Solution:

Step 1: Gather the Requirements

Functional Requirements

a. System must generate Unique IDs on demand
b. System must generate non-colliding ID (unique local and globally)
c. IDs generated must be sortable
d. It must be possible to ascertain the order of generation from the list of IDs

Non-Functional Requirements

a. IDs must be generated with low latency (10s of milliseconds)
b. IDs must be generated for high throughput

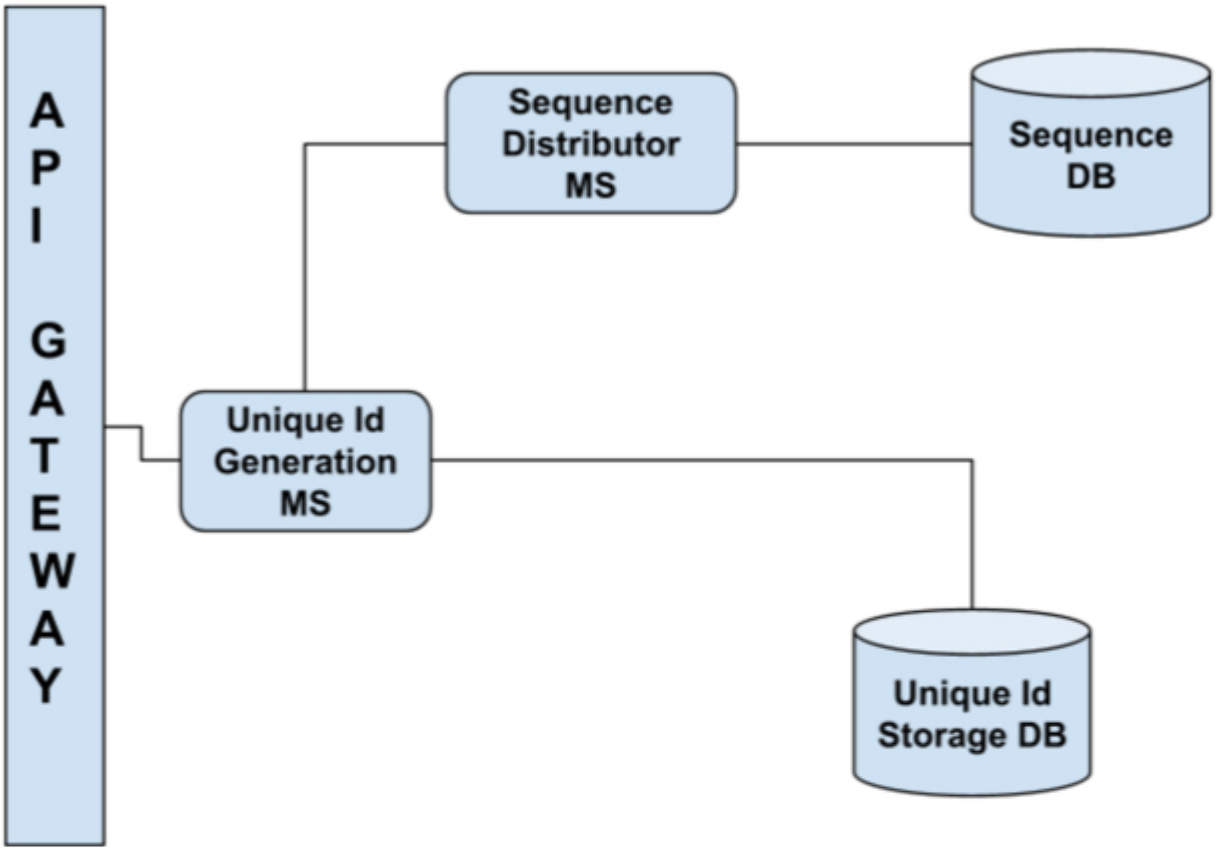
Back of the Envelope Calculation

a. Unique ID generated 10 million per day
b. Each Unique ID is 8-bytes (64-bit) in length, 8 bytes * 10 million / day * 24 * 60 * 60 ~= 800 qps. (Approximated 24 * 60 * 60 ~= 100000 seconds)
c. Storage requirement for 10 years is 8 * 10 mil bytes * 30 * 365 * 10 ~= 9TB

Step 2: Define Microservices

Microservice	Type of Technology
UniqueId Generation Microservice	Offline: Compute Intensive workload, Online: Simple Data Storage on Disk
Sequence Distributor Microservice	K-V Workload, Simple Data Storage on Disk and Hash Table K-V in memory

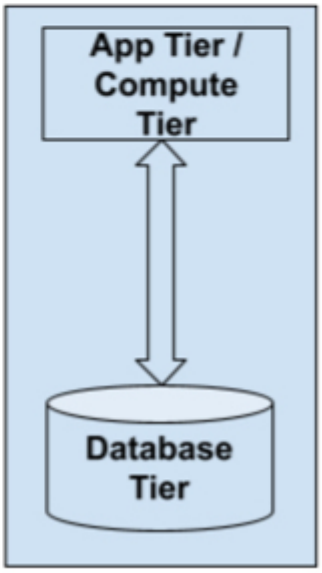
Step 3: Draw the logical Architecture



Step 4: Deep dive into the microservice

UniqueId Generation Microservice

The UniqueId Generation microservice is the critical service that generates an unique id on demand. This description applies for generation of the unique id that is local to a group of machines part of the same domain.



Data Design

a. generateUniqueId()
   This method returns a 64-bit unique id that is sortable by time.
   The 64-bits is split as follows:

Bit Length	Source
41-bits	Time in milliseconds.
10-bits	Unique id (combination of worker_pool_id + machine_id)
12-bits	An auto-incrementing sequence of 4096
1-bit	reserved

The 41-bits can be started with an epoch time / seed time like January 1, 2011, 00:01 am. This gives a custom epoch of almost 41 years. Note, our requirement is to store 10 years of data.

The 10-bits represents the logical id created as per the below logic:
worker_pool_id consisting of 6 bits, resulting in 64 workers
machine_id consisting of 4 bits, resulting 16 machines

The 12-bits means we can generate 4096 IDs per millisecond. This is well within the QPS calculated in the Back of the Envelope calculation section

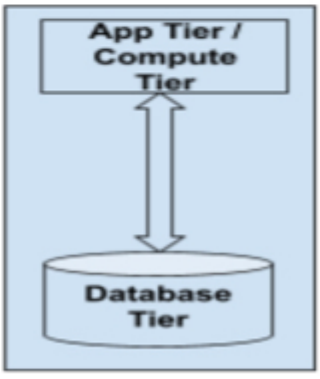
```
import time

# January 1 2011, 12:00:01 am, epoch seed
epoch_seed = 1293868801000L
worker_pool_bits = 6L
machine_id_bits  = 4L
autoincrement_id_bits = 12L
autoincrement_id_mask = -1L ^ (-1L << autoincrement_id_bits)
epoch_seed_shift = autoincrement_id_bits + machine_id_bits +
worker_pool_bits
worker_id_shift = autoincrement_id_bits
machine_id_shift = autoincrement_id_bits + worker_id_shift


def generateUniqueId(worker_pool_id, machine_id):
    last_timestamp = -1
    autoincrement_id = 0

    while True:
       # Get the current time
       current_time = long(time.time() * 1000)
```

```
if last_timestamp > current_time:
    # we have hit a clock that is backward, this catches
    # out of sync clocks in a distributed environment too
    # Just sleep for the difference now
    sleep(last_timestamp - current_time)
    continue
 if last_timestamp == current_time:
    autoincrement_id = (autoincrement_id + 1) & \
                        autoincrement_id_mask
    # check if we are overrunning the sequence now
    if autoincrement_id == 0:
       autoincrement_id = -1 & autoincrement_id_mask
       # Sleep for a second
       sleep(1)
       continue
else:
     # We have turned around, so reset the auto_increment
     autoincrement_id = 0

last_timestamp = current_time

unique_id = ((epoch_seed) << (epoch_seed_shift) |
              machine_id << machine_id_shift |
              worker_pool_id << worker_id_shift |
              autoincrement_id)

return unique_id
```

Sequence Distributor Microservice

This microservice will be used as part of a scalability solution that requires unique ids that need to be unique across geographically separated instances.

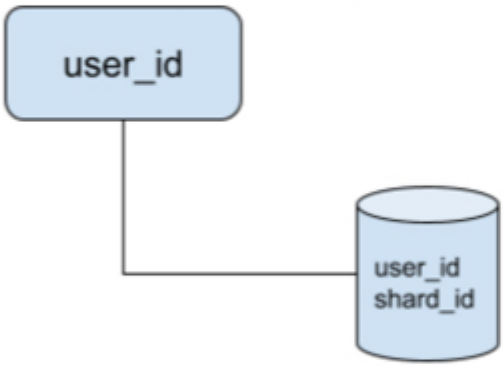
When the sequence distributor microservice is involved, the table design is as follows:



Bit Length	Source
41-bits	Time in milliseconds.
10-bits	Unique id (shard_id)
12-bits	An auto-incrementing block of 4096 range of numbers per instance
1-bit	reserved

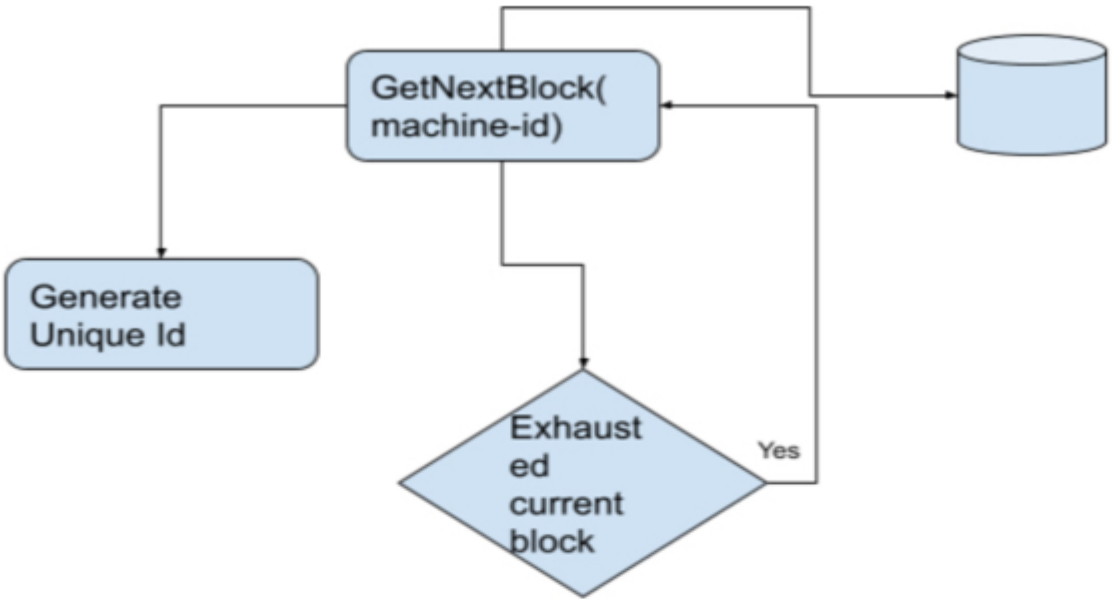
Data Design

a. getShardId(user_id)
This method gets the shard_id given the user_id.



b. getBlockRangeForMachine(shard_id)
Given the shard_id, this method gets the block-range that needs to be used as part of the unique-id generation. This will be particularly useful and does not necessitate the unique-id microservice to query for the next number to be used in the generation. This also helps to avoid collisions in generation. When the assigned range of 4096 number instances are exhausted, the service requests the next block of numbers. The service instance can handle upto 4096 instances per shardid per millisecond

UserId	Shard_id (10 bits)	Block-Range (12 bits)	Unique-Id Generated (64 bits)



Step 5: Identify the need for scale

Service	Scalability Need	Scalability Dimension

UniqueId Microservice	• Compute Intensive
	• SIngle API to be replicated
	  ○ API Parallelization
	  ○ Shard Replication
	• Database Storage
	  ○ Hotspots avoidance
	  ○ Geographic Distribution
	  ○ Throughput
		Horizontal Scaling for Sharded API Parallelization
		
		Horizontal Scaling of database

Sequence Distributor Microservice	• Compute Intensive
	• SIngle API to be replicated
	  ○ API Parallelization
	• Database Storage
	  ○ Hotspots avoidance
	  ○ Geographic Distribution
		Horizontal Scaling for Sharded API Parallelization
		
		Horizontal Scaling of database

Scalability Feature	How achieved	Component
Throughput	Replication 3X Sharding	a. Database (Horizontal Sharding based on Key, Consistent Hashing for servers) b. API Parallelization
Hotspot Avoidance	Load-Balanced Servers Sharding	a. Database
Availability	Replication Geographical Distribution	a. Database
Consistency	Eventual Consistency	a. Database (CP system)
Latency	Caching	a. Sequence Distributor Service b. UniqueId Generation service

Step 6: Propose the distributed architecture

Distributed Architecture for Cache/ Compute Layer