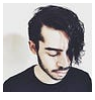
shahinrahbariasl.medium.com

Data Engineering Cheat Sheet - Shahin Rahbariasl - Medium

Shahin Rahbariasl

12–15 minutes

Photo by Ovan from StockSnap



Data engineering is all about designing, developing, testing and delivering offerings using leading edge and/or proven technologies. Most companies nowadays work Agile, which makes it possible to have a collaborative environment to understand stakeholder requirements and develop valuable features. Data engineering also includes designing, coding, and testing innovative component-level software solutions. You always need to ensure that the implemented solutions are unit tested and ready to be integrated into their product.

As a data engineer, you're in charge of gathering and collecting the data, storing it, do batch processing or real-time processing on it, and serve it via an API to data scientist who can easily query it. Data engineers may work closely with data architects (to determine what data management systems are appropriate) and data scientists (to determine which data are needed for analysis). They often wrestle with problems associated with database integration and messy, unstructured data sets.

Here's a list of minimum topics that you need to have deep knowledge about as a data engineer. Keep in mind the intention of this article is not to explain the subjects in details, but to list important points about them. As an #IBM engineer, I catch myself using most of these tips pretty often. Hopefully, they'll be useful for you as well.

Data Structures

Array

A container object that holds a fixed number of values of a single type. Arrays are good for indexing, but not efficient for searching, inserting, and deleting.

Linear arrays: One dimensional arrays declared with a fixed

size.

Dynamic arrays: Hold reserved space for expansion (if it gets full, it copies itself to a larger array).

Multi dimensional arrays: Multi dimensional or nested arrays of arrays.

Linked List

A linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next. Linked lists are fast for insertion and deletion, but slow at indexing and searching.

Each node contains a value and a pointer to the next node. **Doubly linked** list nodes have pointers to the previous node as well. **Circularly linked list** has a tail (the last node), references the head (the first node).

Stack (LIFO) usually implemented with linked lists (the head being the only place for insertion and removal).

Queues (FIFO) can be implemented with linked lists as well (doubly linked list that only removes from head and adds to tail).

Time Complexity:

- Indexing: Linked Lists: O(n)
- Search: Linked Lists: O(n)
- Optimized Search: Linked Lists: O(n)
- Insertion: Linked Lists: O(1)

Hash Table or Hash Map

A data structure that implements an associative array abstract

data type, a structure that can map keys to values. A **hash table** uses a **hash** function to compute an index, also called a **hash** code, into an array of buckets or slots, from which the desired value can be found.

Hashing

Uses a **hash function** accept a key and return an output unique only to that specific key. An input data gets passed to the hash function and a much smaller data gets generated which can be a unique address in memory for that data.

- Hashing is not encryption
- Hashing functions are typically one-way
- Information can be lost when hashing
- Two objects that are equal should return the same hash
- The same hash may also result from different objects (**Hash Collision**)
- Hash tables are very fast!

  Hash collisions are when a hash function returns the same output for two distinct inputs. Linked lists can be used to implement list of objects for the same hash.

  Time Complexity:

- Indexing: Hash Tables: O(1)
- Search: Hash Tables: O(1)
- Insertion: Hash Tables: O(1)

Binary Tree

A **tree** data structure in which each node has at most two

children, which are referred to as the left child and the right child.

Binary Search Tree:

- A binary tree that uses comparable keys to assign which direction a child is.
- Left child has a key smaller than it's parent node.
- Right child has a key greater than it's parent node.
- There can be no duplicate node.

Time Complexity:

- Indexing: Binary Search Tree: O(log n)
- Search: Binary Search Tree: O(log n)
- Insertion: Binary Search Tree: O(log n)

Sorting

Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining elements.

Heap sort algorithm:

1. Build a max heap from the input array.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap and reduce the size of heap by 1. Finally, heapify the root of the tree.
3. Repeat step 2 while size of heap is greater than 1.

```java
public class HeapSort {
    public void sort(int arr[])
    {
        int n = arr.length;

        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        // One by one extract an element from heap
        for (int i = n - 1; i > 0; i--) {
            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }

    // To heapify a subtree rooted with node i which is
    // an index in arr[]. n is size of heap
    void heapify(int arr[], int n, int i)
    {
        int largest = i; // Initialize largest as root
        int l = 2 * i + 1; // left = 2*i + 1
        int r = 2 * i + 2; // right = 2*i + 2

        // If left child is larger than root
        if (l < n && arr[l] > arr[largest])
            largest = l;

        // If right child is larger than largest so far
        if (r < n && arr[r] > arr[largest])
            largest = r;

        // If largest is not root
        if (largest != i) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;

            // Recursively heapify the affected sub-tree
            heapify(arr, n, largest);
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    // Driver code
    public static void main(String args[])
    {
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        int n = arr.length;

        HeapSort ob = new HeapSort();
        ob.sort(arr);

        System.out.println("Sorted array is");
        printArray(arr);
    }
}
```

Source: GeeksforGeeks

Time Complexity: Time complexity of heapify is O(Logn). Time complexity of building max heap is O(n) and overall time complexity of Heap Sort is O(nLogn).

Quick Sort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater

elements (greater than x) after x.

```
    /* The main function that implements QuickSort()
    arr[] --> Array to be sorted,
    low --> Starting index,
    high --> Ending index */
    void sort(int arr[], int low, int high)
    {
        if (low < high)
        {
            /* pi is partitioning index, arr[pi] is
            now at right place */
            int pi = partition(arr, low, high);

            // Recursively sort elements before
            // partition and after partition
            sort(arr, low, pi-1);
            sort(arr, pi+1, high);
        }
    }
```

```
// Java program for implementation of QuickSort
class QuickSort
{
    /* This function takes last element as pivot,
    places the pivot element at its correct
    position in sorted array, and places all
    smaller (smaller than pivot) to left of
    pivot and all greater elements to right
    of pivot */
    int partition(int arr[], int low, int high)
    {
        int pivot = arr[high];
        int i = (low-1); // index of smaller element
        for (int j=low; j<high; j++)
        {
            // If current element is smaller than the pivot
            if (arr[j] < pivot)
            {
                i++;

                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;

        return i+1;
    }
```

Source: GeeksforGeeks

Worst case: O(n2). when the partition process always picks greatest or smallest element as pivot.
Best case: O(nLogn). when the partition process always picks the middle element as pivot.

Merge Sort

Merge sort divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

```
/* Java program for Merge Sort */
class MergeSort
{
    // Merges two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    void merge(int arr[], int l, int m, int r)
    {
        // Find sizes of two subarrays to be merged
        int n1 = m - l + 1;
        int n2 = r - m;

        /* Create temp arrays */
        int L[] = new int[n1];
        int R[] = new int[n2];

        /*Copy data to temp arrays*/
        for (int i = 0; i < n1; ++i)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[m + 1 + j];

        /* Merge the temp arrays */

        // Initial indexes of first and second subarrays
```

```
        int i = 0, j = 0;

        // Initial index of merged subarry array
        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            }
            else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        /* Copy remaining elements of L[] if any */
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        /* Copy remaining elements of R[] if any */
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
```

```
    // Main function that sorts arr[l..r] using
    // merge()
    void sort(int arr[], int l, int r)
    {
        if (l < r) {
            // Find the middle point
            int m = (l + r) / 2;

            // Sort first and second halves
            sort(arr, l, m);
            sort(arr, m + 1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }
```

Source: GeeksforGeeks

Time complexity of Merge Sort is θ(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Bubble Sort

Bubble sort repeatedly swaps the adjacent elements if they are in wrong order.

```
class BubbleSort
{
    void bubbleSort(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n-1; i++)
            for (int j = 0; j < n-i-1; j++)
                if (arr[j] > arr[j+1])
                {
                    // swap arr[j+1] and arr[j]
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
    }
```

Source: GeeksforGeeks

Worst and Average Case: O(n*n). Worst case occurs when array is reverse sorted.
Best Case: O(n). Best case occurs when array is already sorted.

Searching

Breadth First Search

An algorithm for traversing or searching tree or graph data structures. It starts at the tree root and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

It uses the opposite strategy of depth-first search, which instead explores the node branch as far as possible before being forced to backtrack and expand other nodes.

- It finds every node on the same level, most often moving left to right.
- While doing this it tracks the children nodes of the nodes on the current level.
- When finished examining a level it moves to the left most node on the next level.
- The bottom-right most node is evaluated last (the node that is deepest and is farthest right of it's level).
- Optimal for searching a tree that is wider than it is deep.
- Uses a queue to store information about the tree while it traverses a tree.
- Because it uses a queue it is more memory intensive than depth first search.
- The queue uses more memory because it needs to store pointers

Time Complexity:

- Search: Breadth First Search: O(V + E)

- E is number of edges
- V is number of vertices

Depth First Search

An algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking.

- It traverses left down a tree until it cannot go further.
- Once it reaches the end of a branch it traverses back up trying the right child of nodes on that branch, and if possible left from the right children.
- When finished examining a branch it moves to the node right of the root then tries to go left on all it's children until it reaches the bottom.
- The right most node is evaluated last (the node that is right of all it's ancestors).
- Optimal for searching a tree that is deeper than it is wide.
- Uses a stack to push nodes onto.
- Because a stack is LIFO it does not need to keep track of the nodes pointers and is therefore less memory intensive than breadth first search.
- Once it cannot go further left it begins evaluating the stack.

Time Complexity:

- Search: Depth First Search: O(|E| + |V|)
- E is number of edges
- V is number of vertices

REST APIs

Representational state transfer (**REST**) is a de-facto standard for a software architecture for interactive applications that typically use multiple Web services.

- Uniform **stateless** operations
- No session information is retained by the server
- A request should not rely on another request
- **HTTP** methods used in **RESTful** APIs are **GET**, **PUT** (replace resource), **POST**, **PATCH** (update resource), **DELETE**

Advantages:

- Client-server architecture (separation of concern)
- Statelessness
- Cache-ability
- Layered system (eg. proxies should not prevent communication and no update to client/server code)
- Code on demand

Important Design Patterns

1. **Singleton**

- limit creation of a class to only one object
- Private/Protected constructor
- eg. used in caches, thread pools, registries

2. Factory method

- Produces class objects

- Decouples the code from the implementation class

3. Strategy

- Instead of directly implementing a single algorithm, the code receives runtime instructions to decide which of the group of algorithms to run

4. Observer

- There are observers (interested in updates) and subjects (generate updates)

5. Builder

- Builds more complex objects
- Step by step setting of properties vs factory method that creates objects in one go

6. Adaptor

- Adapter between two interfaces like translators
- eg. converting XML to JSON

7. State

- Allows object to alter its behaviour when its internal state changes
- eg. context for defining states

System Design

1. Outline use cases, constraints, and assumptions
   - Who, how, what, how much data, how many requests
2. High level design
   - Main components and connections
3. Design core components

4. Scale the design
   - Load balancer, horizontal/vertical scaling, sharding, caching, etc.

Microservices Architecture

A variant of the service-oriented architecture (SOA) structural style — arranges an application as a collection of loosely coupled services. In a micro services architecture, services are fine-grained and the protocols are lightweight.

Strengths

- Independent components
- Easier understanding
- Better scalability
- Flexibility in choosing technology
- Higher level of agility

Weaknesses

- Extra complexity
- System distribution
- Cross cutting concerns
- Testing

When to choose Microservices architecture?

- Expertise availability (DevOps)
- A complex and scalable app
- Enough engineering skills

Monolithic system

In monolithic systems, functionally distinguishable aspects (for example data input and output, data processing, error handling, and the user interface) are all interwoven, rather than containing architecturally separate components.

Strengths

- Less cross cutting concerns
- Easier debugging and testing
- Simpler to deploy
- Simpler to develop

Weaknesses

- Gets complicated over time
- Making changes are hard (tightly coupled)
- Hard/expensive to scale
- New technology barriers

When to choose monolithic?

- Smaller team
- Simpler application
- Quicker launching
- No micro services expertise

Query Optimization Tips

SQL

- Define business requirements and select required fields instead of *select **

- Select more fields to avoid selecting *distinct* which is expensive
- Use inner joins instead of outer joins if possible
- *Where* is more efficient than *Having*
- Use * at the end of the phrase only
- Run analytical queries during off-peak hours
- Order select fields based on **granularity**
- **Index** tables based on int IDs

NoSQL

- Query rewrites based on heuristics (remove unnecessary stuff)
- Index selection (avoid searching whole tables)
- Join reordering
- Join types
- Add index on collections (eg. ascending vs descending)

Good Habits

- Log everything in sys logs
- Always have unit tests
- Validate data before inserting to DB
- Automate everything you do again and again
- Don’t delete code, use git
- Instead of deleting in DB, hide from user
- Using too many globals is usually not a good idea
- Skim the whole manual instead of the parts you need

- WTFM: Write The Fucking Manual !
- If you can’t verify if something is correctly working, it’s probably not working

Interview Coding Canvas

Separate the constraints, limitations, ideas, code, and test cases. For idea generation, try to:

- Simplify the task
- Try examples to find patterns
- Think of suitable data structures