1. Comparing programming languages

Dynamically typed: Types are determined at runtime

Statically typed: Types are determined when compile

Weak typing: Types can be implicitly changed (For example: In js, “1125” + 5 = “11255”, “1150” – 10 = 1140

Strong typing: Types can not be implicitly changed (For example, in Python: “1125” + 5 => exception)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Node.js / Javascript | Python | Scala |
| Typing System | + Dynamically typed  + Weak typing | + Dynamically typed  + Strong typing | + Statically typed  + Strong typing |
| Memory management | + Automatically reorganize memory  + Use mark-and-sweep algorithm and generational algorithm  + Will pause program to perform garbage collection | + Does not automatically reorganize memory  + Use reference counting with cycle detection  + Will pause program to perform garbage collection | + Automatically reorganize memory  + Use mark-and-sweep algorithm and generational algorithm  + Support concurrent garbage collection. |
| Concurrency support | + Single-thread model, NodeJS will only execute 1 task at a time and send callbacks to the underlying operating system to handle  + Disadvantage: Does not take all advantages of multicore processors. | + Use threads to handle concurrency. Multiple async tasks can be sent to a thread and then combined the result at the end.  + Disadvantage: Managing thread is pretty difficult and can lead to race condition.  + Python uses GIL which ensures that only ONE thread can execute Python code at a moment | + Use Actors to handle concurrency, multiple actors can be managed in a thread. In a thread, an actor will process messages sequentially, if there is a blocking task like IO, it will send the task to the underlying system to handle  + Disadvantage: Introducing overhead like deserialize / serialize the message, life cycle management. |
| Performance | + Moderate: Low overhead as it does not care about things like thread creation or context switching …. | + Slowest: This is due to the GIL, GIL ensures that only ONE thread can execute Python code at a moment. This introduces overhead like context switching, managing resources, race conditions  + Python will stop the program to collect garbage and detect cycle. | + Fastest: Compiled language, concurrent garbage collection, multiple threads can be executed at the same time. |

1. Compiled Language vs Interpreted Language

|  |  |  |
| --- | --- | --- |
|  | Compiled Language | Interpreted Language |
| Execution | * Optimize and convert into machine code before producing an executable that the computer can run | * Code will be converted into machine code line by line in runtime. |
| Performance | * Faster due to the optimization when compiling code and does not have overhead for deciding the type of the variable. | * Slower due to the fact that the language needs to convert each line into machine code and overheads like deciding the type of the variables. |
| Dependency | * Platform specific executables, so a file can run in one computer might not be able to run in another computer | * Require a runtime environment (Python interpreter or JS engine) |

1. SQL vs NoSQL (ACID)

|  |  |  |
| --- | --- | --- |
|  | SQL | NoSQL |
| Atomic | * Guarantee atomicity (basically means that in a SQL session, either all transactions are successful or none are successful) * Advantages: Guarantee consistency * Disadvantages:   Might not be the best for performance since it will introduce overheads like reversing the changes | * Does not guarantee atomicity, so there will be situations like some transactions might be successful but the other might not. |
| Consistency | * Guarantee consistency * Advantages: The data that you query will always be latest data. This is really good for transactional queries. * Disadvantages: Difficult for scaling, might not have partition tolerance since it will have to sync the data to all nodes to ensure consistency. | * Eventual consistency so the data you query might not be the latest data. * Advantages: Good for performance and user experience. * Disadvantages: Might not be suitable for tasks that require strong consistency like money transactions related tasks. |
| Isolation | * Guarantee isolation, it means that if a process wants to perform a query on a row, it must first acquire a lock/mutex on that row. * Advantages: Make sure that the data is always correct and consistent. * Disadvantages: Might not be good for performance since each row can only be updated once at a time and because of overheads like managing locks | * Does not guarantee isolation, for instance, MongoDB allows you to config running multiple queries at the same time. * Advantages: Great for performance * Disadvantages: Might lead to incorrect data. |
| Durability | * The data will be written to the disk before marking the query as successful. * Advantages: Strong reliability * Disadvantages: Not that great for performance | * Durability might vary depends on the DB and configuration. For example , CassandraDB allows developers to config the durability level like you can config it to mark a query as successful after it is being written to only ONE node. * Advantages: Might be really good for performance * Disadvantages: Might not be suitable for applications that require high durability |

1. CAP Theorem

Consistency: Users always get the latest data

Availability: Every request will be answered with a response (it can be failed or successful)

Partition tolerance: The system still operates even if some nodes in the system are down.

AP: CassandraDB

CP: MongoDB (might reject certain write operations and might not response to all queries)

CA: PostgreSQL (if the communication between the master node and the slave nodes fail, the system will not be able to operate normally)

1. SOLID

Single Responsibility: A class should have only one purpose

Open-Closed Principle: Open for extension but closed for modification

Liskov Substitution Principle: If class B is a child of class A, then every method that uses class A should also be able to use class B normally.

Interface Segregation Principle: This principle encourages developers to split a large / complicated interface into smaller ones so if you want to inherit a new class from that interface you don’t have to implement methods that you don’t need.

Dependency Inversion Principle: High level code should not depend on low level code

1. Work experience