System Design

## Introduction

This document covers

Introduction

## Cheat Sheets

Measuring Time

|  |  |  |
| --- | --- | --- |
| ➊ Millisecond | One thousandth of a second |  |
| ➋ Microsecond | One millionth of a second |  |
| ➌ Nanosecond | On billionth of a second |  |

Measuring Space

|  |  |  |
| --- | --- | --- |
| ➊ 1K | 1024 Bytes |  |
| ➋ 1MB | 1,048,576 |  |
| ➌ 1GB | 1,077,741,824 |  |
| ➍ 1TB | 1,099,511,627,776 |  |
|  |  |  |

## Systems Design Techniques

### Partitioning

Sharding splits data across multiple data stores in such a way that we can work out which information is on which host. For an in-depth discussion of partitioning see

[Azure Best Practices For Data Partitioning](https://docs.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning)

Table Benefits of Partitioning

|  |  |
| --- | --- |
| Scalability | Dividing data across multiple data stores prevents us being limited by the physical limits of a single store |
| Performance | Splitting data cross multiple data stores can lead to better performance as we need to search through a smaller amount of data on each partitioned store. |
| Flexibility | We can allocate different types of data to different types of data store. In this way the data store used is the one most appropriate for the type of data. |
| Availability |  |

There are several different ways of partitioning the data.

#### Horizontal Partitioning / Sharding

Each partition is separate data store. All data stores have the same schema and holds a subset of the data. The following sections describe some strategies for allocating subsets to partitions

##### Range Strategy

#### Vertical

Each partition holds a subset of the fields. Fields are divided according to how they are used e.g. frequently accessed fields might go into one partition.

#### Functional

The partitions are determined by the bounded contexts of the architectural solution. We might put orders in one partition and product definitions in another partition.

#### Key/Hash Based

Given N servers put the data on mod(key,n). As we add servers we need to repartition all the data which is expensive.

#### Directory based

Use a lookup table to prevent repartitioning as we add servers. The drawback of this approach is that the lookup can become a single source of failure and the extra level of performance can impact performance.

## Immutability

Immutability is a design property that has two main benefits. Firstly, it makes code easier to reason about. If I pass an immutable object as the parameter to a method, I don’t have to look inside that method to see what it does to my object. I know it can’t do anything. Secondly, immutable objects do not require synchronization to work correctly in the presence of multiple threads of execution. Before we look at the new Immutable types that .NET supports, we should look at some different kinds of immutability.

Table Kinds of Mutability and Immutability

|  |  |
| --- | --- |
| Mutable | A type whose internal state can be modified in place. |
| Immutable | A type who internal state cannot be modified in place. Pseudo mutating methods create new objects that share much of the same memory as the original object but with some new memory that describes the change |
| Freezable | A type that can be mutated until such time as it is frozen after which it cannot be changed |
| Read only | A type through which references cannot mutate the underlying data. Typically, the underlying type can be mutated via other references |

Another benefit of immutable collections is that we get a record of state changes. When we change a mutable collection, the original state is lost.

Immutable collections are also useful when we have snapshot semantics. If we want some threads to see all or none of a batch of modifications. If we were to use a concurrent collection or basic collection, we would need to use a lock and copy the whole collection

## Technique

1. Determine the use cases and requirements

At this stage we want to establish the requirements and use cases of the system we are building. We need to establish the functional and non-functional requirements of the system. There is not one correct solution to these kinds of problem so it is very important to ask enough questions at the requirements stage to scope the solution

1. List assumptions

Agree with your interviewer reasonable assumptions such as number of requests per second.

1. Capacity and Constraints

It is worth considering capacity and constraints. At this state we can interested in several measures that will impact our design.

* Reads per second
* Writes per second
* Bandwidth (Reads/Writes per second in bytes)
* Storage Estimates
* Memory/Cache estimates

The results of this section are important later on when we need to consider scalability, partitioning, caching and load balancing

1. Define the System level API

Defining the API to the system will give us the chance to clarify with our interviewer that we are on the correct lines.

1. Design the database

At this stage it is worth jotting down some database tables that will hold our data. It is worth considering the type of database technology. Consider when we might use a relational database and why we might use a NoSQL database. By identifying the different data entities and their relationships.

1. Diagram the components (High level design)

Show boxes etc for major components such as servers and data stores. Once this is done show how data and requests flow through the components to satisfy the requests from part 2. It is acceptable at this stage to not worry too much about details such as scalability.

1. Design the components (detailed design)

Often, we will not deep dive into every aspect. Ideally the interviewer would help decide which aspects are most deserving of further consideration Topics that might be relevant include

* Application Layer
* Database storage
* Data Partitioning and replication
* Caching
* Load Balancing
* Security