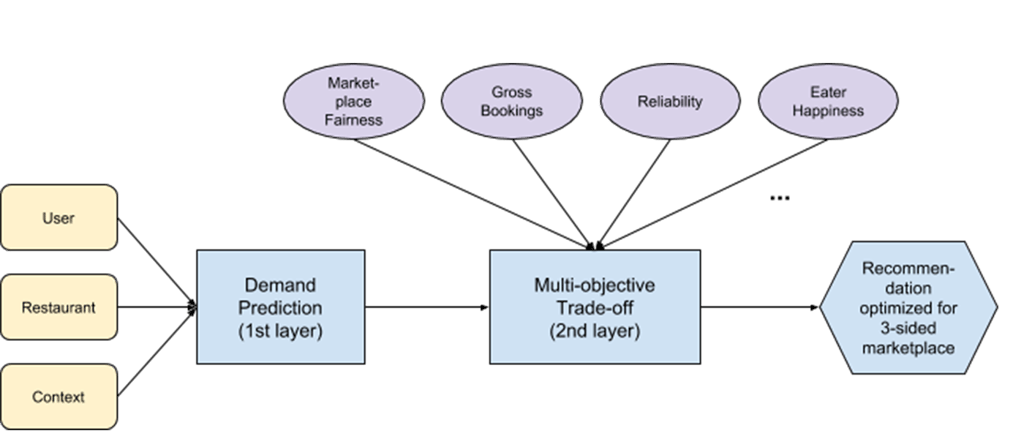
<https://www.uber.com/en-IE/blog/scaling-michelangelo/>

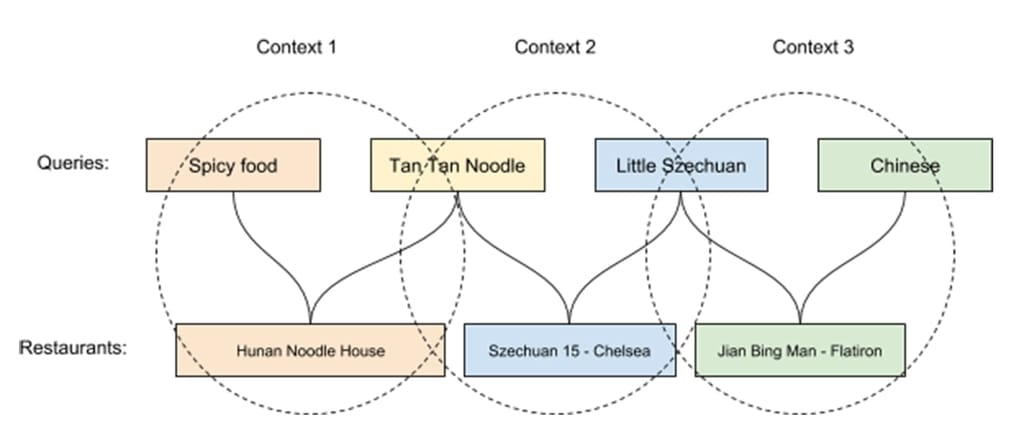
Food Discovery with Uber Eats: Recommending for the Marketplace

<https://www.uber.com/en-IE/blog/uber-eats-recommending-marketplace/>



Through multi-objective optimization, we can help eaters discover a diverse array of restaurants and ensure that our restaurant-partners receive a fair amount of exposure in the app based on eater interest.

# Food Discovery with Uber Eats: Building a Query Understanding Engine



<https://www.uber.com/en-IE/blog/uber-eats-query-understanding/?_ga=2.105497419.994849439.1670862526-2102806785.1670862526>

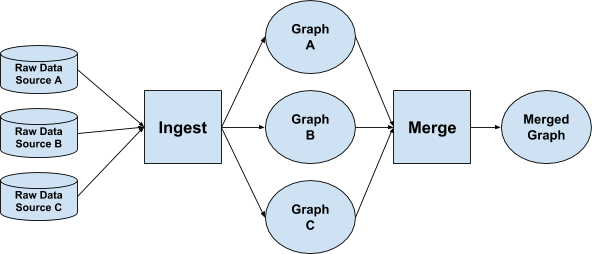
Through frameworks such as [multi-objective optimization](https://en.wikipedia.org/wiki/Multi-objective_optimization) and [multi-armed bandit](https://en.wikipedia.org/wiki/Multi-armed_bandit), we balance the needs of both restaurants and eaters in the Uber Eats marketplace.

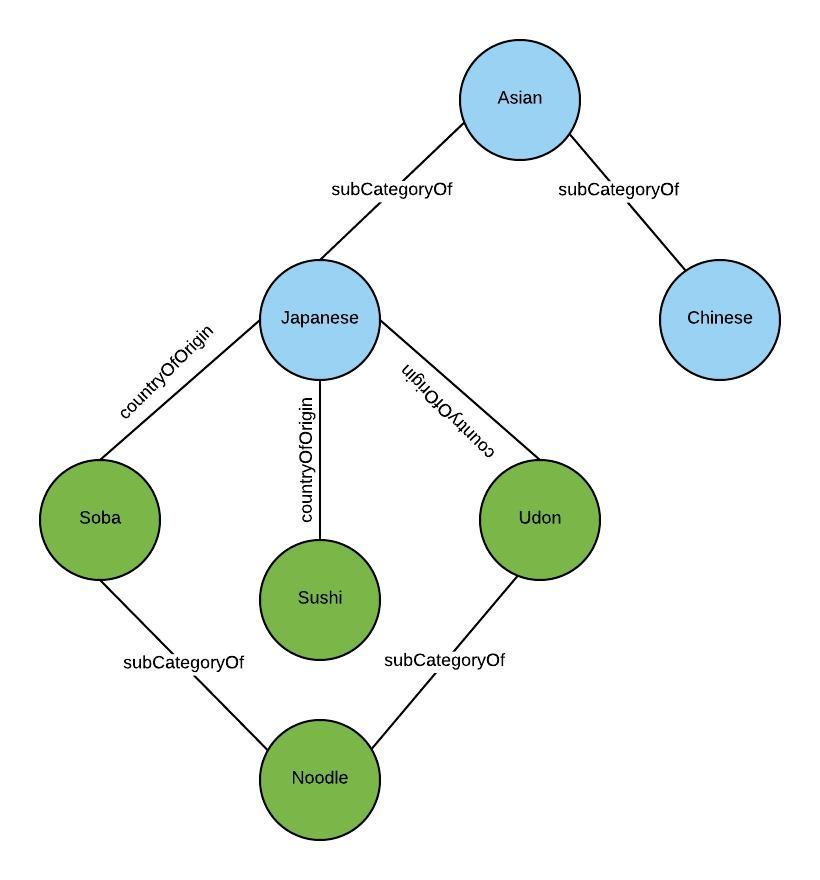
Consequently, we use [query understanding](https://en.wikipedia.org/wiki/Query_understanding) to figure out eater intent. Although query understanding is a common problem for different types of search engines, it poses unique challenges and additional opportunities when faced with food and restaurants.

### **Building a food knowledge graph**

At Uber, we are building a food-focused knowledge base to enable better understanding of food-related queries.

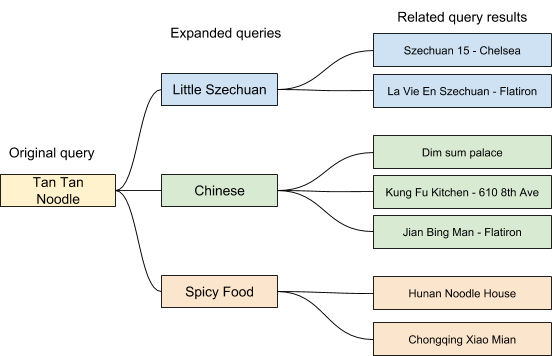
 Achieving this balance requires building an [ontology](https://en.wikipedia.org/wiki/Ontology_(information_science)), or language, to describe the graph, including properties of different entities and the relationships between them.





#### **Representation learning**

[Query expansion](https://en.wikipedia.org/wiki/Query_expansion) (QE) is a commonly used technique in search engines to improve recall coverage and search quality in general. QE is especially useful in the following two cases: 1) the eater’s search intent is ambiguous and can’t be fully captured by the query itself; and 2) the quality of results retrieved from the query is limited. For example, when an eater searches for “[tan tan noodles](https://en.wikipedia.org/wiki/Dandan_noodles),” it’s likely that the eater might also be interested in spicy Chinese food or other Szechuan restaurants; moreover, there might not be many restaurants that sell “tan tan noodles.”



* **How popular is the restaurant itself and what are the best features to determine popularity?** Do we rely on the rating, the historic total orders, or the most recent month’s total orders? The order/impression ratio?
* **What are the most representative attributes of the restaurant?** Does the restaurant prepare and deliver food faster than others? What type of cuisines does it offer?
* **What are the current contextual factors?** Is it breakfast time or dinner time? What are the current traffic conditions along the delivery path? Is it a weekday or a weekend?
* **What attributes best describe the eater?** How many orders has the eater placed so far or in the last month? What kind of cuisines does the eater order most frequently? Is this person a new eater? Did this eater put any dishes into their shopping cart from the restaurant they just clicked into?
* **What factors represents an eater’s preference for a particular restaurant?** Do we look at how often this eater has clicked into/ordered from this restaurant? Did the eater give this restaurant a high rating?
* **What would like-minded eaters order?** How do we represent and find similar eaters? How about similar restaurants and dishes? How do we cold start new eaters in the context of [collaborative filtering](https://en.wikipedia.org/wiki/Collaborative_filtering)?

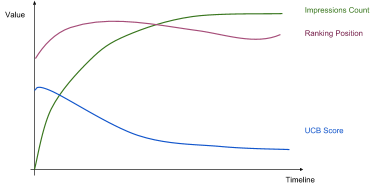
##### **Diversity**

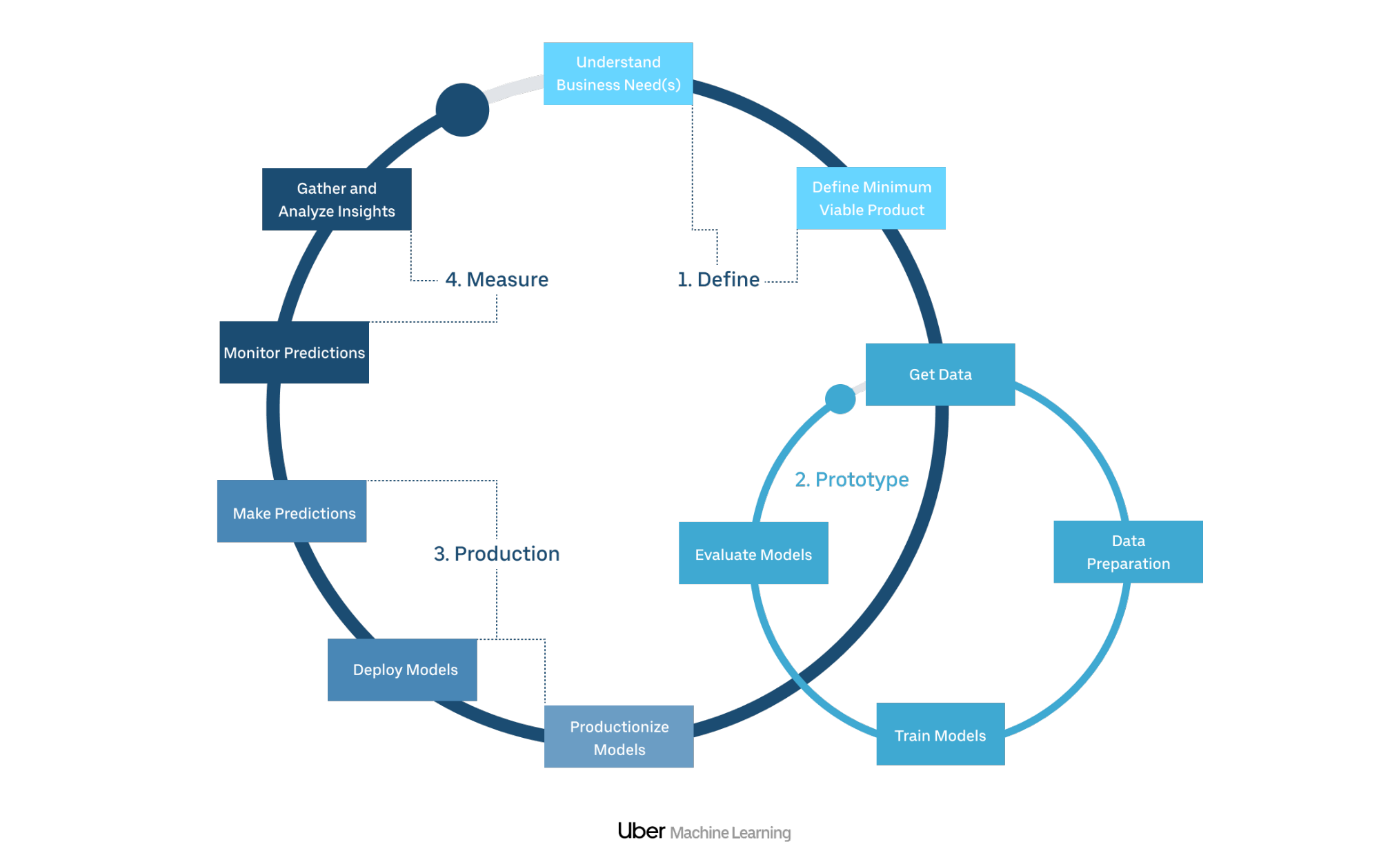
While optimizing restaurant and dish display for relevance to the eater is the obvious approach, it is not necessarily the best means to arrive at recommendations.

##### **Keeping eaters on our platform**

#### **Restaurant-partners**

To give new restaurants a fair opportunity to rank high and gather exposure, we used the [multi-armed bandit](https://en.wikipedia.org/wiki/Multi-armed_bandit) (MAB) framework.





# Using Deep Learning at Scale in Twitter’s Timelines

<https://blog.twitter.com/engineering/en_us/topics/insights/2017/using-deep-learning-at-scale-in-twitters-timelines>

In order to predict whether a particular Tweet would be engaging to you, our models consider characteristics (or features) of:

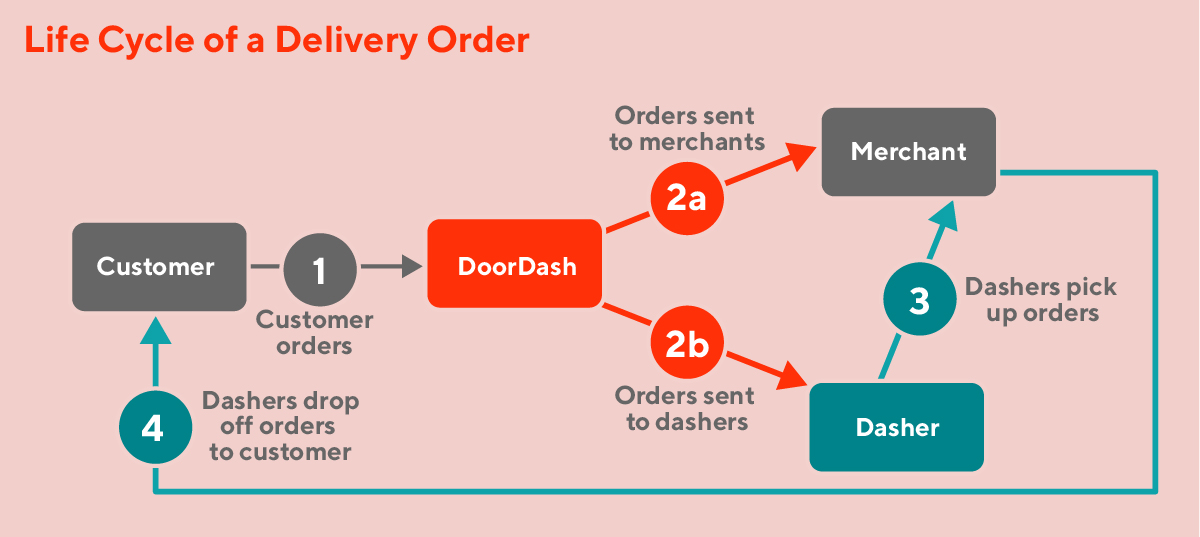
* The Tweet itself: its recency, presence of media cards (image or video), total interactions (e.g. number of Retweets or likes)
* The Tweet’s author: your past interactions with this author, the strength of your connection to them, the origin of your relationship
* You: Tweets you found engaging in the past, how often and how heavily you use Twitter

 Deep learning modules can be composed in various ways (stacked, concatenated, etc.) to form a computational graph.

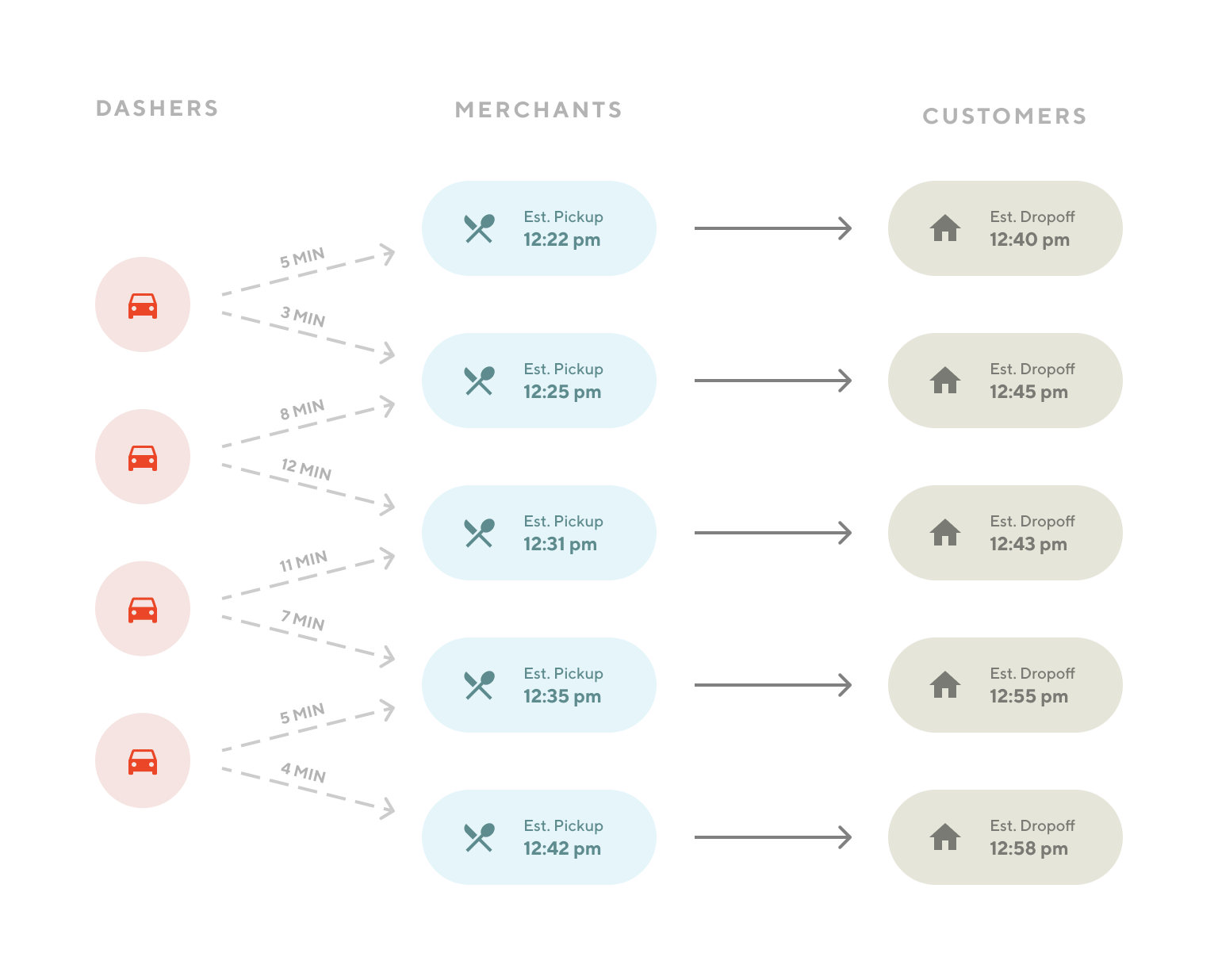
The parameters of this graph can then be learned, typically by using back-propagation and SGD (Stochastic Gradient Descent) on mini-batches.

<https://doordash.engineering/2020/02/28/next-generation-optimization-for-dasher-dispatch-at-doordash/>

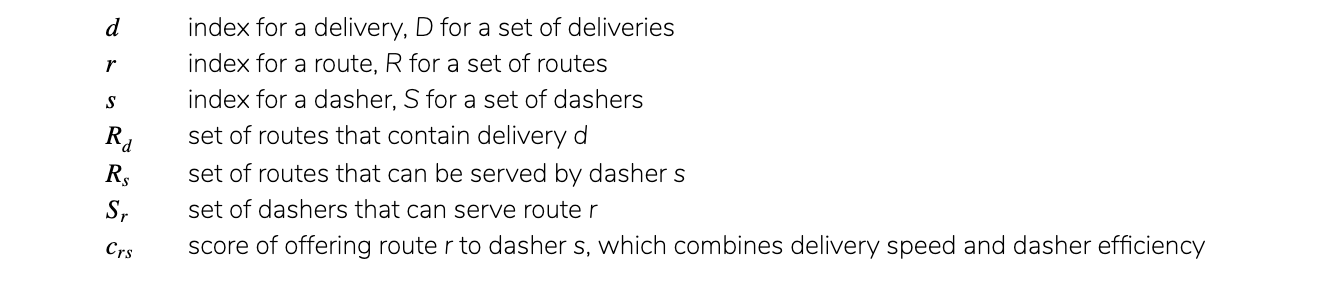
# Next-Generation Optimization for Dasher Dispatch at DoorDash



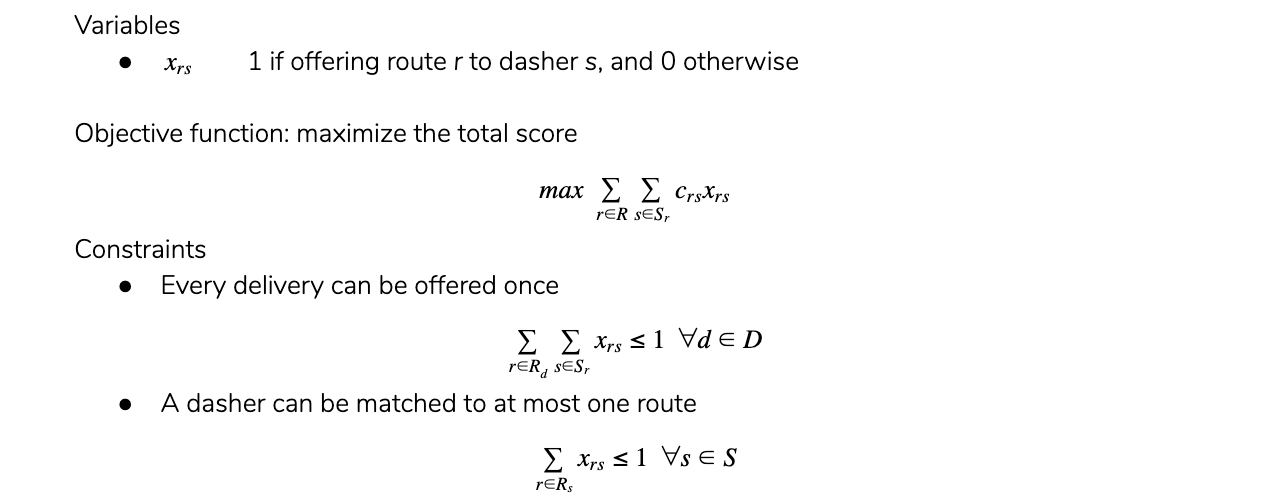
The problem can then be solved using the [Hungarian algorithm](https://en.wikipedia.org/wiki/Hungarian_algorithm). There are two limitations with this approach: 1) though the Hungarian algorithm is polynomial, the runtime of large instances is excessive for our real-time dynamic system; 2) it doesn’t support more complicated routes with 2 or more deliveries.



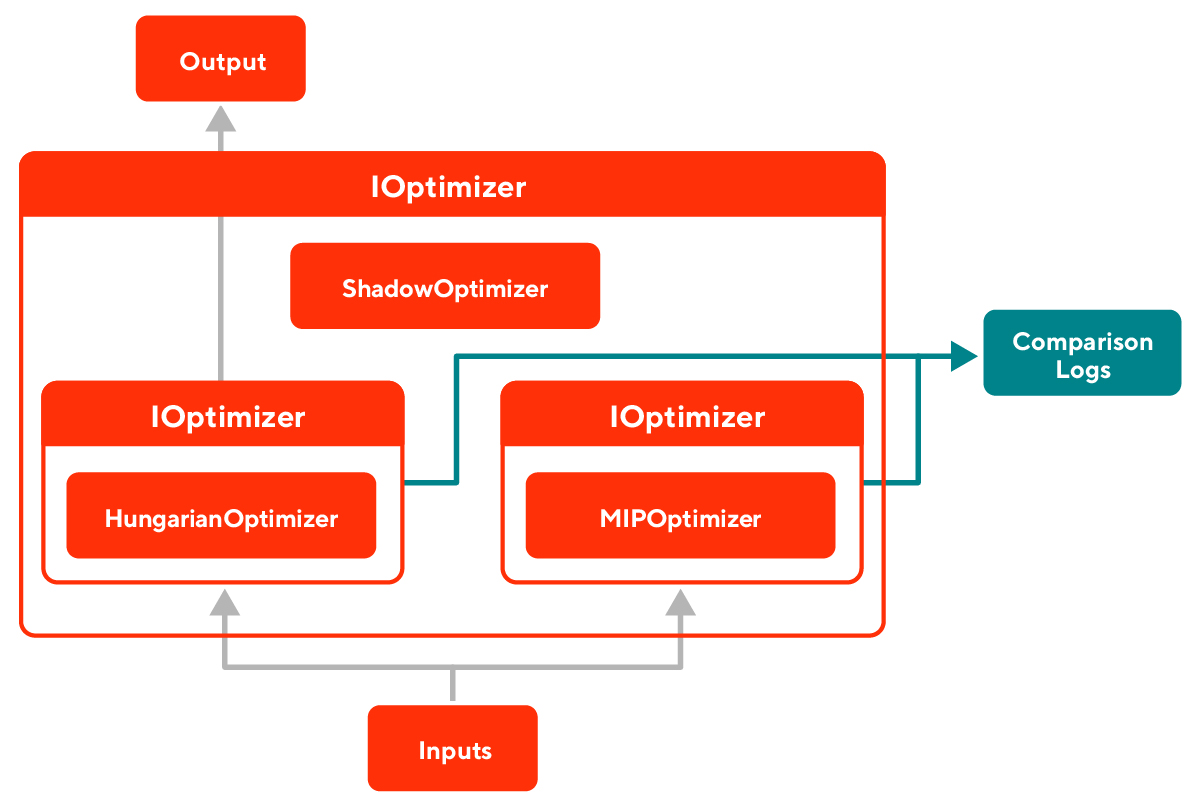
Furthermore, the solver provides flexibility in formulating the problem as a [vehicle routing problem](https://en.wikipedia.org/wiki/Vehicle_routing_problem), which allows multiple deliveries in a route.



## Optimization solvers

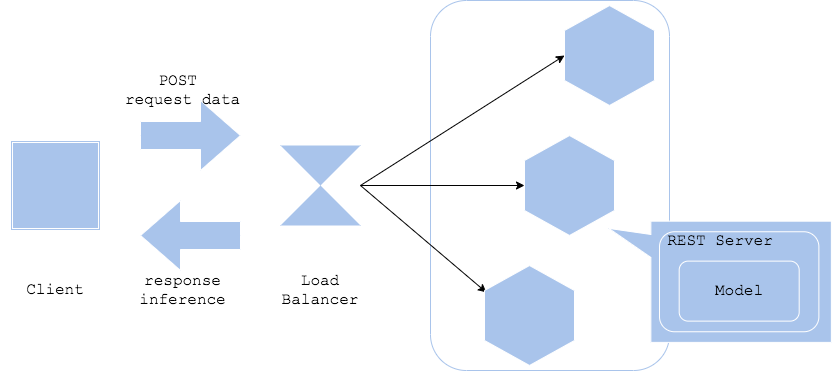


With this data we found that the two optimizers’ output matched more than 99% of the time, and the only mismatches were due to situations where there were multiple equally-good solutions.

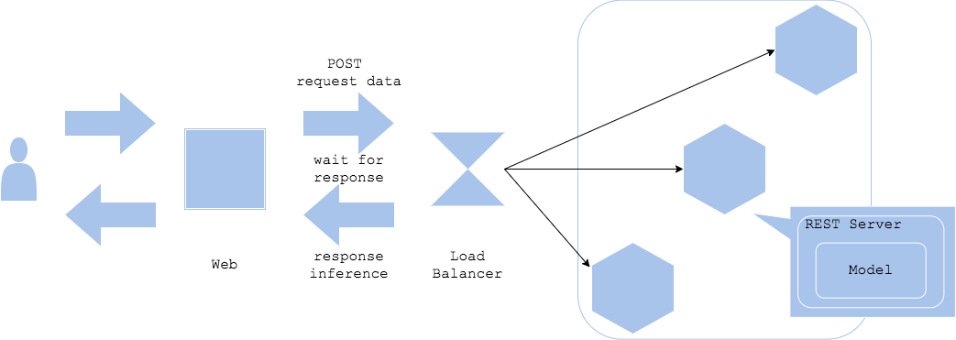


<https://github.com/prakhargurawa/ml-system-design-pattern>

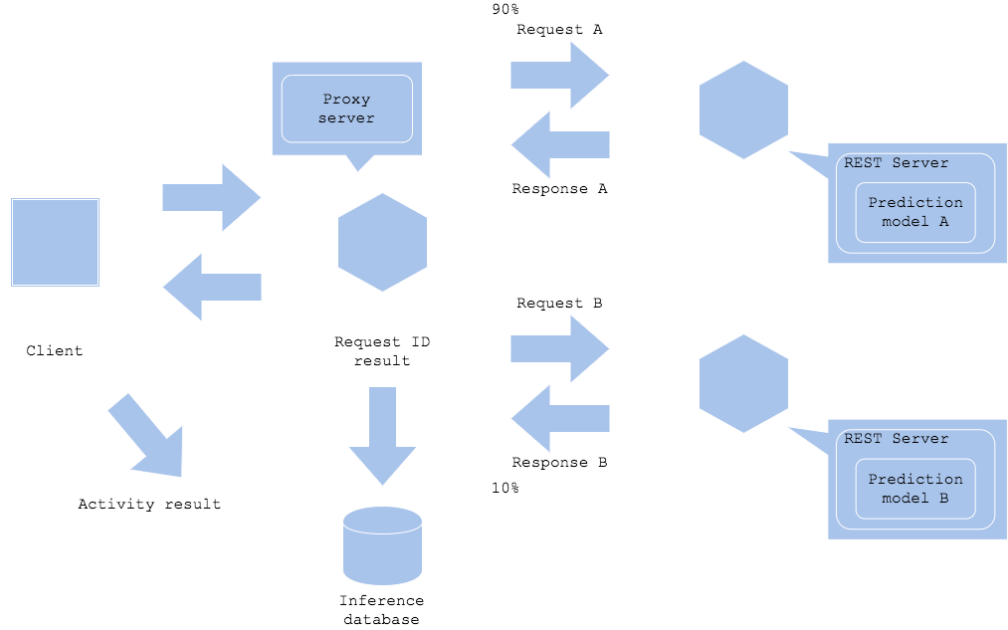
# Web single pattern



# Synchronous pattern



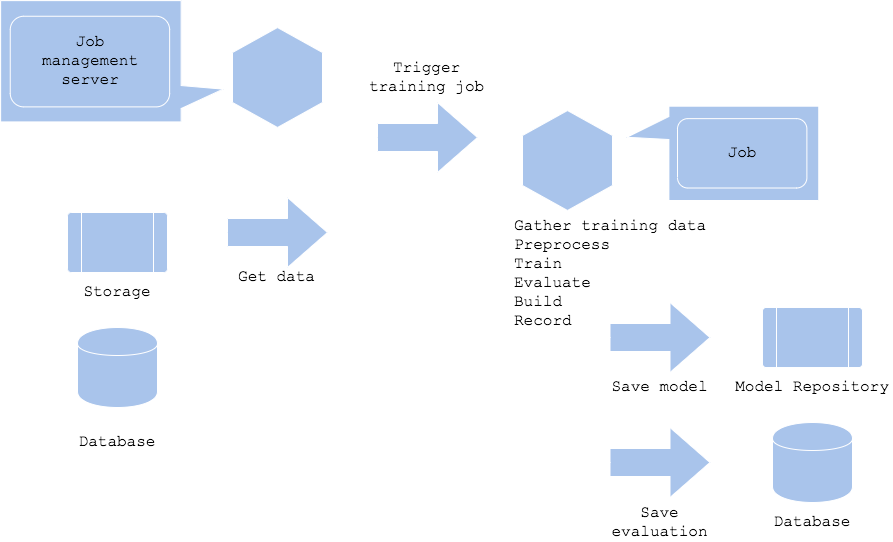
AB Test



# Batch training pattern

If you need to train your machine learning model regularly, the batch training pattern is applicable. In the pattern, you will define the training as a batch job and configure the trigger and schedule in a job management server. The server will execute the batch job. One of the easiest ways to define it is with Linux crontab, and it is also possible to make it with services in cloud. Or, it is possible to use a job management server.  
The pattern is one of the most common architecture to train a model offline. The workflow would be like:

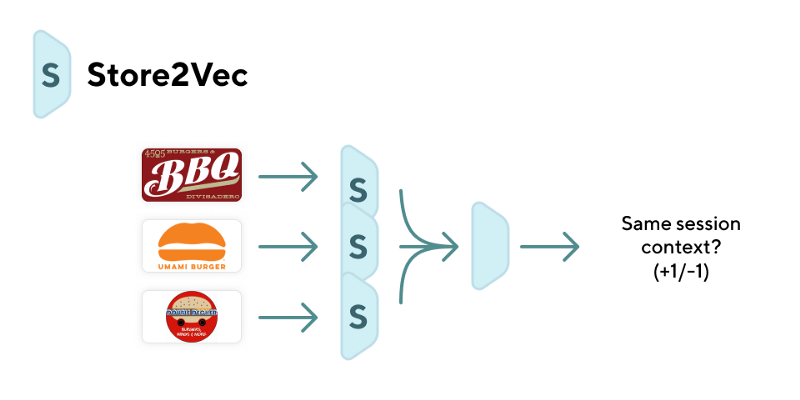
1. Retrieve data from DWH (may need data cleansing)
2. Preprocess data
3. Train
4. Evaluate
5. Build model to prediction server
6. Store the model and server, and record the evaluation



# Personalized Store Feed with Vector Embeddings

<https://doordash.news/company/personalized-store-feed-with-vector-embeddings/>

By incorporating latent information, as well as preparing a training pipeline and a gradient-boosted machine setup we use in other systems at DoorDash, we’ve been able to see an increase in click through rate by another 5% in initial email tests and are in the process of testing and rolling out these changes more broadly in email and in-app.



For word context, we found a context window size of 5 to work the best. As quality constraints, we enforce minimum thresholds on number of stores in a session and number of sessions a store appears in.

we take the cosine distance between the store’s vector and the consumer’s vector.

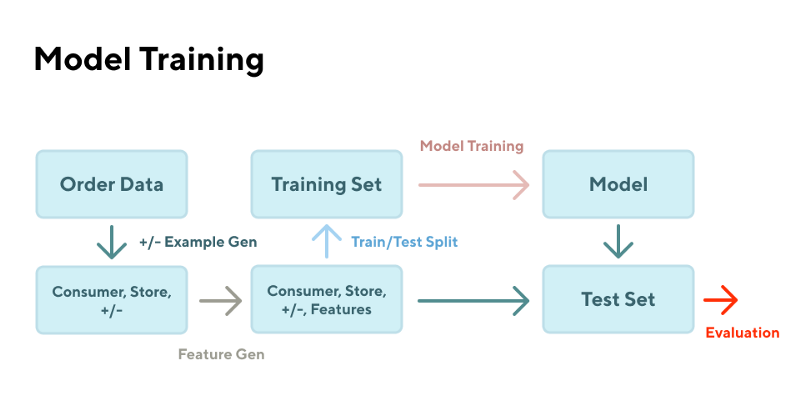
The points are plotted using [t-SNE](https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding) and the [Tensorflow embedding projector](http://projector.tensorflow.org/). The distances listed on the right are the cosine distance between the consumer vector and the store vector.

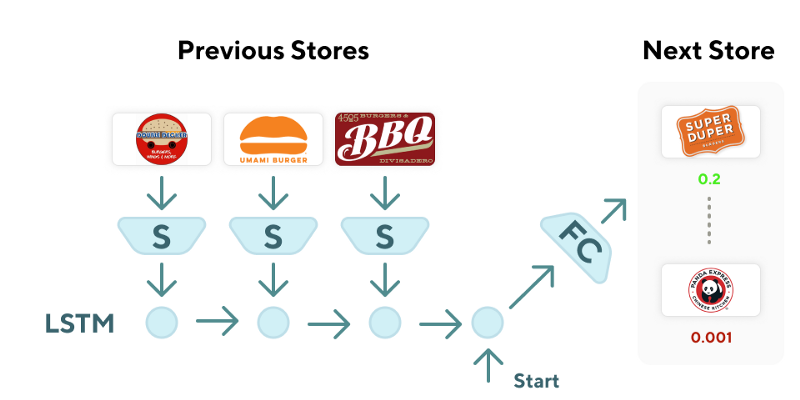
**Train/test split:** We split 70% training and 30% test as a time split so that we are not testing on data that occurred before data we trained on.

**Feature generation**: Based on data for consumer and stores, we extract many features having to do with the annotated data on consumer and stores such as categories, rating, popularity, and browse / click / order information.

**Model training**: We train logistic regression and gradient-boosted machine (GBM) models. For GBM models, we use [LightGBM](https://github.com/Microsoft/LightGBM). These are the same frameworks we use for many other machine learning systems at DoorDash such as prep time prediction and batching prediction

**Model evaluation**: The model is predicting P(order | consumer, store) and is a binary classifier. To evaluate it for this ranking problem, we use area under curve (AUC) of the precision/recall curve. This provides an evaluation metric that does not change if the score values are inflated or deflated but the ranking remains the same. We also output business metrics to check for the models such as average delivery fee, average rating, and check for example users with order history conforming to certain patterns in order to sanity check the output models.





# Powering Search & Recommendations at DoorDash

<https://doordash.news/company/powering-search-recommendations-at-doordash/>

We use [Elasticsearch](https://www.elastic.co/products) to power the consumer search for our website and apps. Elasticsearch is an open source, distributed, Lucene-based inverted index that provides search engine capabilities without reinventing the wheel.

The first is the **indexing module (offline)**. This component reads the store object from the database (Postgres in our case) and writes it to Elasticsearch for bootstrapping, as well as for partial asynchronous updates on the database store object.

Second is the **search module (online)**. Web and mobile clients call the backend search API with the specified consumer location. A JSON-based Elasticsearch query is constructed at the Django backend to call Elasticsearch. The query is executed inside Elasticsearch to retrieve relevant results, which are deserialized and returned to the client. The Elasticsearch query is primarily designed to achieve two purposes

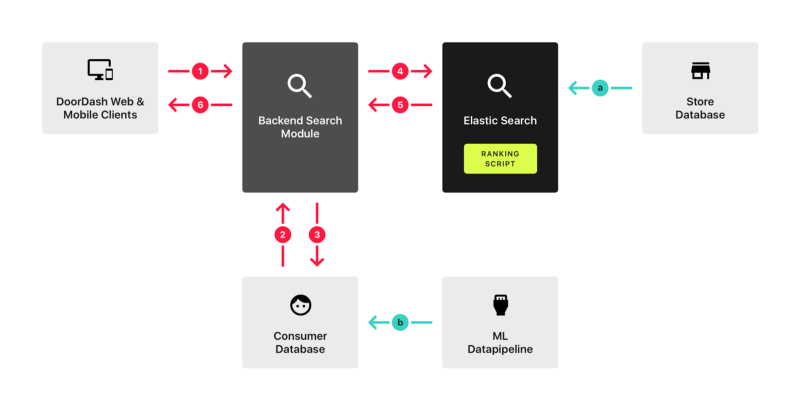
* Selection: Out of all the available stores, only select those that are orderable from the consumer’s address. This is primarily achieved by the [geoshape](https://www.elastic.co/guide/en/elasticsearch/reference/current/geo-shape.html) features of Elasticsearch. How we compute a geoshape to get an accurate driving distance for each address and store pair is a discussion for a separate blog post.
* Ranking or scoring: Out of the selected subset of stores, we need to rank them according to relevance. Before the personalized ranking we ran a number of sorting experiments including ranking by popularity, price, delivery, estimated time of arrival, ratings, and more. The main learning from the experiments was that there was no global best ranking for every user, but rather the notion of “best” varied across each user, which led us to use personalization.
* *c\_i*: consumer with unique id *i*
* *s\_j*: store with unique id *j*
* d(*c\_i*): data profile of consumer *c\_i*
* d(*s\_j*): data profile of store *s\_j*
* *f^k*: kth feature in the ML model
* *f^k\_ij*: value of kth feature for (*c\_i*, *s\_j*) pair

The data profile of store s\_j is stored in Elasticsearch by the indexing pipeline.

We use this data to compute the probability of consumer *c\_i* ordering from *s\_j* given by:

*Probability(c\_i orders from store s\_j) = 1/(1+e^(-1\* ( w\_k \* f^k\_ij)) )*where *w\_k* is the weight of kth feature.

We trained the data using the [logistic regression](https://en.wikipedia.org/wiki/Logistic_regression) model to estimate w\_k for our dataset.



1. Elasticsearch ranking script, which is an implementation of the logistic regression scoring function described in the ML modeling section above, is executed as part of the Elasticsearch JVM process. This script is essentially a function of d(*c\_i*) and d(*s\_j*). The script gets d(*c\_i*) as arguments from step 4 and gets d(*s\_j*) as part of the index data, which was stored from offline step a. The script generates the score and Elasticsearch ranks them by script score.
2. **Horizontally scalable:** Higher search volume results in more heap usage, which can be addressed by adding more nodes to the Elasticsearch cluster or increasing head size per node.

<https://github.com/alirezadir/Production-Level-Deep-Learning>

## Efficiency

* Latency: response time, the delay to obtain the first piece of data.
* Bandwidth: throughput, amount of data delivered in a given time.

Load balancing

## Algorithms

* Least connection
* Least response time
* Least bandwidth
* Round robin
* Weighted round robin
* IP hash

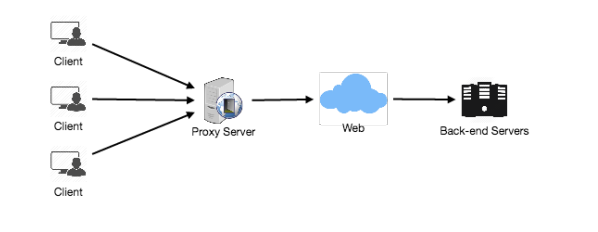
Good examples of horizontal scaling are Cassandra and MongoDB as they both provide an easy way to scale horizontally by adding more machines to meet growing needs.

Similarly, a good example of vertical scaling is MySQL as it allows for an easy way to scale vertically by switching from smaller to bigger machines. However, this process often involves downtime.

Health Checks - Load balancers should only forward traffic to “healthy” backend servers. To monitor the health of a backend server, “health checks” regularly attempt to connect to backend servers to ensure that servers are listening. If a server fails a health check, it is automatically removed from the pool, and traffic will not be forwarded to it until it responds to the health checks again.

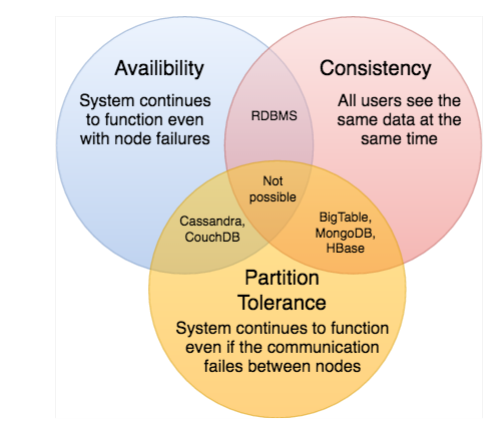
Cache eviction policies Following are some of the most common cache eviction policies: 1. First In First Out (FIFO): The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before. 2. Last In First Out (LIFO): The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before. 3. Least Recently Used (LRU): Discards the least recently used items first. 4. Most Recently Used (MRU): Discards, in contrast to LRU, the most recently used items first. 5. Least Frequently Used (LFU): Counts how often an item is needed. Those that are used least often are discarded first. 6.Random Replacement (RR): Randomly selects a candidate item and discards it to make space when necessary.

How do Indexes decrease write performance? An index can dramatically speed up data retrieval but may itself be large due to the additional keys, which slow down data insertion & update. When adding rows or making updates to existing rows for a table with an active index, we not only have to write the data but also have to update the index. This will decrease the write performance. This performance degradation applies to all insert, update, and delete operations for the table. For this reason, adding unnecessary indexes on tables should be avoided and indexes that are no longer used should be removed. To reiterate, adding indexes is about improving the performance of search queries. If the goal of the database is to provide a data store that is often written to and rarely read from, in that case, decreasing the performance of the more common operation, which is writing, is probably not worth the increase in performance we get from reading.



NoSQL Following are the most common types of NoSQL: Key-Value Stores: Data is stored in an array of key-value pairs. The ‘key’ is an attribute name which is linked to a ‘value’. Well-known key-value stores include Redis, Voldemort, and Dynamo. Document Databases: In these databases, data is stored in documents (instead of rows and columns in a table) and these documents are grouped together in collections. Each document can have an entirely different structure. Document databases include the CouchDB and MongoDB. Wide-Column Databases: Instead of ‘tables,’ in columnar databases we have column families, which are containers for rows. Unlike relational databases, we don’t need to know all the columns up front and each row doesn’t have to have the same number of columns. Columnar databases are best suited for analyzing large datasets - big names include Cassandra and HBase. Graph Databases: These databases are used to store data whose relations are best represented in a graph. Data is saved in graph structures with nodes (entities), properties (information about the entities), and lines (connections between the entities). Examples of graph database include Neo4J and InfiniteGraph.

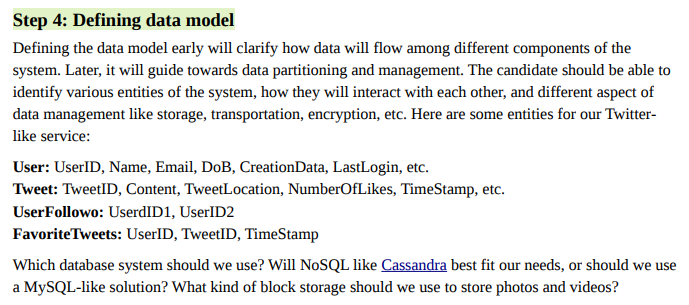
. Making the most of cloud computing and storage. Cloud-based storage is an excellent cost-saving solution but requires data to be easily spread across multiple servers to scale up. Using commodity (affordable, smaller) hardware on-site or in the cloud saves you the hassle of additional software and NoSQL databases like Cassandra are designed to be scaled across multiple data centers out of the box, without a lot of headaches.

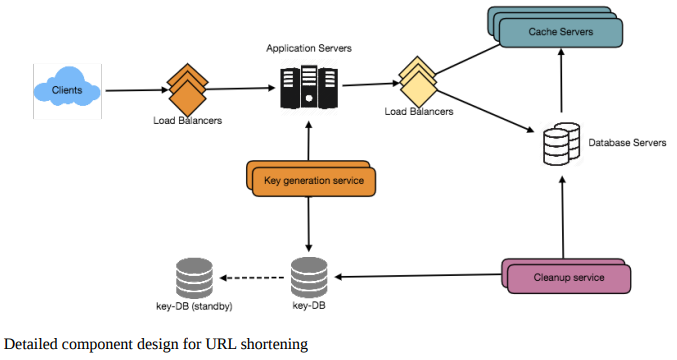


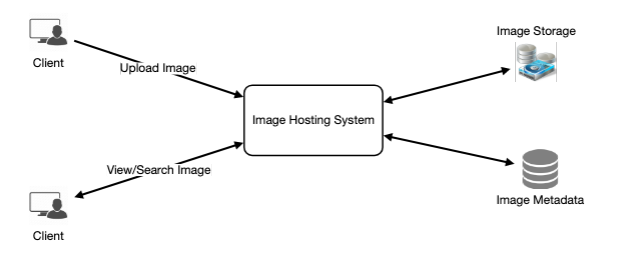
Consistent hashing is a very useful strategy for distributed caching system and DHTs. It allows us to distribute data across a cluster in such a way that will minimize reorganization when nodes are added or removed. Hence, the caching system will be easier to scale up or scale down.

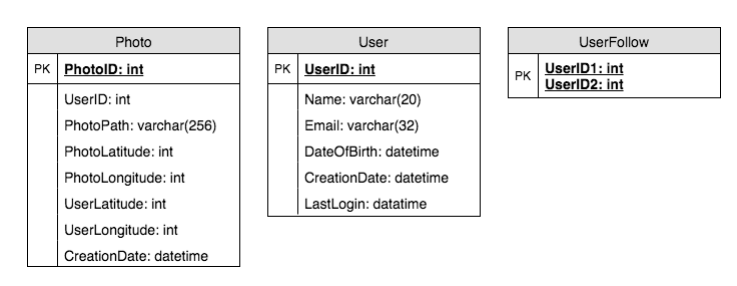
postTweet(user\_id, tweet\_data, tweet\_location, user\_location, timestamp, …)

markTweetFavorite(user\_id, tweet\_id, timestamp, …)

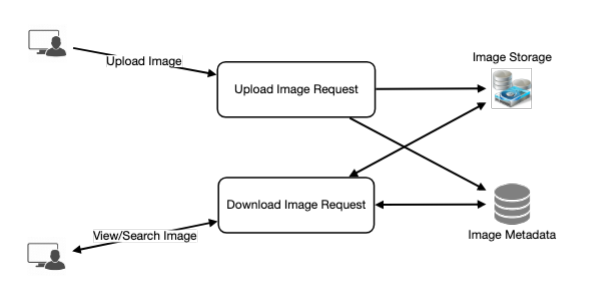


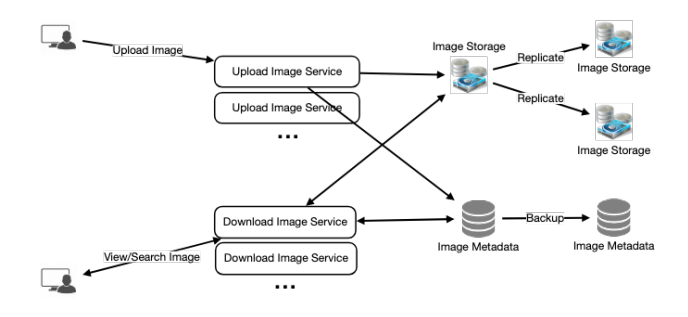


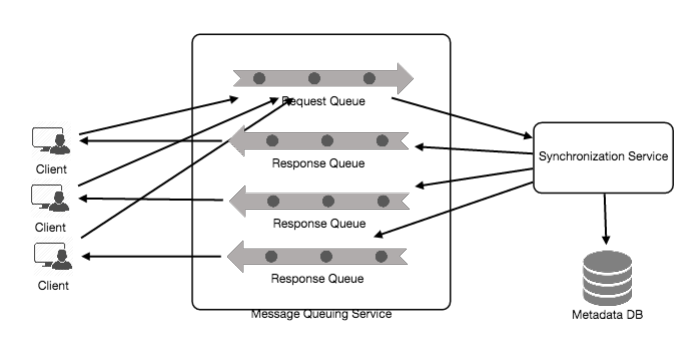


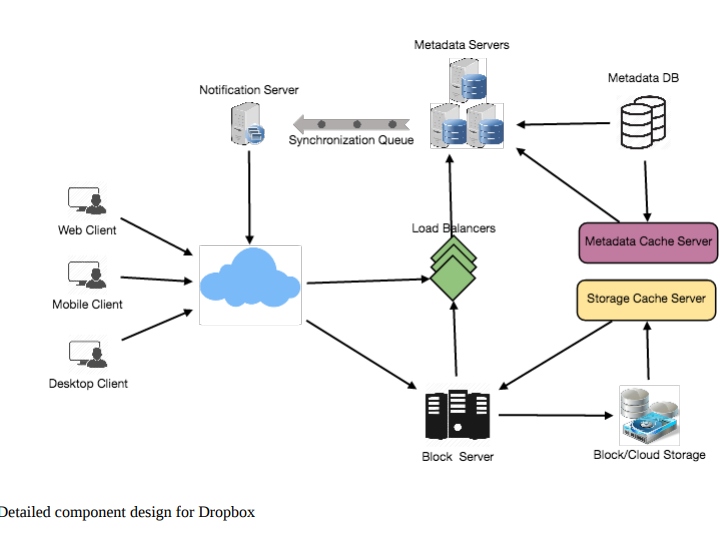


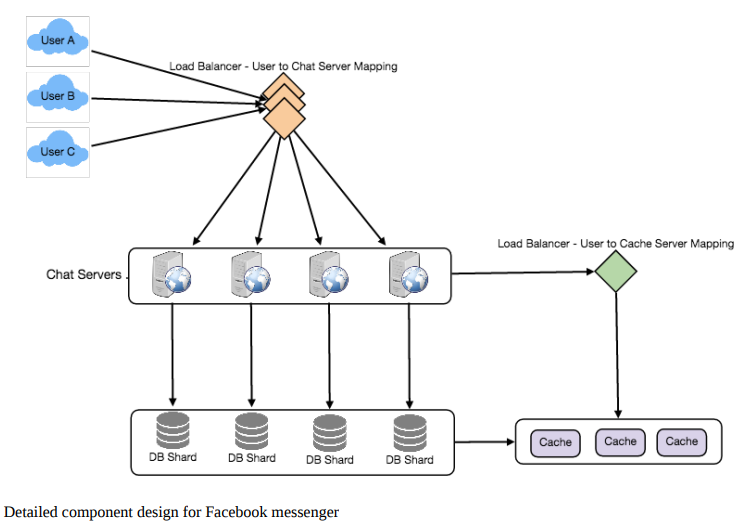
Cassandra or key-value stores in general, always maintain a certain number of replicas to offer reliability. Also, in such data stores, deletes don’t get applied instantly, data is retained for certain days (to support undeleting) before getting removed from the system permanently.

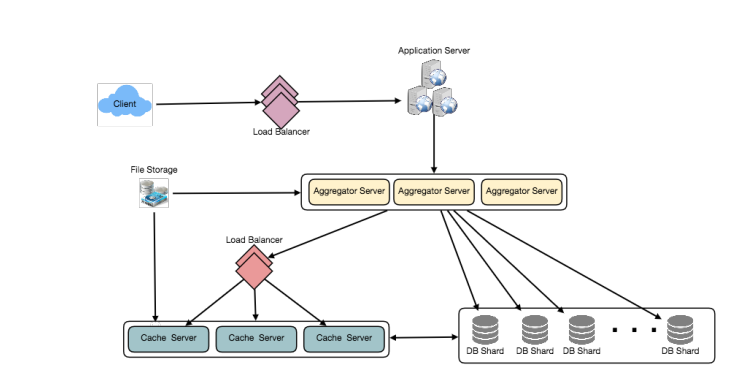


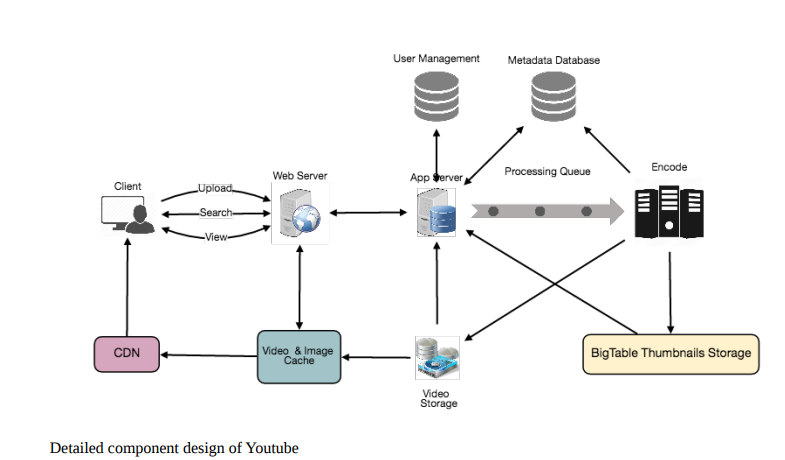








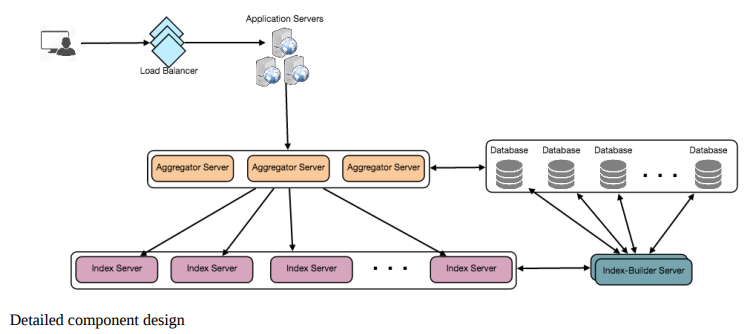


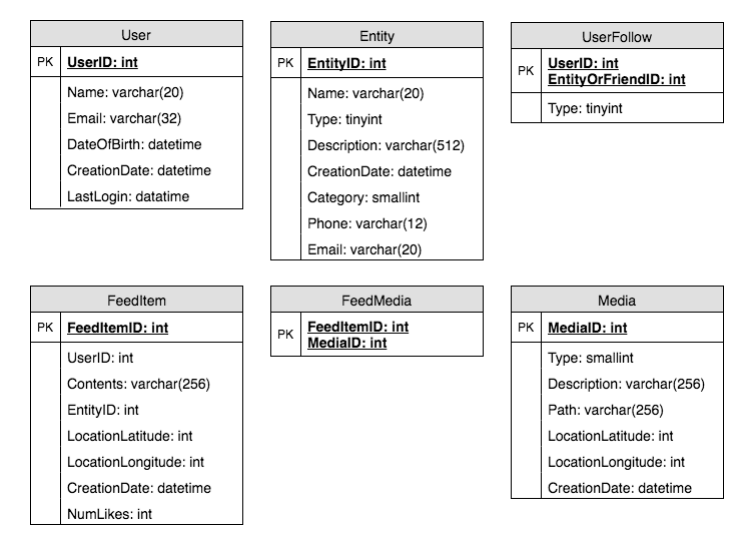


Bigtable can be a reasonable choice here as it combines multiple files into one block to store on the disk and is very efficient in reading a small amount of data. Both of these are the two most significant requirements of our service. Keeping hot thumbnails in the cache will also help in improving the latencies and, given that thumbnails files are small in size, we can easily cache a large number of such files in memory.

10. Load Balancing We should use Consistent Hashing among our cache servers, which will also help in balancing the load between cache servers. Since we will be using a static hash-based scheme to map videos to hostnames it can lead to an uneven load on the logical replicas due to the different popularity of each video.

search(api\_dev\_key, search\_terms, maximum\_results\_to\_return, sort, page\_token)





(SELECT FeedItemID FROM FeedItem WHERE UserID in ( SELECT EntityOrFriendID FROM UserFollow WHERE UserID = and type = 0(user)) ) UNION (SELECT FeedItemID FROM FeedItem WHERE EntityID in ( SELECT EntityOrFriendID FROM UserFollow WHERE UserID = and type = 1(entity)) ) ORDER BY Cre

Struct { LinkedHashMap feedItems; DateTime lastGenerated; }