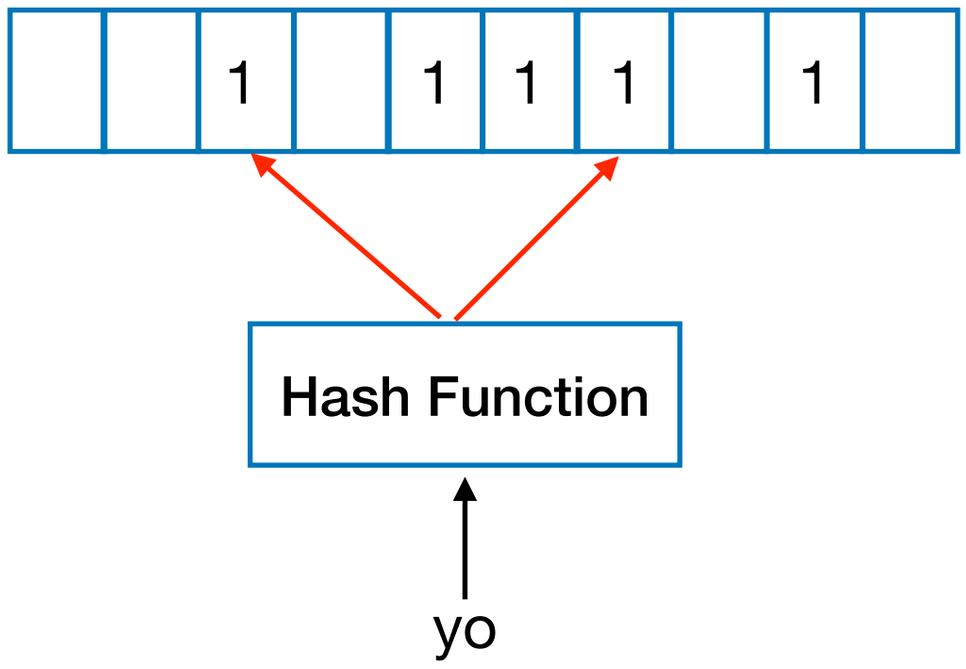
Database:
foo
bar

4. Now our bit vector includes both “bar” and “foo”.



Bloom Filter

5. If we query for “foo”, or even “fo” it uses the hash function to convert them to bit stream. Then the algorithm goes and check its bit vectors. If those bits are set to 1, then it returns “**maybe**” they exist in the Bloom filter structure.



Database:
foo
bar

6. If we query for “x”, it again uses the hash function to convert it to a bit stream. Then the algorithm goes and check its bit vectors. If its related bits are not set to 1, then it returns no.

Bloom Filter Example

- Here is the link: https://www.javamex.com/tutorials/collections/bloom_filter_java.shtml

- Bloomfilter visualization

https://www.jasondavies.com/bloomfilter

Combination of Bloom filter and Huffman encoding

- Smartphone: https://www.youtube.com/watch?v=ySfvkJIoDIE
- Smartwatch: https://www.youtube.com/watch?v=XSA8atKUvVM

Outline

- Java Memory Structure
- Algorithm Complexity
- Search Improvement Methods
- Read and Write Optimization (SSTables, LSM-Tree and Bloom Filter)
- **Lucene**

- Lucene is an open source java engine, which supports fast indexing mechanism, and it is provided by Apache organization.

- It is a library that can be added as jar file to a java project and encapsulates lots of underlying code for performance optimization.

- It creates indices for documents and instead of searching the content of raw documents it searches indices.

- Lucene uses the inverted index to search a document.

Inverted Index

strings, or they may be complex structures. Programmers usually need to define their own data types, although a number of other libraries such as Protocol Buffers, Thrift, and Avro may simplify the task.

3.4 Hadoop Limitations

Generally each node in a Hadoop cluster may typically have a few gigabytes of memory. However, if the input dataset is several terabytes, then this would require a thousand or more computers to hold the data in the RAM, and then, no single machine would be able to process all of the data. Although, hard drives are much bigger, and a single machine can these days hold multiple terabytes of information on its hard drives. The intermediate result generated from a data processing task in a large-scale computation can quickly fill up more space than what the original input data has required. Also, during data processing, some of the hard drives employed by the system may become full, and the system may need to route this data to other nodes that can store the overflow.

Finally, bandwidth is limited even for an internal network. While a set of nodes are directly connected by a high capacity network link, when all of the nodes try to transmit multi-gigabyte of data, they could easily saturate the network capacity. Furthermore, the remote procedure calls and other data transfer requests which are using a channel may be delayed or simply dropped. To maintain successfully a robust distributed system, a large-scale distributed system must be able to manage its resources efficiently. For example, it must allocate some of its resources for maintaining the system as a whole, while devoting as much time as possible to the computations.

4. IMPLEMENTATION

This section presents the implementation of the E-Book content indexing system, which represents the inverted index implementation using MapReduce within the Hadoop environment. In this section we show three different implementations of inverted index in MapReduce: Indexer, IndexerCombiner, and IndexerMap.

4.1 Indexer

The Indexer is the simplest implementation of inverted index. Like any other MapReduce programs, this implementation consists of two classes (Map and Reduce). As for the input of mapper class, Hadoop splits data based on the input format, which is specified in job configuration. In this project the input format is TextInputFormat, which means each file in the dataset has at least one split. Split data size should be less than 64 MB (default Block size). If a file size is more than the block size, then Hadoop splits the file into more than one split. These splits are the inputs for the mapper class. In the map class, the mapper function tokenizes the words within a file as actual text and obtains the name of the file. At the end of this process it creates <word, file name> output for counting the occurrence of each word, then it forwards the output to the reducers. The output of mapper is also called intermediate key [10].

4.2 IndexerCombiner

In the IndexerCombiner implementation, the mapper class has extra responsibility to partially reduce the outputs in the mapper phase. This reduction is called combiner. The mapper is not based on creating an output <Word, Page name> for each word that it finds. Instead, the mapper counts the occurrence of the words in each split and creates an output <word, <file name: number>> to forward it to reducer. Therefore, the numbers of intermediate keys are fewer in comparison with Indexer. The reducer class for IndexerCombiner is very similar to reducer class in Indexer. However, these reducers handle the list of values differently.

4.3 IndexerMap

The Apache Hadoop's SequenceFile provides the data structure for the key-value pairs. This data structure is append-only. That means it cannot be edit or remove a specific element from the file.

The MapFile is a directory that contains two SequenceFile files: the data file ("/data") and the

ISBN: 978-0-9891305-4-7 ©2014 SDIWC

Index

Page numbers in **bold face** refer to key term definitions
Page numbers in *italics* refer to images or diagrams
Page numbers followed by a "t" indicate a table

A

absolute temperature scale, **350**–351
absolute zero, **351**
acceleration of gravity, A.23t
accuracy, A.5
acetic acid (CH_3COOH)
  buffers, 575–576, 581–582
  conjugate acid-base pairs, 540
  ionization constant, 553, 554t
  manufacture of, 451
  titrations, 590–592
  as weak acid, 144t, 145, 551–552
acid-base pairs, conjugate, **540**–544
acid-base reactions, **538**
  autoionization of water, 545–547
  gas-forming exchange, 150–151
  net ionic equations for, 148–150
  neutralization, 146–150, 561–566t
  of salts, 146–151, 561–566
acid-base titrations, 587–594
acidic solutions, **546**, 650–652, A.32t–A.33t
acid ionization constant expressions, **550**–551
acid ionization constants (K_a). *See* ionization constants, acids (K_a)
acidosis, 576
acids, **143**. *See also* acid-base reactions; ionization constants, acids (K_a); *specific entries, e.g.* carboxolic acids, lactic acids
  Brønsted-Lowry concept, 538–544
  buffer solutions, 575–586
  conjugate acid-base pairs, **540**–544
  equilibrium constants, 475t
  ionization constants, 550–561, A.28t
  Lewis, 566–568
  organic, 544
  pH scale, 547–550
  properties, 143–145
  solubility of salts, 597–598
  solutions, **546**, 650–652, A.32t–A.33t
  strengths, 145–146, 555–556
  titrations, 587–594
  water's role, 540
actinides, **55**, 250–251
activated complex, **433**
activation energies (E_a), **434**, 438–440, 445, 447, 449–450
active sites, **449**
activities, 466, 547, **703**–706
activity series, metal, **159**–160
actual yields, **121**
addition, 45, A.6, A.9
addition, significant figures in, A.6
air, 342–343, 366–370, 380–381, 706
alkyl groups, **70**–71
alcohols, 64, 505–507
aldehydes, 278–279
alkali metals, **55**, 106
alkaline batteries, 670
alkaline earth metals, **55**
alkaline fuel cells, 674
alkalosis, 576
alkanes, **68**–71, 277–278, A.25–A.26
alkenes, **280**–283, A.26–A.27
alkyl group, **70**–71, A.25–A.26
alkynes, **281**, A.27
allotropes, **23**–24, 208, 403–*405*
alpha particles, 38–39, **693**–696, *697*, 699–700
alpha radiation, **693**
alpha rays, 36–37
aluminum (Al), 7, 8t, 103, 634–635, 682
amines, 544–**545**
ammonia (NH_3)
  amines, 545
  Brønsted-Lowry base, 538–539
  complex ions, 567
  ionization constant, 554t, 561
  standard molar enthalpy of formation, 210t
  structure, 64t
  synthesis, 107–108, 462, 494–495
  VSEPR model, 312–313
  as weak base, 145, 539
ammonium ions, 74, 77
amorphous solids, **396**
ampheres (A), **658**
amphiprotic species, **540**
amphoteric metal hydroxides, **567**, 602
amplitude, 223
amu (atomic mass unit), **46**
analytical chemistry, 114
Anderson, Carl, 696
angular geometries, 314
anions, **72**, 76, 78, 257–258
anodes, 37, **654**. *See also* electrochemistry
anodic inhibition, **683**–684
antibonding molecular orbitals, **298**
antimatter/antielectrons, 696
aqueous equilibria. *See also* acid-base titrations; buffers
  factors affecting solubility, 597–602
  precipitation, 603–604
  solubility product constant, 594–597
aqueous solutions (aq), **100**. *See also* buffer solutions; solutions
  electrolysis, 675–678
  electrolytes, 82–83, 136, 527–528, 654–655
  ionic compounds, 136–139
  molarity, 166–168
  standard reduction potentials in, A.32t–A.34t
  stoichiometric relationships in, *167*

Lucene Data Entities

- **Documents:** A document in Lucene is a unit of search and each index consists of one or more documents.
  - The process of indexing involves adding documents to `indexWriter.`
  - The process of searching involves retrieving documents from an index via `IndexSearcher`.
- **Fields:** A field is a **name-value pair**, and a document is composed of many **fields.**
  - The process of indexing in Lucene involves creating documents, which each has many fields.
- **Searching:** The process of searching starts with creating a query (via `QueryParser`) and handling this query to a `IndexSearcher`. The `IndexSearcher` returns the list of hits.
- **Queries:** Lucene has a small set of keywords, used to define queries including binary operators (AND, OR, NOT), …

How does a search engine work?

1. Acquire raw content
2. Build the document
3. Analyze the document
4. Index the document
5. Provide a UI for search
6. Parse the user’s query
7. Execute the query by extracting data from the underlying dataset
8. Renders the result for the user

Searching Classes

- **IndexSearcher:** It is the main class that reads indexes after the indexing process.

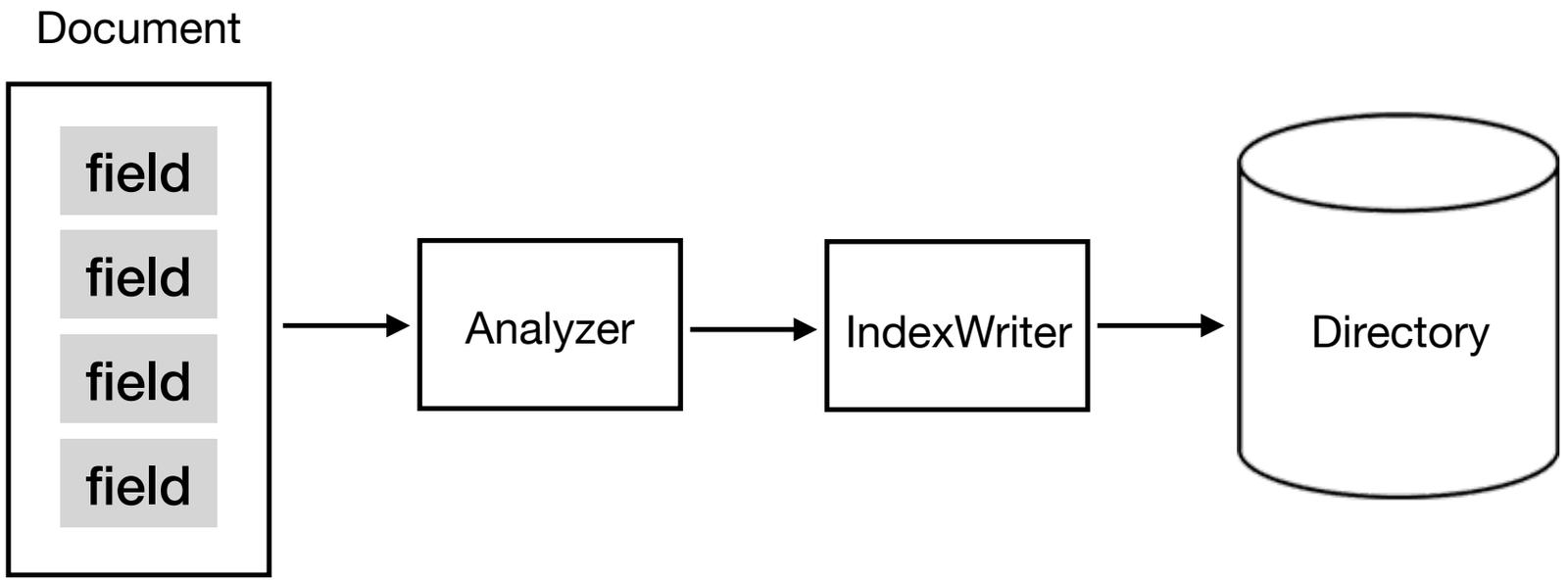
- **Term:** It looks for fields (smallest search unit)

- **Query:** It is an abstract class which includes lots of children classes used to create the query for searching.

- **TermQuery:** constructs a query for the given term.

- **TopDocs:** returns the top *n* hits that matches the search criteria.

Common Lucene Classes and Concepts

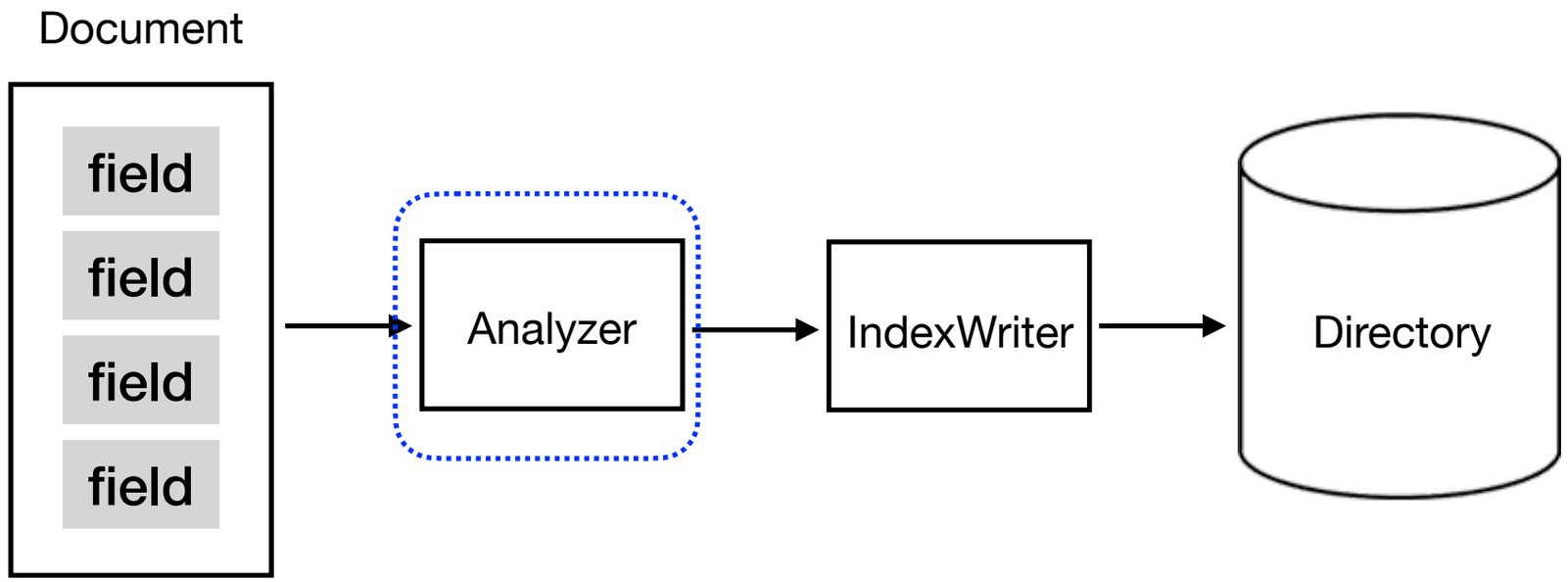


- **Field:** It is the smallest unit of data and presented as key-value pair. The key is used to identify the value to be indexed.
- **Document:** It presents the input dataset (as a set of fields) that will be given to the Analyzer.
- **Analyzer:** It analyzes the given document and extract tokens (fields) which is need to be indexed.
- **IndexerWriter:** It is the core of Lucene which makes indexes for the given document.
- **Directory:** It class represents the storage location of the indexes.

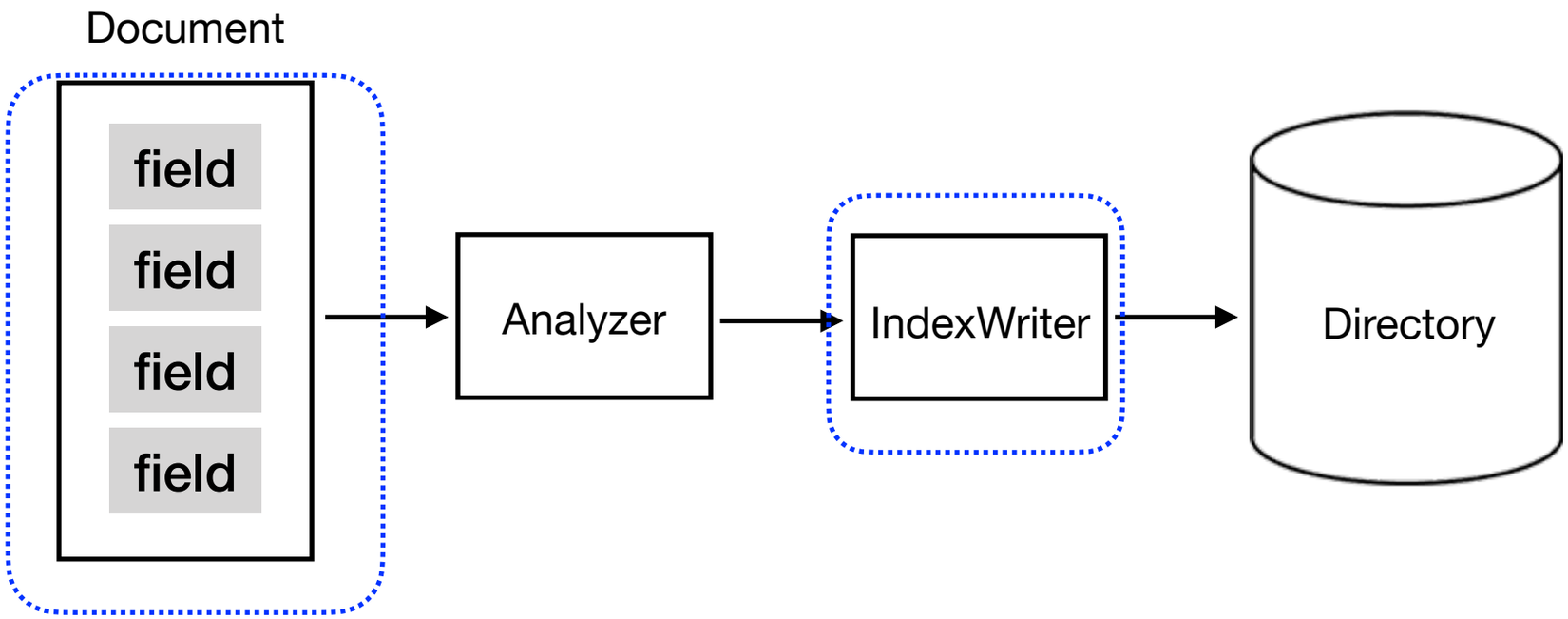
Hello World in Lucene

Please copy and paste this code in your IDE

```
package edu.bu.met622.lucene;

import java.io.IOException;
import java.nio.file.Paths;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.document.StringField;
import org.apache.lucene.document.TextField;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.queryparser.classic.ParseException;
import org.apache.lucene.queryparser.classic.QueryParser;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.FSDirectory;


public class HelloworldExample {
	public static void main(String[] args){
	try {
	 // 0. Specify the analyzer for tokenizing text.  The same analyzer should be used for indexing and searching
	 StandardAnalyzer analyzer = new StandardAnalyzer();

		// 1. create the index
	//Directory index = new Directory(Files.createTempDirectory("XXXXX"));
		//Directory index = new ByteBuffersDirectory();
	 FSDirectory index = FSDirectory.open(Paths.get("/Users/rawassizadeh/WORK/eclipseworkspace/CSMET622"));

	 IndexWriterConfig config = new IndexWriterConfig(analyzer);

	 IndexWriter w = new IndexWriter(index, config);
	 addDoc(w, "Lucene in Action", "193398817");
	 addDoc(w, "Lucene for Dummies", "55320055Z");
	 addDoc(w, "Managing Gigabytes", "55063554A");
	 addDoc(w, "The Art of Computer Science", "9900333X");
	 addDoc(w, "Chicken and Data: Machine Leanrning" ,"2343235X");
	 w.close();

	    // 2. query
	    String querystr = new String("Lucene in Action"); //

	    // the "title" arg specifies the default field to use
	    // when no field is explicitly specified in the query.
	    Query q = new QueryParser("title", analyzer).parse(querystr);

	    // 3. search
	    int hitsPerPage = 5;
	    IndexReader reader = DirectoryReader.open(index);
	    IndexSearcher searcher = new IndexSearcher(reader);
	    TopDocs docs = searcher.search(q, hitsPerPage);
	    ScoreDoc[] hits = docs.scoreDocs;

	    // 4. display results
	    System.out.println("Found " + hits.length + " hits.");
	    for(int i=0; i<hits.length; ++i) {
	        int docId = hits[i].doc;
	        Document d = searcher.doc(docId);
	        System.out.println((i + 1) + ". isbn: " + d.get("isbn") + "\t title: " + d.get("title"));
	    }

	    // reader can only be closed when there
	    // is no need to access the documents any more.
	    reader.close();

	   } catch(IOException ex){
	         ex.printStackTrace();
	    } catch (ParseException ex2) {
	                 ex2.printStackTrace();
	    }
	}

	private static void addDoc(IndexWriter w, String title, String isbn) throws IOException {
	    Document doc = new Document();
	    doc.add(new TextField("title", title, Field.Store.YES));

	    // use a string field for isbn because we don't want it tokenized
	    doc.add(new StringField("isbn", isbn, Field.Store.YES));
	    w.addDocument(doc);
	}
}
```

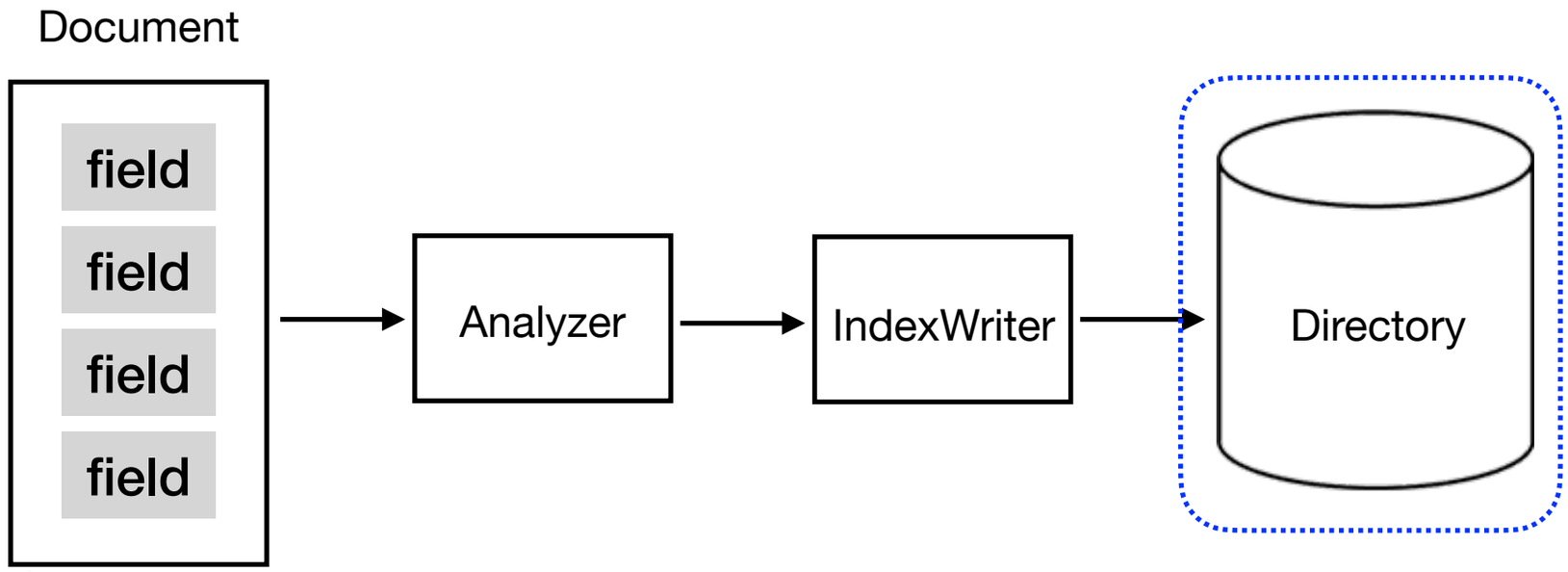



```
public class HelloworldExample {
public static void main(String[] args){
try {
    // 0. Specify the analyzer for tokenizing text.
    //    The same analyzer should be used for indexing and searching
    StandardAnalyzer analyzer = new StandardAnalyzer();

   // 1. create the index
   FSDirectory index = FSDirectory.open(Paths.get("/Users/rawassizadeh/WORK/eclipseworkspace/CSMET622"));
   IndexWriterConfig config = new IndexWriterConfig(analyzer);
   IndexWriter w = new IndexWriter(index, config);
   addDoc(w, "Lucene in Action", "193398817");
   addDoc(w, "Lucene for Dummies", "55320055Z");
   addDoc(w, "Managing Gigabytes", "55063554A");
   addDoc(w, "The Art of Computer Science", "9900333X");
   w.close();

   // 2. query
   String querystr = new String("Chicken");

   // the "title" arg specifies the default field to use when no field is explicitly specified in the query.
   Query q = new QueryParser("title", analyzer).parse(querystr);

   // 3. search
   int hitsPerPage = 10;
   IndexReader reader = DirectoryReader.open(index);
   IndexSearcher searcher = new IndexSearcher(reader);
   TopDocs docs = searcher.search(q, hitsPerPage);
```




```
// 1. create the index
FSDirectory index = FSDirectory.open(Paths.get("/Users/rawassizadeh/WORK/eclipseworkspace/CSMET622"));
IndexWriterConfig config = new IndexWriterConfig(analyzer);
IndexWriter w = new IndexWriter(index, config);
addDoc(w, "Lucene in Action", "193398817");
addDoc(w, "Lucene for Dummies", "55320055Z");
addDoc(w, "Managing Gigabytes", "55063554A");
addDoc(w, "The Art of Computer Science", "9900333X");
w.close();
```

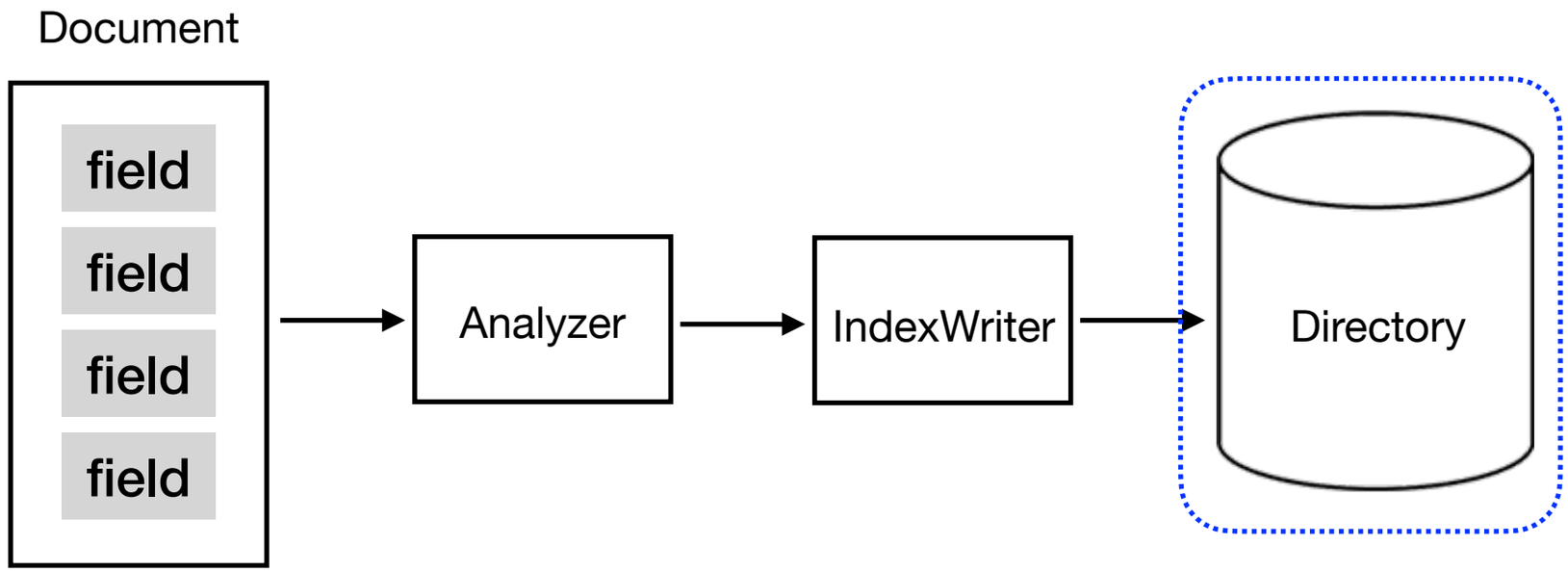
```
private static void addDoc(IndexWriter w,
                           String title,
                           String isbn)
                    throws IOException {
    Document doc = new Document();
    doc.add(new TextField("title", title,
            Field.Store.YES));

    // use a string field for isbn because we
       don't want it tokenized
    doc.add(new StringField("isbn", isbn,
                            Field.Store.YES));
    w.addDocument(doc);
```




```
ic class HelloworldExample {
 public static void main(String[] args){
 try {
  // 0. Specify the analyzer for tokenizing text.
  //    The same analyzer should be used for indexing and searching
  StandardAnalyzer analyzer = new StandardAnalyzer();

  // 1. create the index
  Directory index = new RAMDirectory();
  IndexWriterConfig config = new IndexWriterConfig(analyzer);
  IndexWriter w = new IndexWriter(index, config);
  addDoc(w, "Lucene in Action", "193398817");
  addDoc(w, "Lucene for Dummies", "55320055Z");
  addDoc(w, "Managing Gigabytes", "55063554A");
  addDoc(w, "The Art of Computer Science", "9900333X");
  w.close();
```

```
    // 2. query
    String querystr = new String("Lucene");

    // the "title" arg specifies the default field to use when no field is explicitly specified in the  query
    Query q = new QueryParser("title", analyzer).parse(querystr);

    // 3. search
    int hitsPerPage = 10;
    IndexReader reader = DirectoryReader.open(index);
    IndexSearcher searcher = new IndexSearcher(reader);
    TopDocs docs = searcher.search(q, hitsPerPage);
    ScoreDoc[] hits = docs.scoreDocs;
```

```
// 4. display results
System.out.println("Found " + hits.length + " hits.");
for(int i=0;i<hits.length;++i) {
    int docId = hits[i].doc;
    Document d = searcher.doc(docId);
    System.out.println((i + 1) + ". " + d.get("isbn") + "\t" + d.get("title"));
}

// reader can only be closed when there
// is no need to access the documents any more.
reader.close();
```




```
// 3. search
 int hitsPerPage = 10;
 IndexReader reader = DirectoryReader.open(index);
  IndexSearcher searcher = new IndexSearcher(reader);
  TopDocs docs = searcher.search(q, hitsPerPage);
  ScoreDoc[] hits = docs.scoreDocs;

     // 4. display results
     System.out.println("Found " + hits.length + " hits.");
     for(int i=0;i<hits.length;++i) {
         int docId = hits[i].doc;
         Document d = searcher.doc(docId);
         System.out.println((i + 1) + ". " + d.get("isbn") + "\t" + d.get("title"));
     }

     // reader can only be closed when there
     // is no need to access the documents any more.
     reader.close();

} catch(IOException ex){
    ex.printStackTrace();
 } catch (ParseException ex2) {
        ex2.printStackTrace();
```

Experiment to Read from File

- Download the data for this sample user: https://drive.google.com/file/d/0ByUEDpaqX-2fdlNQaEhUWmUzbm8/view?usp=sharing

- Read the content of the data for one user and its battery sensor, then index it via Lucene.


BYEEE

References

- https://www.infoq.com/presentations/lsm-append-data-structures/
- http://www.benstopford.com/2015/02/14/log-structured-merge-trees/
- http://www.lucenetutorial.com/lucene-query-syntax.html
- http://www.tutorialspoint.com/lucene