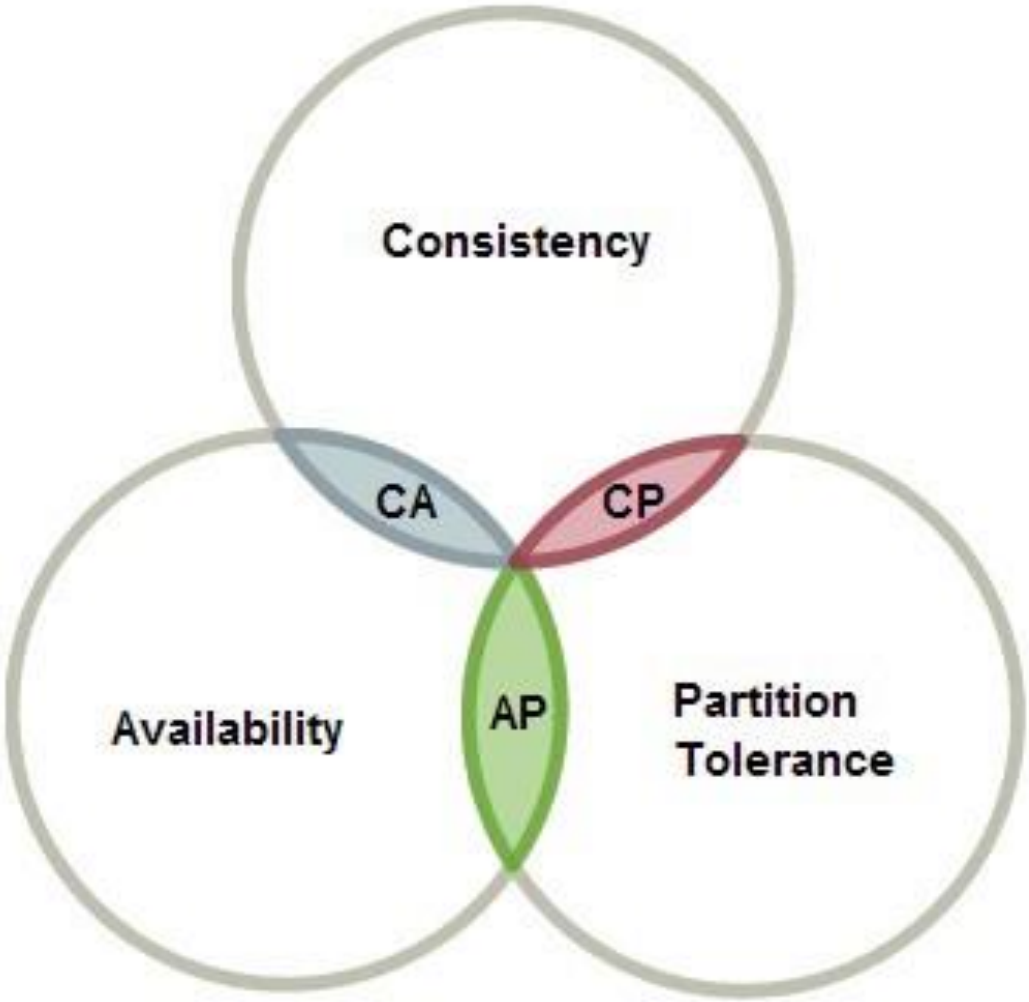
CAP Theorem and Distributed Database Management Systems

(https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e)

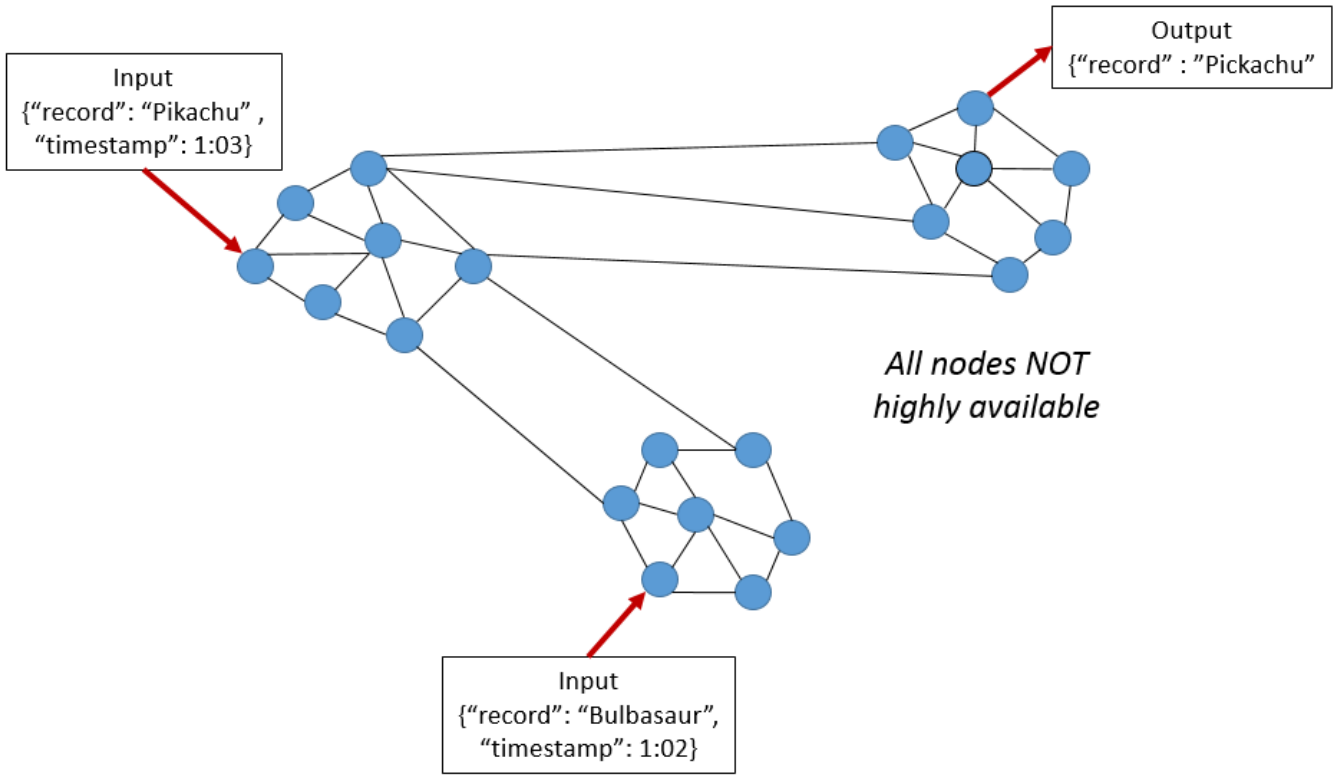
Introduction

- In the past, when we wanted to store more data or increase our processing power, the common option was to scale vertically (get more powerful machines) or further optimize the existing code base. However, with the advances in parallel processing and distributed systems, it is more common to expand horizontally, or have more machines to do the same task in parallel. We can already see a bunch of data manipulation tools in the Apache project like Spark, Hadoop, Kafka, Zookeeper and Storm. However, in order to effectively pick the tool of choice, a basic idea of CAP Theorem is necessary. CAP Theorem is a concept that a distributed database system can only have 2 of the 3: **Consistency, Availability and Partition Tolerance**.

CAP Representation



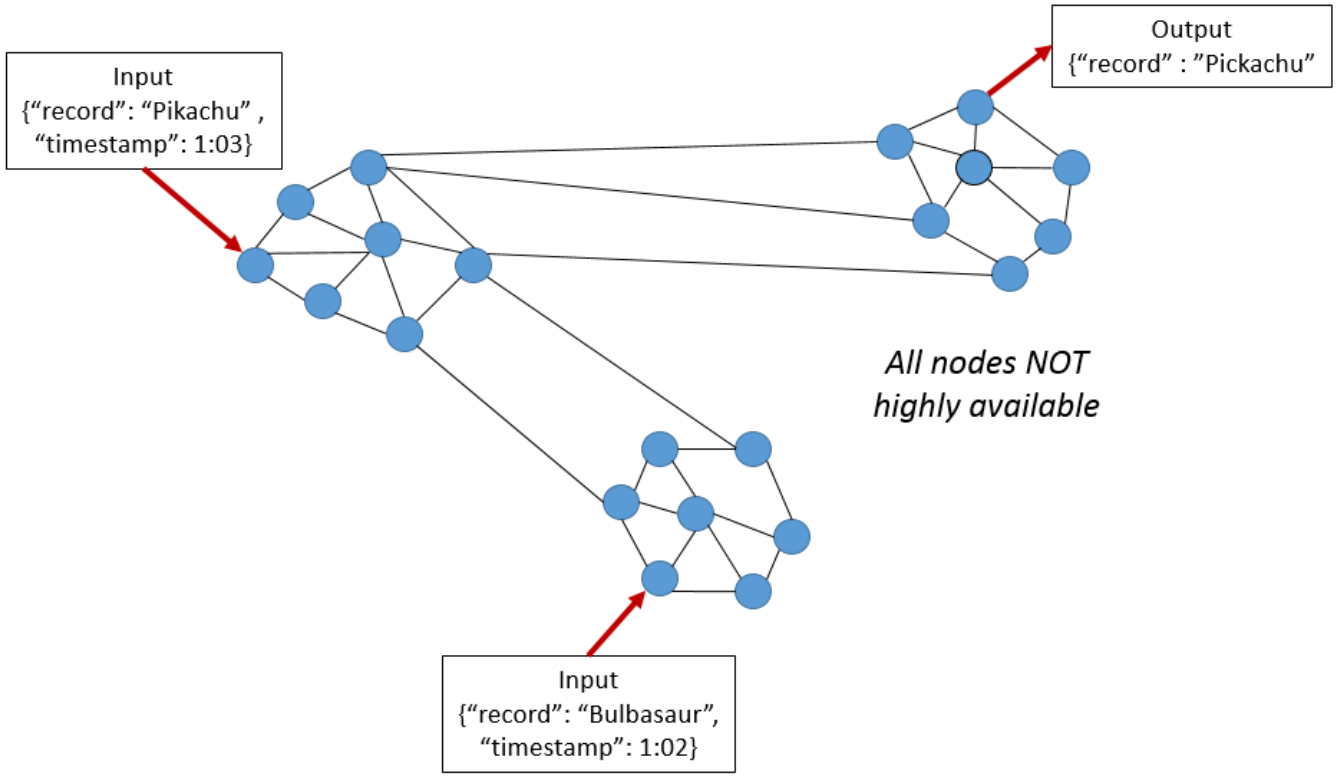
Partition Tolerance



Partition Tolerance

- This condition states that the system continues to run, despite the number of messages being delayed by the network between nodes. A system that is partition-tolerant can sustain any amount of network failure that doesn’t result in a failure of the entire network. Data records are sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages. When dealing with modern distributed systems, Partition Tolerance is not an option. It’s a necessity. **Hence, we have to trade between Consistency and Availability.**

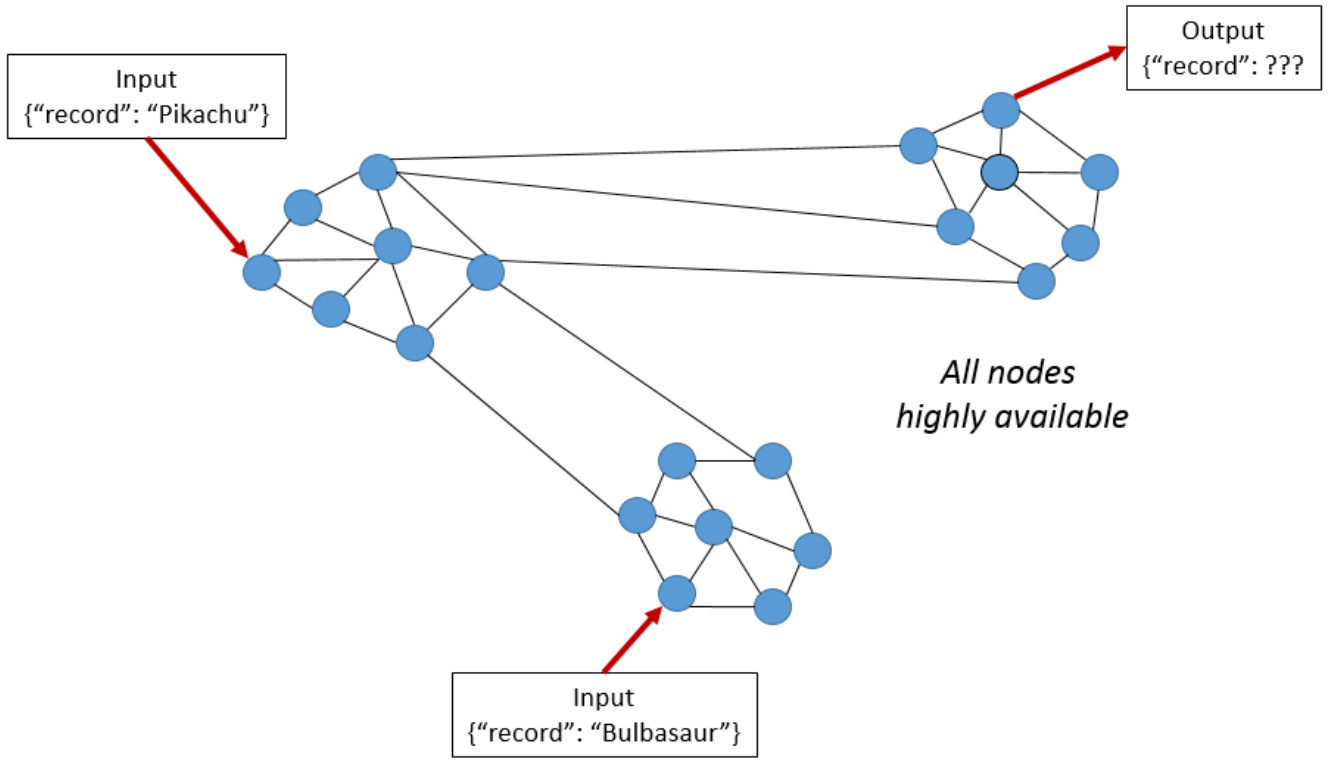
High Consistency



High Consistency

- This condition states that all nodes see the same data at the same time. Simply put, performing a *read* operation will return the value of the most recent *write*operation causing all nodes to return the same data. A system has consistency if a transaction starts with the system in a consistent state, and ends with the system in a consistent state. In this model, a system can (and does) shift into an inconsistent state during a transaction, but the entire transaction gets rolled back if there is an error during any stage in the process. In the image, we have 2 different records (“Bulbasaur” and “Pikachu”) at different timestamps. The output on the third partition is “Pikachu”, the latest input. However, the nodes will need time to update and will not be Available on the network as often.

High Availability



High Availability

- This condition states that every request gets a response on success/failure. Achieving availability in a distributed system requires that the system remains operational 100% of the time. Every client gets a response, regardless of the state of any individual node in the system. This metric is trivial to measure: either you can submit read/write commands, or you cannot. Hence, the databases are time independent as the nodes need to be available online at all times. This means that, unlike the previous example, we do not know if “Pikachu” or “Bulbasaur” was added first. The output could be either one. Hence why, high availability isn’t feasible when analyzing streaming data at high frequency.

Conclusion

- Distributed systems allow us to achieve a level of computing power and availability that were simply not available in the past. Our systems have higher performance, lower latency, and near 100% up-time in data centers that span the entire globe. Best of all, the systems of today are run on commodity hardware that is easily obtainable and configurable at affordable costs. However, there is a price. Distributed systems are more complex than their single-network counterparts. Understanding the complexity incurred in distributed systems, making the appropriate trade-offs for the task at hand (CAP), and selecting the right tool for the job is necessary with horizontal scaling.