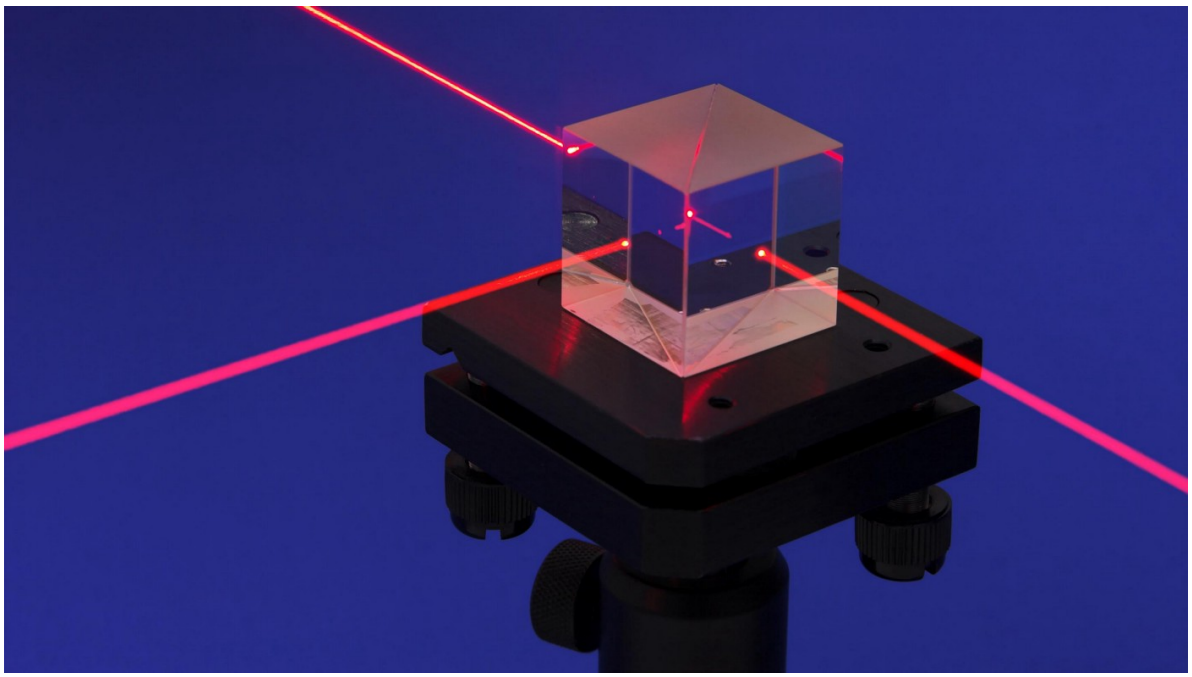
mecha-mind.medium.com

System Design — Billing System - Abhijit Mondal - Medium

Abhijit Mondal

12-16 minutes



source: quanta magazine

> Design a system for a SaaS based Billing application such as AWS

Functional Requirements

- Users must be able to see what are the costs per usage for each service.

- The billing system should be able to show the breakdown of the bill amount along with usage for each service and for each user.
- By default the bill is computed for a period of 30 days for each user but users can query their usage for a range of start to end dates.
- Users are asked to pay the bill for a 30 day period and the system should be able to track whether an user has paid the bill or not. It is not required to implement the payment gateway here just the accounting.

Non Functional Requirements

- **Availability** — System should be highly available so that the correct usage and transactions are tracked.
- **Durability** — No usage data should be lost as this could lead to loss in revenue for the service.
- **Correctness and Reliability**— Bill amount should be as accurate as possible. Although we can use some approximations along the way but total bill should not be more than the exact bill amount. At-most once semantics for all billing data.
- **Consistency** — System should be able to track usage across multiple regions as well as devices.
- **Cost** — The cost of maintaining the billing system should be lower than the individual service costs.

Traffic and Throughput Requirements

- Number of total customers = 1 billion

- Number of DAU = 500 million
- Number of different services that has billing associated = 50
- Expected QPS across multiple services = 1 million

The “On My Computer” Approach

Lets assume that each of those services are being implemented as micro-services and we just need to worry about how to process the usage and then compute the billing.

How to compute the usage ?

Different services might require different usage and billing criteria.

- For API related services, it would be most likely be some **$x per invocation**.
- For CPU intensive services, it would be something like **$x per hour**.
- For storage/DB services, it would be something like **$x for each GB/s throughput of read/write queries**.

To make this data available to users, we can create a database table on our local machine:

usage_cost_per_unit
service_id, service_name, region, unit, per_unit_cost_in_usd

For **API services**, whenever a client makes an API call, on receiving the request, the system puts a message into Kafka message queue identifying that a call has been made.

topic: billing_<service_name>**message_format**
account_id, service_id, timestamp

For **CPU based services**, when the process ends (returning from function call or client kills the process), the system puts a message into Kafka with the duration of the process run-time.

topic: billing_<service_name>**message_format**
account_id, service_id, start_timestamp, end_timestamp, duration

For **Storage/DB services**, whenever a client makes a query, the results are fetched from the DB and the size of data is computed. The system puts a message into Kafka with the data size.

topic: billing_<service_name>**message_format**
account_id, service_id, start_timestamp, end_timestamp, size_of_data

Multiple Kafka consumers will be reading from different topics related to services.

1. Fetch per unit cost for service_id from ‘usage_cost_per_unit’ table.
2. For each message with service_id, **compute the cost incurred for each usage by multiplying the per_unit_cost with usage data in that message**.
3. For e.g. for API service, we just multiply per_unit_cost by 1, for CPU services we multiply by duration (converted to hours) and for DB services we convert usage into GB/s and then multiply this quantity with per_unit_cost.
4. Insert the following message format into append only log files:

   message_format
   account_id, service_id, start_timestamp, end_timestamp, total_cost

   The log files is partitioned by:

<date>-<account_id>-<service_id>

Create a folder structure similar to the partitioning strategy above.

> In this way, we can very quickly fetch usage for a particular account for a given date and also show usage per service just by going one level deep inside account_id folder.

How to handle situations where a service usage spans across multiple days ?

Based on the start_timestamp and end_timestamp, if end_timestamp is in the next day then create one entry with (start_timestamp, 12:00 AM) and the another entry (12:01 AM, end_timestamp). ‘total_cost’ is proportionately divided between the 2 entries.

If it spans across multiple days, then create multiple entries.

How to aggregate the results at account and service levels ?

One solution is to do **aggregation in real time**.

> Create a HashMap from (date, account_id) key to the location of the folder containing all log files created by account_id on that date.

Thus in order to find the total bill for an account_id from date X to date Y:

- Lookup HashMap to fetch all folders in the date range X to Y.
- For each folder, read the log files recursively and then add up the total_cost values from each file and from each folder.

> For aggregating by service_id, recursively read the files inside each **service_id sub-folder** and sum the total_cost.

The drawbacks here are as follows:

- For a **very large range of dates**, we might need to read millions of files for which lots of **expensive disk I/O** is required. This would be very time consuming for real time workloads.
- For the same account_id, if he/she is issuing multiple queries with **overlapping date ranges** still we are scanning all files again and again.

How to optimize ?

Do pre-computations

- For each account_id, run background jobs using Airflow, Spark or similar tools to aggregate the cost for each day, then for 2 days, then for 4 days, 8 days and so on.
- Create multiple HashMaps with key as (account_id, X, X+1) for cost from day X to X+1, key as (account_id, X, X+2) for cost from day X to X+2, key as (account_id, X, X+4) for cost from day X to X+4 and so on.

  H(account_id, X, X+2^k) = Total cost of customer from day X to X+2^k
- Create another HashMap with key as account_id and value is the maximum date range for which cached cost is available.

  G(account_id) = max_range
  i.e. max value of 2^k in H(account_id, X, X+2^k)
- For a query from Y to Y+100, assuming max_range is 128, then we compute:

  C1=H(account_id, Y, Y+64)
  C2=H(account_id, Y+64, Y+64+32)

C3=H(account_id, Y+64+32, Y+64+32+4)Then return the **Total Cost = C1+C2+C3**

- If max_range is say 16, then we do not have the key (account_id, Y, Y+64) or (account_id, Y+64, Y+64+32) because 16 < 64 and 16 < 32. In such cases, we can compute:

  C1=H(account_id, Y, Y+16)
  C2=H(account_id, Y+16, Y+16+16)
  C3=H(account_id, Y+16+16, Y+16+16+16)
  C4=H(account_id, Y+16+16+16, Y+16+16+16+16)
  C5=H(account_id, Y+16+16+16+16, Y+16+16+16+16+16)
  C6=H(account_id, Y+16+16+16+16+16, Y+16+16+16+16+16+16)
  C7=H(account_id, Y+16+16+16+16+16+16, Y+16+16+16+16+16+16+4)Then return the **Total Cost = C1+C2+C3+C4+C5+C6+C7**

  Since these are all in-memory computations with HashMap, it should be very fast as compared to disk I/O.

  In order to compute the bill for the cycle Y to Y+30.

  C1=H(account_id, Y, Y+16)
  C2=H(account_id, Y+16, Y+16+8)
  C3=H(account_id, Y+16+8, Y+16+8+4)
  C4=H(account_id, Y+16+8+4, Y+16+8+4+2)Then return the **Total Cost = C1+C2+C3+C4**

  What if a particular key does not exist in memory ?

  Read the total cost from the log files as our earlier method and update the in-memory HashMap with the result.

  How to ensure at-most once writes to the log files ?

  > One possible solution is to delete the message from Kafka queue

> after it is read by one of the consumer processes. But this process requires a lock.

Without locking, it might happen that after a message has been read by one consumer, by the time the consumer deletes the message from the topic, another parallel consumer reads it and thus we have 2 consumers that have read the same transaction. On insertion into the log file, we would have counted the cost twice for the single transaction which would lead to bad customer experience.

> Either we use a single consumer process or we use locks during read, both of which will likely cause **slow processing of the messages**.

Also we cannot just use an unique id for each transaction because we are writing these to append only log files which do not guarantee uniqueness as in a database.

> Instead of using another HashMap to track which transaction id has already been added, we can use a less expensive but approximate **Bloom Filter** to track which transaction ids has already been added in the log files.

If a transaction id is added, Bloom Filter will always say Yes, but there will be some situations where even if the transaction id is not added it will still say Yes but by optimizing the size of the BF and the number of hash functions we can control that error rate.

> **False positives are less costly for us than false negatives**. In the worst case the estimated cost for a customer would be lower than actual but never higher.

How to track which users has paid their bills ?

We can use SQL database such as Postgres to track the payments

payments:
account_id, bill_period_start, bill_period_end, total_bill_amount, amount_paid

> SQL database because we need transactional properties of relational DBs for updating payments.

1. Check if user outstanding amount (total_bill_amount - amount_paid) > 0.2. If outstanding amount > 0, then proceed to bill payment for the same time period.

In order to check if the user has any outstanding bill for a month, we can just make a query:

SELECT **total_bill_amount-amount_paid as outstanding** FROM payments WHERE account_id=<account_id> WHERE bill_period_start = CURRENT_DATE - INTERVAL '30 DAYS' AND bill_period_end = CURRENT_DATE;

What are some drawbacks with the single machine system ?

Assuming that we store billing data for 5 years max, number of key-value pairs of the form **H(account_id, X, X+2^k):**

For each account_id:For k=0, we have 365*5 K-V pairs corresponding to 365*5 days
For k=1, we have 365*5-1 K-V pairs
For k=2, we have 365*5-3 K-V pairs ... and so on Till k = log2(365*5) = 11Approximately 365*5*11 K-V pairs for each account_id. Since there are 1 billion account_ids, we would have 1 billion*365*5*11 K-V pairs.Number of Bits required for storing account_id = log2(1 billion) = 30.

Number of Bits required for storing 365*5 days = 11
Number of Bits for Keys = 30+11 = 41 bits
Number of Bits for value (cost) = 64 bitsTotal size of HashMap = (64+41)*1 billion*365*5*11 bits = 240TB

240TB of in-memory data cannot be stored on a single machine.

> Also if the system restarts or shuts down, all in-memory data will be lost. Thus violating durability.
>
> Since the log files, Kafka queue, database for storing cost per unit usage are all stored in the same machine, if the machine crashes everything becomes unavailable.

Expected QPS for write queries = 1 million. i.e. 1 million messages are inserted into Kafka queue per second.

Assuming that each line of the log file is 100 B in size, total number of writes per second = 1million*100 bytes= 100 MB/s. **Typical disks can support write throughput of 80–150MB/s**.

Using Distributed Systems

> Number of Redis instances required to store 240TB of HashMap data = 240*1024/64=3840 instances.

So many Redis instances would be **very expensive** because the cost for maintaining Billing system should be lower than the cost of individual services.

Another solution is to use **NoSQL database such as Cassandra**:

usage_data:
account_id, start_date, end_date, cost

NoSQL because it is **optimized for writes** and since most of the time we would be writing new entries into the DB. Reads are relatively low since users might be checking their bills likely only after 30 days.

Total size of the log files for 5 years = 1 million*100*365*5*24*3600 bytes = 14 PB

How to partition the NoSQL DB and the logs ?

Cassandra DB:

We partition on '**account_id**', then it is easy to retrieve billing data for an individual account as it will be fetched from the same partition.

partition_id = hash(account_id) % num_partitions.

But if the number of partitions are increased, the same formula above will give incorrect partition_id for most account_ids. To overcome this problem we can use **Consistent Hashing** to re-partition the account_ids.

Logs:

<date>/<account_id>/<service_id>

The logs would be written and uploaded to **S3 buckets** according to the above scheme. This could have hot partitioning issues but it is beneficial for real time reads from logs in case we do not find the cached cost in Cassandra DB.

How to achieve high availability and fault tolerance ?

Using replication.

For both Cassandra and the Log files in S3, we keep **at-least 3 replicas**.

> For both we need **high consistency** because if a user checks his bill and if it is say $X, but when he went to pay the bill, it says $Y. This could be due to replication lag, where one of the replicas shows stale data.

We can use **QUORUM** for both these scenarios. (R+W > N)

In Cassandra, we have **all master nodes** and writes are successful only when **2 out of 3 masters are successfully updated** and for reads we read from 2 out 3 master nodes. Thus we are guaranteed that **at-least one master node has the up to date data**.

> Similarly for logs, we can also have all master nodes as writes are more as compared to reads.

The Pipeline