# Google Analytics-like Solution

**Design Document**

Contents

[Google Analytics-like Solution 1](#_Toc535262625)

[High level design 1](#_Toc535262626)

[Requirements design: 1](#_Toc535262627)

[Step-1 Whenever a new user visits a page, add a row to the user\_ping table. 1](#_Toc535262628)

[Step-2 2](#_Toc535262629)

[Step-3 2](#_Toc535262630)

[Step-4 2](#_Toc535262631)

[High level architecture 2](#_Toc535262632)

[For handling billions of write requests 3](#_Toc535262633)

[For handling large read/query requests 3](#_Toc535262634)

[For handling large read and write requests 3](#_Toc535262635)

[Provide metrics to customers with at most one hour delay 4](#_Toc535262636)

[Reprocessing: 5](#_Toc535262637)

# High level design

There are 4 important steps for an analytics system:

1. Collect user data from web traffic

2. Reconciling user information across websites

3. Importing data from other systems

4. Running queries for Tracking Customer lifecycle from signups to user conversion and their retention.

## Requirements design:

### Step-1 Whenever a new user visits a page, add a row to the user\_ping table.

created\_at | cookie| url | referrer

Send a AJAX request and set a cookie in the response header. Following information will be stored in the DB

1. The time the user visited the site

2. The URL they visited

3. The referring URL they came from-can be Null

4. Id for the user to connect the multiple visits

5. Browser and OS information and pass it to user\_ping table to separate mobile and desktop users

Step-2 **Collect the signup form submission data. Store the same user cookie in the signup table.**

created\_at|cookie|email |referring\_channel

This cookie value joins the dot between first time visitor to when they become a customer. We can join web traffic to form submission and fetch how many pages did a user view before signing up.

A given user can visit from multiple channels. It would be simpler to attribute user to first incoming channel like twitter, facebook, youtube etc. Define cookie channel view that will map all pings from originating referrer to canonical social channel.

We can create materialized view of the first ping for each cookie.

Step-3 **Import data from other systems**

If client has sales data in an external CRM, it may already include fields which are of interest for this purpose. The most important of which are whether a user has had a conversation, a product trial, or eventually became a customer. We can setup a cron job to periodically pull the CRM data and reconcile based on the signup email. We match on email because the CRM doesn’t have our portal cookie, but it does have the user’s email.

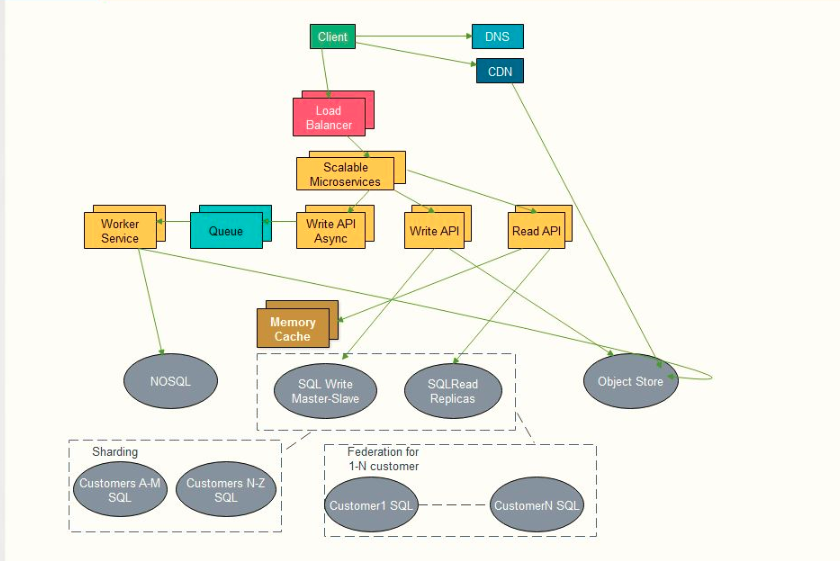
Using the above joined information fromthe signup table and sales data , we track individual bits — *demo*, *trial*, *customer* — which help us to see how far a user got into the website, even if eventually they decide to not become a customer.

created\_at| cookie| url| referrer| demo| trial| customer

Step-4 **Track Application user and Customer behavior by running queries**

When user logins, join customer account information to the cookie value by entering info in account table. By this we can track user behavior like usage, conversion from visitor to customer, type of plan(if any), upgrade, preferences, channels visited, etc.. We could generate customer reports based on these parameters. With all this data we can calculate retention of visitors, users and paid customers.

High level architecture



### For handling billions of write requests

If our DB starts to grow too large, we need to purge data in DB and store only limited time period of data in the DB. We need to store rest of the data in a data warehouse such as Redshift. A data warehouse can handle the terabytes of content easily. Additionally we can use SQL scaling pattern like Federation, Sharding, Denormalization and SQL Tuning.

### For handling large read/query requests

Read traffic for popular content can be handled by scaling the Memory cache, which is also useful for handling the unevenly distributed traffic. SQL read replicas might have trouble handling cache misses, so we'll need to employ additional scaling pattern. In SQL, scaling patterns like Federation, Sharding, Denormalization and SQL Tuning can be used.

### For handling large read and write requests

We can use CQRS framework at the service level to segregate read /write command request patterns . Additionally, it is recommended to have databases on NoSQL DB such as DynamoDB.

We can separate out App servers for independent scaling. Batch process or computations that do not need real time processing can be done asynchronously via Queues/workers. For example, while it’s great to have all of our signup and visitor data, several important events happen in separate systems. Batch processing can be used for requirements like explained in **Step3**

We can scale out and remove performance bottleneck of application using effective pattern applied at **SQL, NoSQL, Caching, Asynchronism and Microservices level**

**SQL-** Federation, Sharding, Denormalization and SQL Tuning

**NoSQL Patterns** -Key-Value store, Document store, Wide column store, Graph DB

**Caching Pattern** - Client caching, CDN caching, Web server caching, DB caching, Application Caching, Caching at DB query level and Object level.

**Mechanisms to update cache** - Cache-aside, Write-through, Write-behind and Refresh-ahead

**Asynchronism and Microservices -** Message queues, Task queues, Microservices

### 

**Alternatively, for effective DB model**

**Immutable rows:**  There shouldn't be frequently updated rows instead a new row should be appended when new state is entered. All events entered to DB should be made completely immutable.

**No DB Joins**: Rather than dealing with locks or joins across DB or tables, query data on per job basis. This allows us to massively parallelize our DBs since we'll never need to join across separate jobs

**Write heavy queries with small working set** - We can cache new items in memory and then age entries out of cache as they are delivered.

### Provide metrics to customers with at most one hour delay

Review EC2 instance ;load balancer stats; log analysis using cloudwatch,cloudtrail,loggly,splunk; External site performance using Pingdom or New Relic; Handle notifications and incidents using Pagerduty, Error reporting using Sentry. We can use in memory cache like redis cache to reduce IO call to Relation DB for the most recent data.

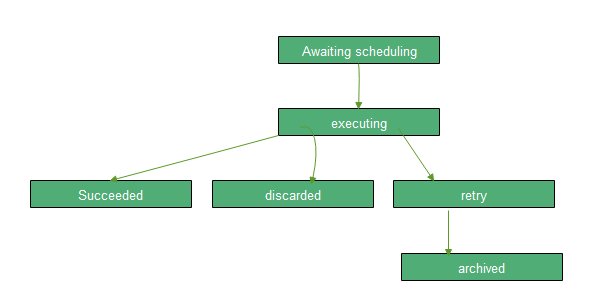
**Run with minimum downtime**

* Use [Horizontal Scaling](https://github.com/donnemartin/system-design-primer#horizontal-scaling) to handle increasing loads and to address single points of failure.Add a [Load Balancer](https://github.com/donnemartin/system-design-primer#load-balancer) such as Amazon's ELB or HAProxy
  + - ELB is highly available
    - If you are configuring your own Load Balancer, setting up multiple servers in [active-active](https://github.com/donnemartin/system-design-primer#active-active) or [active-passive](https://github.com/donnemartin/system-design-primer#active-passive) in multiple availability zones will improve availability
    - Terminate SSL on the Load Balancer to reduce computational load on backend servers and to simplify certificate administration
* Use multiple Web Servers spread out over multiple availability zones
* Use multiple MySQL instances in [Master-Slave Failover](https://github.com/donnemartin/system-design-primer#master-slave-replication) mode across multiple availability zones to improve redundancy
* Separate out the Web Servers from the [Application Servers](https://github.com/donnemartin/system-design-primer#application-layer)
  + - Scale and configure both layers independently
    - Web Servers can run as a [Reverse Proxy](https://github.com/donnemartin/system-design-primer#reverse-proxy-web-server)
    - For example, you can add Application Servers handling Read APIs while others handle Write APIs
* Move static (and some dynamic) content to a [Content Delivery Network (CDN)](https://github.com/donnemartin/system-design-primer#content-delivery-network) such as CloudFront to reduce load and latency

## Reprocessing:

**Have the ability to reprocess historical data in case of bugs in the processing logic**

Event state transition table is responsible for logging all the state transitions of a single event which pass through



An event first enters with the awaiting\_scheduling state. It is yet to be executed and delivered to the downstream endpoint.

From there, an event will begin execution, and the result will transition to one of three separate states.

If it succeeds, job will mark the event as succeeded. There’s nothing more to be done here, and we can expire it from our in-memory cache.

Similarly, if the job fails , then job will mark the event as discarded in DB. Even if we try to re-send the same event multiple times, the server will reject it. So we’ve reached another terminal state.

However, it’s possible that we may hit a failure like a timeout, network disconnect, or a 500 response code. In this case, retrying can actually bring up our delivery rate for the data we collect, so we will retry delivery.

Finally, any events which exceed their expiration time transition from awaiting\_retry to archiving. Once they are successfully stored on cache, the events are finally transitioned to a terminal archived state.