### **Problem Understanding:**

1. Scale of the system -> 1 Billion users.

2. I think following features are important:

* Authentication
* Send and receive emails
* Fetch all emails
* Filter emails by read and unread status
* Search emails by subject, sender and body
* Anti-spam and anti-virus

-> Lets not worry about authentication

3. How do users connect with email servers?

-> Traditionally, users connect with mail servers through native clients that use SMTP, POP, IMAP and vendor specific protocols. Those protocols are legacy to some extent, yet still very popular. For this interview let’s assume HTTP is used for client and server communication.

4. Can emails have attachments?

-> Yes

### **Non-functional Requirements:**

1. Reliability: should not lose email data

2. Availability: Email and user data should be automatically replicated across multiple nodes to ensure availability. Besides, the system should continue to function despite partial system failures.

3. Scalability: As the number of users grows the system should be able to handle increasing numbers of users and emails. The performance of the system should not degrade with more users or emails.

4. Flexibility and extensibility: A flexible/extensible system allows us to add new features or improve performance easily by adding new components. Traditional email protocols such as POP and IMAP have very limited functionality (more on this in high-level design). Therefore, we may need custom protocols to satisfy the flexibility and extensibility requirements.

### **Back-of-the-envelope estimation:**

1. 1 billion users

2. Assuming the average number of emails a person sends per day is 10.

QPS for sending emails = 10^9\*10 emails / 10^5 second = 10^5

3. Assuming the average number of emails a person receives in a day is 40 and the average size of email metadata is 50 KB. Metadata refers to everything related to an email, excluding attachment files.

4. Assume metadata is stored in a DB. Storage requirements for maintaining metadata for 1 year: 10^9 users\* 40 emails/day \* 365 days \* 50KB = 730 PB

5. Assume 20% of the mails contain attachments and the average attachment size is 500KB.

6. Storage for attachment in 1 year is: 10^9 users \* 40email/day \* 500KB \* 365 days \* 20% = 146\*10^4\*10^9 = 146\* 10^13 KB = 146\* 10^16 = 1460 \* 10^15 PB

**High Level Design and Get Buy-in:**

**Email Protocols**

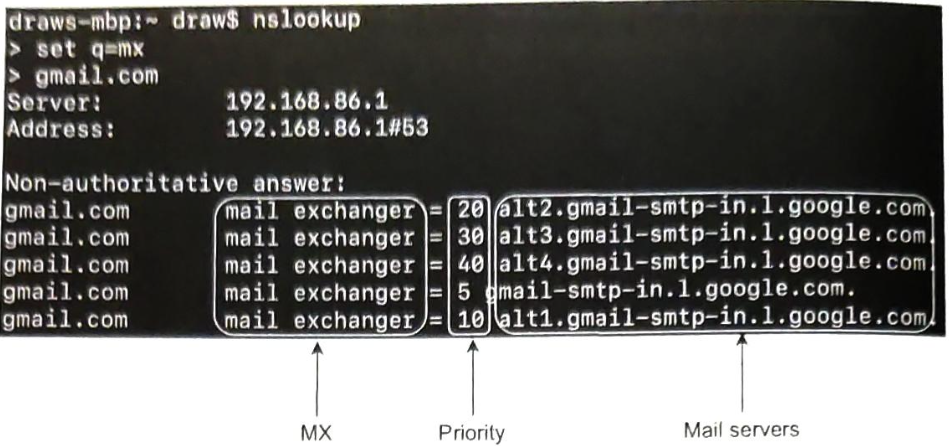
* **SMTP:** Simple Mail Transfer Protocol is the standard for sending emails from one server to another

**Protocol for retrieving mails:**

* **POP:** to receive and download emails from a remote mail server to a local client. Once emails are downloaded to your computer or phone, they are deleted from the email server, which means you can only access emails on one computer or phone. POP requires mail clients to download the entire email. This can take a long time if an email contains a large attachment.
* **IMAP:** standard mail protocol for receiving emails for a local email client. When you read an email, you are connected to an external mail server, and data is transferred to your local device. IMAP only downloads a message when you click it, and emails are not deleted from mail servers, meaning that you can access emails from multiple devices. IMAP is the most widely used protocol for individual email accounts. It works well when the connection is slow because only the email header information is downloaded until opened.

**Domain Name Service(DNS)**

A DNS server is used to lookup the email exchanger record(MX record) for the Recipient's domain. If you run DNS lookup for gmail.com from the command line, you may get MX records

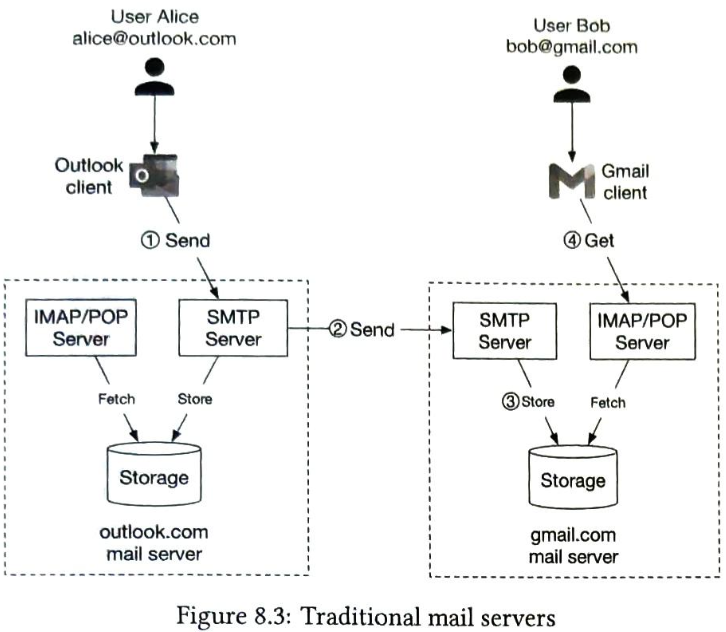


The priority number indicates preferences, where the email server with a lower priority number is more preferred. gmail-smtp-in.1.google.com is used first (priority 5). A sending mail server will attempt to connect to the mail server first. If the connection fails, the sending mail will attempt to connect to the mail server with the next lowest priority.

**Attachment:**

An email attachment is sent along with an email message, commonly with Base 64 encoding. There is usually a size limit for an email attachment. Multi-purpose Internet Mail Extension(MIME) is a specification that allows the attachment to be sent over the internet.

#### **Traditional Mail servers**



#### **Distributed Mail servers**

POST /v1/ messages:

Sends a message to recipients in the To, Cc and Bcc headers.

GET /v1/folders:  
Returns all folders of an email account.

Response:  
[{id: string,

Name: string, // default folders will be: All, Archive, Drafts, Flagged, Junk, Sent and Trash

User\_id: string // reference to account owner

}]

GET /v1/folders/{folder\_id}/messages

Returns all mails/messages under a folder. This is a simplified API. In reality, this needs to support pagination.

GET /v1/messages/{message\_id}

Response:  
{

User\_id: string,

from: <name, email>: <string, string>,

to: [<name: String, email: string>],

subject: string,

body: string,

Is\_read: boolean

}

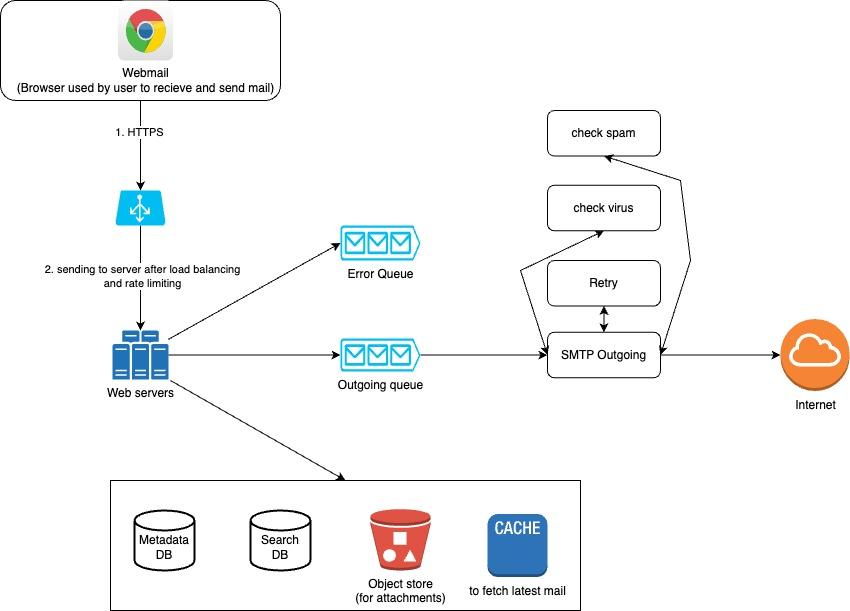
#### **Distributed Mail servers architecture**

### 

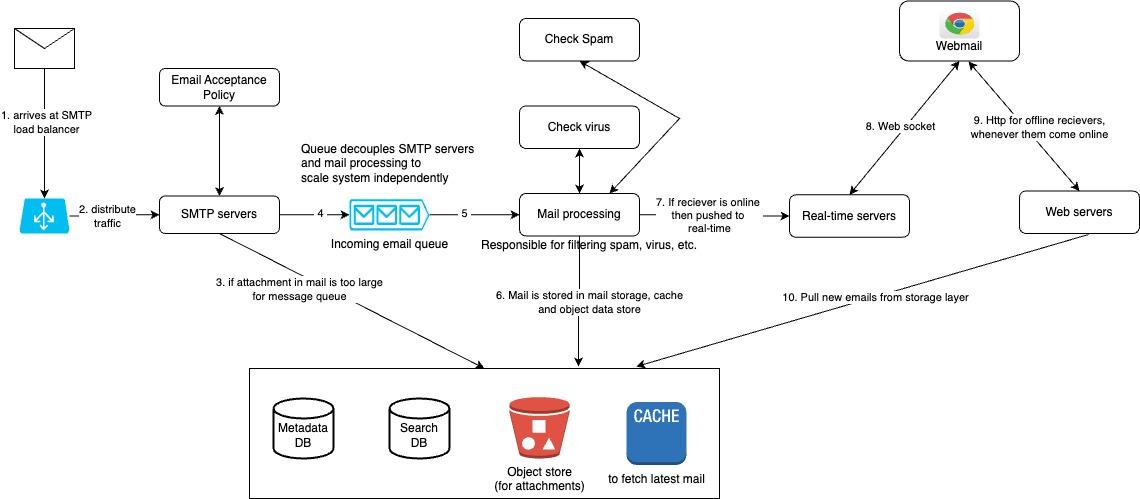
Search store: is a distributed document store. It uses a data structure called inverted index that supports very fast full text searches.  
  
Two main workflows:

1. Email sending flow
2. Email Receiving flow

#### **Email sending Flow**

****

#### **Email receiving Flow**

****

### **Design Deep Dive:**

#### **Metadata DB**

Characteristics:

* Email headers usually small and frequently accessed
* Email body sizes can range from small to big but are infrequently accessed. You normally only read an email once.
* Most mail operations such as fetching mails, marking an email as read and searching are isolated to an individual user. In other words, mails owned by a user are only accessible by that user and all the mail operations are performed by the same user.
* Data recency impacts data usage. Users usually only read the most recent emails 82% of read are for data younger than 16 days.
* Data has high reliability requirements. Data loss is unacceptable.

Considering all options in this scenario:

* Relation DB: For searching mails efficiently. We can build indexes for email header and body. With indexes, simple search queries are fast. However, relational databases are typically optimized for small chunks of data entries and are not ideal for large ones. A typical email is usually larger than a few KB and can easily be over 100KB when HTML is involved. Blob is designed to support large data entries but search queries over unstructured Blob data types are not efficient. So RDBMS is not a good fit.
* Distributed object storage. Another potential solution is to store raw emails in cloud storage such as Amazon S3, which can be a good option for backup storage, but it's hard to efficiently support features such as marking emails as read, searching emails based on keywords, threading emails, etc.
* NoSQL DB: Google Bigtable is used by Gmail, so it’s definitely a viable solution. However, BigTable is not open sourced and how email search is implemented remains a mystery. Cassandra might be a good option as well but we haven’t seen any large providers use it yet.

Large email service providers tend to build their own solutions that seem to fit needs perfectly.

So following characteristics is needed by the customized DB:

* A single column can be a single digit of MB
* Strong data consistency.
* Designed to reduce disk I/O.
* It should be highly available and fault tolerant.
* It should be easy to create incremental backups.

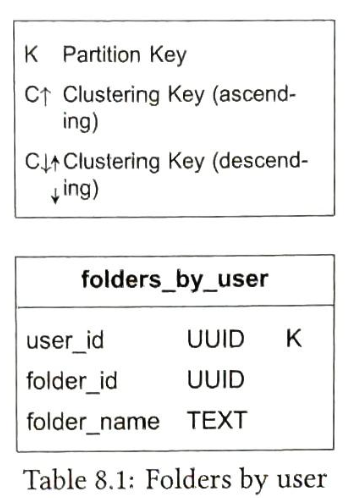
**Data Model:**

We can shard the data using user\_id. The primary key in the table would contain two components:

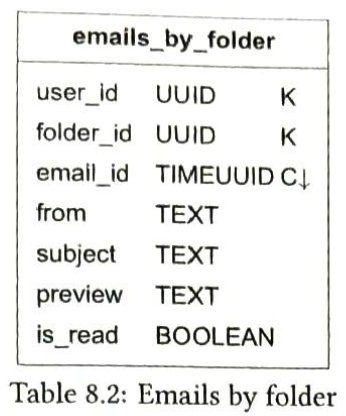
* Partition key: responsible for distributing data across nodes. For spreading data evenly.
* Clustering key: responsible for sorting data within a partition

At high level email service needs to support following queries at data layer:

**Query 1: get all folders for a user:**

User\_id is partition key, so folders owned by same user are located in one partition

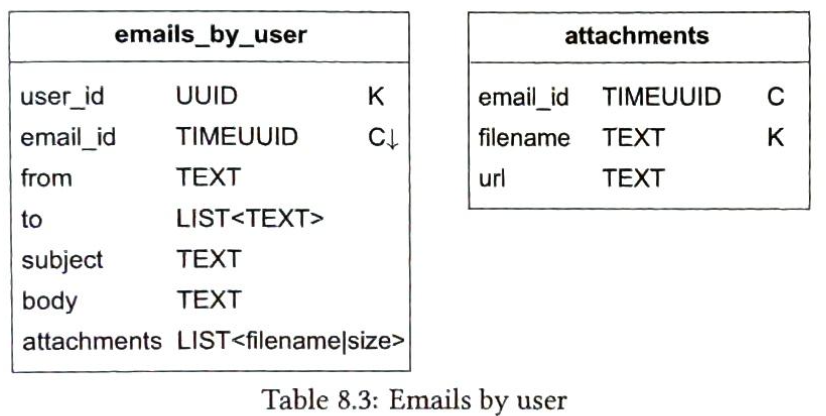
**Query 2: Display all email for a specific folder**

In order to store all emails for the same folder in one partition, composite key <user\_id, folder\_id> is used. Another column to note is email\_id. Its data type is TIMEUUID and it is a clustering key used to sort emails in chronological order.  


**Query 3: create /delete/get an email**

Select \* from emails\_by\_user WHERE user\_id=123.

An email can have multiple attachments and these can be retrieved by the combination of email\_id and filename fields.



**Query 4: fetch all read or unread emails**

If we would have used RDMS then query will be:

Select \* from emails\_by\_folder

WHERE user\_id = <user\_id> and folder\_id = <folder\_id> and is\_read= true

ORDER BY email\_id;

But our data model is designed for NoSQL. A NoSQL database normally only supports queries on partition and cluster keys. Since is\_read in the emails\_by\_folder table is neither of those, most NoSQL databases will reject this query.  
This problem is commonly solved using denormalization in NoSQL. To support the read/unread queries, we denormalize the emails\_by\_folder data into two tables:  
read\_emails: store emails that are in read status

unread\_emails: stores emails that are in unread status

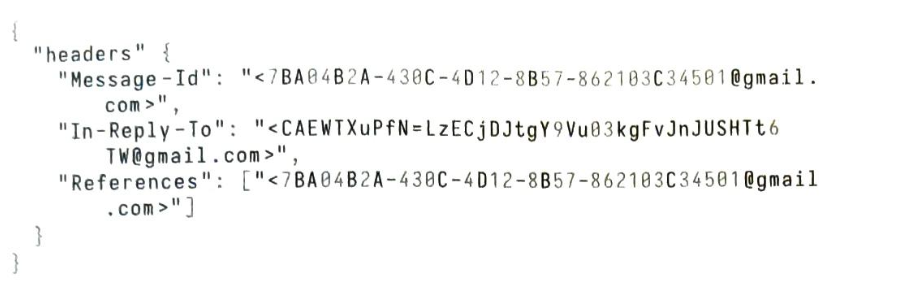
To mark an UNREAD email as READ, the email is deleted from unread\_emails and then inserted to read-emails.

Select \* from unread\_emails

Where user\_id = <user\_id> and folder\_id = <folder\_id>

Order by email\_id;

**Conversation Thread:**  
Threads are a feature which allows users to group email replies with their original message. This allows users to receive all emails associated with one conversation. Traditionally, a thread is implemented using algorithms such as JWZ algorithm.



MessageId: The value of a messageID. It is generated by a client while sending a message.

InReplyTo: The parent MessageID to which the message replies.

References: A list of messageIds related to a thread.

**Consistency trade-off:**

Correctness is very important for email systems, so by design, we want to have a single primary for any given mailbox. In the event of a failover, the mailbox isn't accessible by clients, so their sync/update operation is paused until failover ends. It trades availability in favor of consistency.

#### **Deliverability**

Couple of factors to improve deliverability:

* Dedicated IPs: for sending email. Email providers are less likely to accept emails from new IP addresses that have no history.
* Classify emails: Send different categories of emails from different IP addresses. For example you may want to avoid sending marketing and other important emails from the same servers because it might make ISPs mark all emails as promotional.
* Email sender reputation: Warm up new email server IP address slowly to build a good reputation. According to Amazon Simple Email Service it takes 2 to 6 weeks to warm up a new IP address.
* Ban spammers quickly: to minimize impact of the server’s reputation
* Feedback processing: feedback from ISPs to keep complaint rate low and ban spam accounts quickly.

If an email fails to deliver or a user complains, one of the following is the outcome:

I. Hard bounce(rejected by ISP as recipient email is not valid)

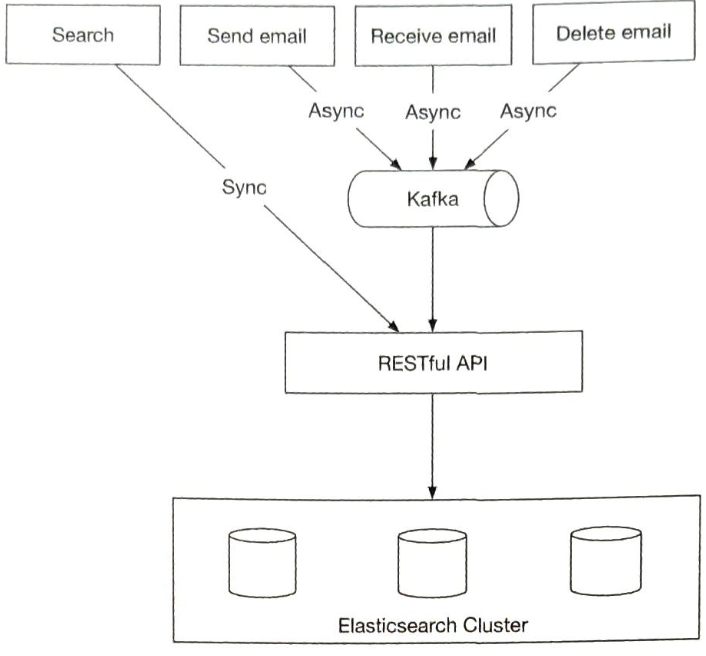
Ii. Soft bounce(ISP not available temporarily)  
Iii. Complaint(report spam usage)

#### **Search**

Basic mail search refers to searching for emails that contain any of the entered key- words in the subject or body. More advanced features include filtering by "From", "Subject", "Unread", or other attributes. On one hand, whenever an email is sent, received, or deleted, we need to perform reindexing. On the other hand, a search query is only run when a user presses the search button. This means the search feature in email systems has a lot more writes than reads.

#### 

**Option 1: Elastic Search:**

Because queries are mostly performed on the user’s own email server, we can group underlying documents to the same node using user\_id as the partition key.

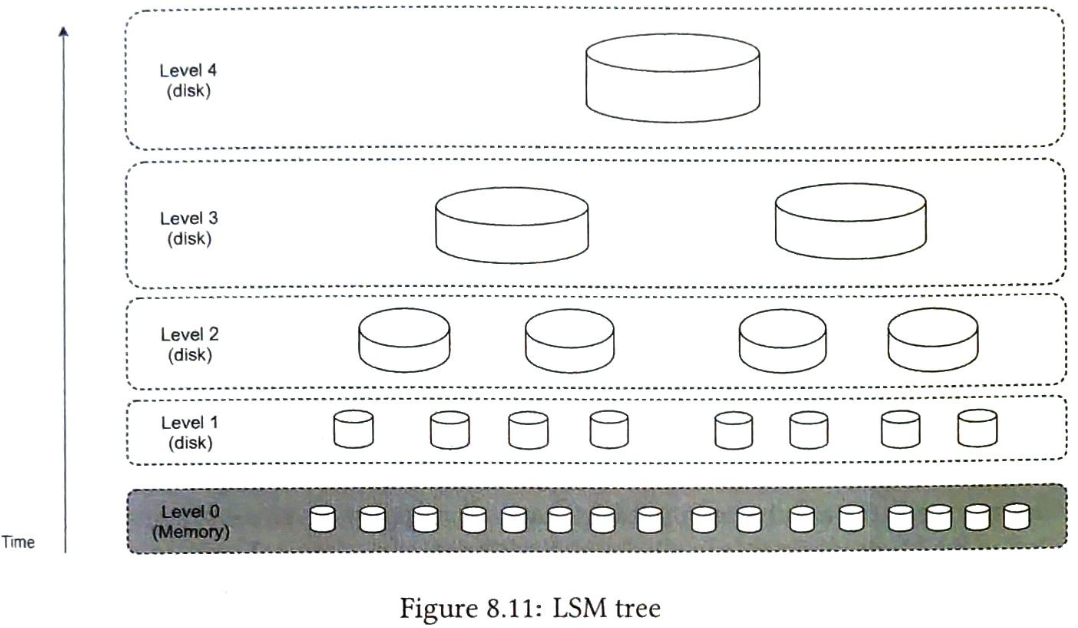
A search request user wise is synchronous. When events such as “send email”, “receive email” or “delete email” are triggered, nothing related to search needs to be done to the client. Reindexing is needed and it can be done with offline jobs. Kafka is used in the design to decouple services that trigger reindexing, from services that actually perform reindexing.

Elasticsearch supports full text search of emails very well. But one challenge of adding elastic search is to keep the primary email store in sync with it.

**Option 1: Custom Search solution:**

Main bottleneck while designing a custom search engine for email is disk I/O. Since the process of building the index is write heavy, a good strategy might be to use Log-Structured Merge-Tree(LSM) to structure the index data on disk. The write path is optimized by only performing sequential writes. LSM trees are the core data structure behind databases such as BigTable, Cassandra, and RocksDB. When a new email arrives, it is first added to level 0 in-memory cache, and when data size in memory reaches the predefined threshold, data is merged to the next level. Another reason to use LSM is to separate data that change frequently from those that don’t.

For example, email data usually doesn't change, but folder information tends to change more often due to different filter rules. In this case, we can separate them into two different sections, so that if a request is related to a folder change, we change only the folder and leave the email data alone.



Each approach has pros and cons:

| **Feature** | **Elasticsearch** | **Custom search engine** |
| --- | --- | --- |
| **Scalability** | Scalable to some extent | Easier to scale as we can optimize the system for the email use case |
| **System complexity** | Need to maintain two different systems: datastore and ElasticSearch | One system |
| **Data consistency** | Two copies of data. One in the metadata DB, and the other in elastic search. Data consistency is hard to maintain | A single of data in the metadata DB |
| **Data loss possibility** | No. Can rebuild the elastic search index from primary storage, in case of failure | NO |
| **Development effort** | Easy to integrate. To support large scale email search a dedicated Elastic search team might be needed. | Significant engineering effort to develop a customer search engine. |

#### **Scalibility**

