**OVERVIEW OF THE ARCHITECTURE STEPS:**

1. The end users access the site ( https://loginRadius.com/track ) through multiple types of devices.

2. The data from the end users reach the edge machines ( acting as the **load balancer** ) which evenly distributes the traffic over the distributed servers , that serve the requests to the end users as well as collect the click streams originating from them . The balancing can be achieved by using **consistent hashing**. To cater to requests originating from **varied geographical areas**, the **CDN** technique can be used to provide faster access to users originating from a specific geographical region.

3. The events are now validated and pushed by the servers at the data centres to the **distributed queues**. For example **Rabbit MQ** can be used here , which provides both the load distribution techniques, consistency of data and failover mechanisms.

4. The ETL processor next picks up the events from the queues and does additional steps of extraction , transformation and processing to make the data ready for querying purpose. For scaling purpose , faster and failure tolerant mechanism , the **computing nodes** from a **cloud** provider can be used which implements the ETL processor, and scales up or down depending upon the load of events at that point of time.

5. The processed data is now pushed and stored onto the database. The database can be designed to be fault tolerant and latency reduced by using the concepts of **sharding** to be able to keep all the related data in one node, thereby **increasing data locality**, which ultimately reduce the data query and analytics time. **No-Sql** database can be used in this case.Apart from replication among many nodes, the data is backed up onto the cloud system, for failure and disaster recovery.

6. From the consumer side, there may be multiple sources ( apps, analytics UI) which will be feeding on the data in the database , and take up the analyzed data based on their queries. The incoming queries from those sources, are load balanced and fed into the query execution engine. The **query execution engine** itself is a distributed data processing framework that employs multi nodes to process the data and analyze them. **Spark** can be a choice , over here, which along with its in-memory processing and streaming analytics, will also ensure fault tolerant mechanisms to ensure the processing of the events.

**Questions & Answers**

**How to design the data ingestion system to ensure the least latency b/w event happening time and event processing time?**

The above architecture used , reduces the time, in effect that right from the time of accessing the content by the end users till the data analytics being generated by the query execution engine, the entire approach is **multi threaded/ parallel-pipelined**, For better interaction with the apis , the **geographic specific CDNs** are used. Then the events are captured and processed using **fault** **tolerant, load balanced queues**, followed by storing the data in the database using the **sharding** techniques, which increases the data locality , so the querying time gets reduced, and the result gets generated much quicker, by employing the **streaming analytics** of the query execution engine.

**What are the database/queue choices?**

The above architecture uses the **RabbitMQ** for storing the raw events coming in from the end users. The queue maintains by itself all the data replication , load balancing and failover mechanism, which makes sure that all the incoming events are processed and persisted in the database for access by the query execution engine and generate analytics.

The database selected can be that of **NoSql** type( ex:**Cassandra**) . The data will be stored and distributed among the nodes using the **hash function ( consistent hashing),** this improves the data locality and reduces data shuffle between nodes during querying. So here the hash can be on the **user-id**. The **problem** in our use-case of the **data being skewed** , can be resolved by either using the **other meta data( like date of records)** along with user id for hash which distributes the data, or a **multi-level sharding** technique can be used. **Replication factor** of `~3 can be kept to make the database data **fault tolerant.**  The database should employ the techniques of compaction to effeciently use space and maintain the merged version of the “Sorted String Tables” ( like in Cassandra ) and employ the technique of **Quorum** to enable proper **consistent retrieval** of data , even in case of failure of some of the nodes.

**How will you handle errors at event processing? Ex: If an event is not processed due to some database issue. What are the ways to ensure it is processed later?**

Issues at the upfront processing by the ETL processor ( like any node failure ) will be managed by the **RabbitMQ**, which basically does all the important tasks of ( Load Balancing, Persisting the mapping of the event assigned to a node and using heartbeat mechanism to check if the assigned node is completing the assigned task ( dead or alive) ). So if there is any loss of a single node with a set of events being processed, then that will be taken care of by the task queue and reassigned to another node for processing.

Issues during the querying part( generating the analytics ) on the data, like in our case we use the in-memory distributed data processing framework ( SPARK) will be taken care of by the framework through it’s **driver executor relation**, in which the driver maintains a list of all the executors working on a set of data , and if there is any failure at the executor level, it will be reassigned to another executor by the driver node.

Also at the data persistence level, we maintain **replicas** of the same set of data, and there is a back up at the cloud storage which induces another level of fault tolerance, and ensures the processing of events.

**Can the system handle 1000+ customers? If not, how can we achieve such scale?**

The architecture is enabled to handle large scale of data. The front end servers interacting with client are distributed geographically, and we have a load balancer in place handling the requests coming in. With proper choice of hashing functions ( like **consistent hashing** techniques) the load will be distributed among the nodes, and during scaling up or down the distribution of load on the nodes will be optimized by ensuring that the **minimum cache invalidation** is done. The central ETL processor is also designed using cloud computing resources, that scales up or down based on data load.

**How will the system scale if the customers are from different geographical locations?**

For handling the geographically distributed events , we have our edge systems distributed geographically, also the application/api which the customers interact with are served by geographic based CDN, that reduces latency, and the events are also captured by our queue, that are geographically distributed, which will help us in reducing latency to some extent in handling geographically dispersed data.