Introduction to architecting systems for scale.

April 4, 2011. Filed under [infrastructure](https://lethain.com/tags/infrastructure/)[architecture](https://lethain.com/tags/architecture/)

Few computer science or software development programs attempt to teach the building blocks of scalable systems. Instead, system architecture is usually picked up on the job by [working through the pain of a growing product](http://engineering.twitter.com/2010/06/perfect-stormof-whales.html) or by working with engineers who have already learned through that suffering process.

In this post I'll attempt to document some of the scalability architecture lessons I've learned while working on systems at [Yahoo!](http://yahoo.com/) and [Digg](http://digg.com/).

I've attempted to maintain a color convention for diagrams:

1. *green* is an external request from an external client (an HTTP request from a browser, etc),
2. *blue* is your code running in some container (a Django app running on [mod\_wsgi](http://code.google.com/p/modwsgi/), a Python script listening to [RabbitMQ](http://www.rabbitmq.com/), etc), and
3. *red* is a piece of infrastructure (MySQL, [Redis](http://redis.io/), RabbitMQ, etc).

Load balancing

The ideal system increases capacity linearly with adding hardware. In such a system, if you have one machine and add another, your capacity would double. If you had three and you add another, your capacity would increase by 33%. Let's call this *horizontal scalability*.

On the failure side, an ideal system isn't disrupted by the loss of a server. Losing a server should simply decrease system capacity by the same amount it increased overall capacity when it was added. Let's call this *redundancy*.

Both horizontal scalability and redundancy are usually achieved via load balancing.

(This article won't address *vertical scalability*, as it is usually an undesirable property for a large system, as there is inevitably a point where it becomes cheaper to add capacity in the form on additional machines rather than additional resources of one machine, and redundancy and vertical scaling can be at odds with one-another.)

Load balancing is the process of spreading requests across multiple resources according to some metric (random, round-robin, random with weighting for machine capacity, etc) and their current status (available for requests, not responding, elevated error rate, etc).

Load needs to be balanced between user requests and your web servers, but must also be balanced at every stage to achieve full scalability and redundancy for your system. A moderately large system may balance load at three layers:

1. user to your web servers,
2. web servers to an internal platform layer,
3. internal platform layer to your database.

There are a number of ways to implement load balancing.

Smart clients

Adding load-balancing functionality into your database (cache, service, etc) client is usually an attractive solution for the developer. Is it attractive because it is the simplest solution? Usually, no. Is it seductive because it is the most robust? Sadly, no. Is it alluring because it'll be easy to reuse? Tragically, no.

*Developers lean towards smart clients because they are developers, and so they are used to writing software to solve their problems, and smart clients are software.*

With that caveat in mind, what is a smart client? It is a client which takes a pool of service hosts and balances load across them, detects downed hosts and avoids sending requests their way (they also have to detect recovered hosts, deal with adding new hosts, etc, making them fun to get working decently and a terror to setup).

Hardware load balancers

The most expensive–but very high performance–solution to load balancing is to buy a dedicated hardware load balancer (something like a [Citrix NetScaler](http://www.citrix.com/English/ps2/products/product.asp?contentID=21679)). While they can solve a remarkable range of problems, hardware solutions are remarkably expensive, and they are also "non-trivial" to configure.

As such, generally even large companies with substantial budgets will often avoid using dedicated hardware for all their load-balancing needs; instead they use them only as the first point of contact from user requests to their infrastructure, and use other mechanisms (smart clients or the hybrid approach discussed in the next section) for load-balancing for traffic within their network.

Software load balancers

If you want to avoid the pain of creating a smart client, and purchasing dedicated hardware is excessive, then the universe has been kind enough to provide a hybrid: software load-balancers.

[HAProxy](http://haproxy.1wt.eu/) is a great example of this approach. It runs locally on each of your boxes, and each service you want to load-balance has a locally bound port. For example, you might have your platform machines accessible via localhost:9000, your database read-pool at localhost:9001 and your database write-pool at localhost:9002. HAProxy manages healthchecks and will remove and return machines to those pools according to your configuration, as well as balancing across all the machines in those pools as well.

For most systems, I'd recommend starting with a software load balancer and moving to smart clients or hardware load balancing only with deliberate need.

Caching

Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have, as well as making otherwise unattainable product requirements feasible.

Caching consists of: precalculating results (e.g. the number of visits from each referring domain for the previous day), pre-generating expensive indexes (e.g. suggested stories based on a user's click history), and storing copies of frequently accessed data in a faster backend (e.g. [Memcache](http://memcached.org/) instead of [PostgreSQL](http://www.postgresql.org/).

In practice, caching is important earlier in the development process than load-balancing, and starting with a consistent caching strategy will save you time later on. It also ensures you don't optimize access patterns which can't be replicated with your caching mechanism or access patterns where performance becomes unimportant after the addition of caching (I've found that many heavily optimized [Cassandra](http://cassandra.apache.org/) applications are a challenge to cleanly add caching to if/when the database's caching strategy can't be applied to your access patterns, as the datamodel is generally inconsistent between the Cassandra and your cache).

Application vs. database caching

There are two primary approaches to caching: application caching and database caching (most systems rely heavily on both).

Application caching requires explicit integration in the application code itself. Usually it will check if a value is in the cache; if not, retrieve the value from the database; then write that value into the cache (this value is especially common if you are using a cache which observes the [least recently used caching algorithm](http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used)). The code typically looks like (specifically this is a *read-through cache*, as it reads the value from the database into the cache if it is missing from the cache):

key = "user.%s" % user\_id

user\_blob = memcache.get(key)

if user\_blob is None:

user = mysql.query("SELECT \* FROM users WHERE user\_id=\"%s\"", user\_id)

if user:

memcache.set(key, json.dumps(user))

return user

else:

return json.loads(user\_blob)

The other side of the coin is database caching.

When you flip your database on, you're going to get some level of default configuration which will provide some degree of caching and performance. Those initial settings will be optimized for a generic usecase, and by tweaking them to your system's access patterns you can generally squeeze a great deal of performance improvement.

The beauty of database caching is that your application code gets faster "for free", and a talented DBA or operational engineer can uncover quite a bit of performance without your code changing a whit (my colleague Rob Coli spent some time recently optimizing our configuration for Cassandra row caches, and was succcessful to the extent that he spent a week harassing us with graphs showing the I/O load dropping dramatically and request latencies improving substantially as well).

In-memory caches

The most potent–in terms of raw performance–caches you'll encounter are those which store their entire set of data in memory. [Memcached](http://memcached.org/) and [Redis](http://redis.io/) are both examples of in-memory caches (caveat: Redis can be configured to store some data to disk). This is because accesses to RAM are [orders of magnitude](http://en.wikipedia.org/wiki/RAM_disk) faster than those to disk.

On the other hand, you'll generally have far less RAM available than disk space, so you'll need a strategy for only keeping the hot subset of your data in your memory cache. The most straightforward strategy is [least recently used](http://en.wikipedia.org/wiki/Cache_algorithms#Least_Recently_Used), and is employed by Memcache (and Redis as of 2.2 can be configured to employ it as well). LRU works by evicting less commonly used data in preference of more frequently used data, and is almost always an appropriate caching strategy.

Content distribution networks

A particular kind of cache (some might argue with this usage of the term, but I find it fitting) which comes into play for sites serving large amounts of static media is the *content distribution network.*

CDNs take the burden of serving static media off of your application servers (which are typically optimzed for serving dynamic pages rather than static media), and provide geographic distribution. Overall, your static assets will load more quickly and with less strain on your servers (but a new strain of business expense).

In a typical CDN setup, a request will first ask your CDN for a piece of static media, the CDN will serve that content if it has it locally available (HTTP headers are used for configuring how the CDN caches a given piece of content). If it isn't available, the CDN will query your servers for the file and then cache it locally and serve it to the requesting user (in this configuration they are acting as a read-through cache).

If your site isn't yet large enough to merit its own CDN, you can ease a future transition by serving your static media off a separate subdomain (e.g. static.example.com) using a lightweight HTTP server like [Nginx](http://nginx.org/), and cutover the DNS from your servers to a CDN at a later date.

Cache invalidation

While caching is fantastic, it does require you to maintain consistency between your caches and the source of truth (i.e. your database), at risk of truly bizarre applicaiton behavior.

Solving this problem is known as *cache invalidation*.

If you're dealing with a single datacenter, it tends to be a straightforward problem, but it's easy to introduce errors if you have multiple codepaths writing to your database and cache (which is almost always going to happen if you don't go into writing the application with a caching strategy already in mind). At a high level, the solution is: each time a value changes, write the new value into the cache (this is called a *write-through* cache) or simply delete the current value from the cache and allow a read-through cache to populate it later (choosing between read and write through caches depends on your application's details, but generally I prefer write-through caches as they reduce likelihood of a stampede on your backend database).

Invalidation becomes meaningfully more challenging for scenarios involving fuzzy queries (e.g if you are trying to add application level caching in-front of a full-text search engine like [SOLR](http://lucene.apache.org/solr/)), or modifications to unknown number of elements (e.g. deleting all objects created more than a week ago).

In those scenarios you have to consider relying fully on database caching, adding aggressive expirations to the cached data, or reworking your application's logic to avoid the issue (e.g. instead of DELETE FROM a WHERE..., retrieve all the items which match the criteria, invalidate the corresponding cache rows and then delete the rows by their primary key explicitly).

Off-line processing

As a system grows more complex, it is almost always necessary to perform processing which can't be performed in-line with a client's request either because it is creates unacceptable latency (e.g. you want to want to propagate a user's action across a social graph) or it because it needs to occur periodically (e.g. want to create daily rollups of analytics).

Message queues

For processing you'd like to perform inline with a request but is too slow, the easiest solution is to create a message queue (for example, [RabbitMQ](http://www.rabbitmq.com/)). Message queues allow your web applications to quickly publish messages to the queue, and have other consumers processes perform the processing outside the scope and timeline of the client request.

Dividing work between off-line work handled by a consumer and in-line work done by the web application depends entirely on the interface you are exposing to your users. Generally you'll either:

1. perform almost no work in the consumer (merely scheduling a task) and inform your user that the task will occur offline, usually with a polling mechanism to update the interface once the task is complete (for example, provisioning a new VM on Slicehost follows this pattern), or
2. perform enough work in-line to make it appear to the user that the task has completed, and tie up hanging ends afterwards (posting a message on Twitter or Facebook likely follow this pattern by updating the tweet/message in your timeline but updating your followers' timelines out of band; it's simple isn't feasible to update all the followers for a [Scobleizer](http://twitter.com/#!/scobleizer) in real-time).

Message queues have another benefit, which is that they allow you to create a separate machine pool for performing off-line processing rather than burdening your web application servers. This allows you to target increases in resources to your current performance or throughput bottleneck rather than uniformly increasing resources across the bottleneck and non-bottleneck systems.

Scheduling periodic tasks

Almost all large systems require daily or hourly tasks, but unfortunately this seems to still be a problem waiting for a widely accepted solution which easily supports redundancy. In the meantime you're probably still stuck with [cron](http://en.wikipedia.org/wiki/Cron), but you could use the cronjobs to publish messages to a consumer, which would mean that the cron machine is only responsible for scheduling rather than needing to perform all the processing.

Does anyone know of recognized tools which solve this problem? I've seen many homebrew systems, but nothing clean and reusable. Sure, you can store the cronjobs in a [Puppet](http://www.puppetlabs.com/) config for a machine, which makes recovering from losing that machine easy, but it would still require a manual recovery, which is likely acceptable but not perfect.

Map-reduce

If your large scale application is dealing with a large quantity of data, at some point you're likely to add support for [map-reduce](http://en.wikipedia.org/wiki/MapReduce), probably using [Hadoop](http://hadoop.apache.org/), and maybe [Hive](http://hive.apache.org/) or [HBase](http://hbase.apache.org/).

Adding a map-reduce layer makes it possible to perform data and/or processing intensive operations in a reasonable amount of time. You might use it for calculating suggested users in a social graph, or for generating analytics reports.

For sufficiently small systems you can often get away with adhoc queries on a SQL database, but that approach may not scale up trivially once the quantity of data stored or write-load requires sharding your database, and will usually require dedicated slaves for the purpose of performing these queries (at which point, maybe you'd rather use a system designed for analyzing large quantities of data, rather than fighting your database).

Platform layer

Most applications start out with a web application communicating directly with a database. This approach tends to be sufficient for most applications, but there are some compelling reasons for adding a platform layer, such that your web applications communicate with a platform layer which in turn communicates with your databases\

First, separating the platform and web application allow you to scale the pieces independently. If you add a new API, you can add platform servers without adding unnecessary capacity for your web application tier. (Generally, specializing your servers' role opens up an additional level of configuration optimization which isn't available for general purpose machines; your database machine will usually have a high I/O load and will benefit from a solid-state drive, but your well-configured application server probably isn't reading from disk at all during normal operation, but might benefit from more CPU.)

Second, adding a platform layer can be a way to reuse your infrastructure for multiple products or interfaces (a web application, an API, an iPhone app, etc) without writing too much redundant boilerplate code for dealing with caches, databases, etc.

Third, a sometimes underappreciated aspect of platform layers is that they make it easier to scale an organization. At their best, a platform exposes a crisp product-agnostic interface which masks implementation details. If done well, this allows multiple independent teams to develop utilizing the platform's capabilities, as well as another team implementing/optimizing the platform itself.

I had intended to go into moderate detail on handling multiple data-centers, but that topic truly deserves its own post, so I'll only mention that cache invalidation and data replication/consistency become rather interesting problems at that stage.

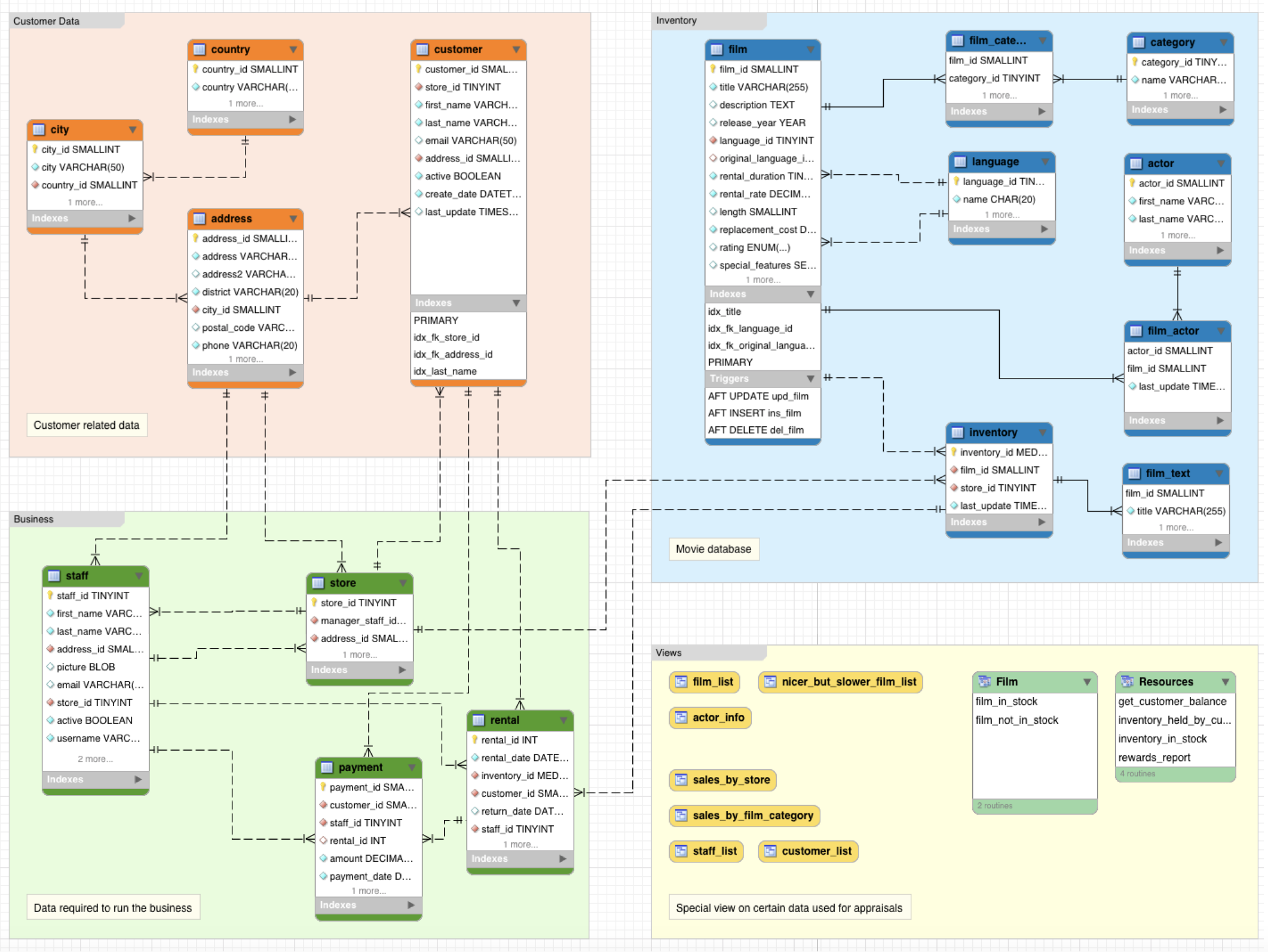
**EMG 549 Software Development and Architecture Homework 1**

**Question 0. Working with the Sakila MySQL database**

In this exercise, we will be working with the Sakila MySQL database. It can be downloaded by following the instructions at [http://dev.mysql.com/doc/sakila/en/sakilaD installation.html](http://dev.mysql.com/doc/sakila/en/sakilaD%20installation.html)

The database Files are downloaded at [https://www.dropbox.com/s/803itqyqzw8w174/ sakilaDdb.zip?dl=0](https://www.dropbox.com/s/803itqyqzw8w174/%20sakilaDdb.zip?dl=0)

Download the Files and follow the instructions completely.

Examine the structure of the tables by looking at MySQL workbench or the included PDF File. Following image is enlarged at [https://www.dropbox.com/s/guc1zu70u0dn7bt/ Untitled 2.png?dl=0](https://www.dropbox.com/s/guc1zu70u0dn7bt/%20Untitled%202.png?dl=0)

**Give SQL queries to answer the following questions and Run these queries print answers:**

1. How many distinct countries are there?

2. Find out the top 5 countries with most number of clients.

3. What are the names of all the languages in the database (sorted alphabetically)?

4. Return the full names (First and last) of actors with “SON” in their last name, ordered by their First name. Hint: To search for a text in a Field use the Filter: WHERE First\_name LIKE ‘%son%’ ; This will search for “SON” anywhere in the Field.

5. Create a list of Films and their corresponding categories.

6. Create a list of categories and the number of Films for each category.

7. Create a list of actors and the number of movies by each actor.

8. List the Film id and titles of those Films that are not in inventory.

9. Find a list of customers who have not rented a movie yet.

10. Find the number of **English** Films having category of ‘**Documentary’**.

**Question 1. Read the following Case Study on Intro to System Architecture and Answer following questions:**

1. What is web server? Give example name of a web server? Why is it used?

Web server can refer to either hardware or software or both that stores and delivers content for a website such as text, image, video and application data. An example for a web server is Node.js. It’s a server side JavaScript environment for network applications. It is used to create a server that listens to server ports and gives a response back to the client.

1. What is message queuing system? Where is it used? What are the features of Messaging system?

It’s a system that provides a temporary message storage when the destination program is busy or not connected. Applications communicate by sending messages to each other, for example to tells one system to start a task, etc. The basic features of messaging system is to receive messages that are created by producers and deliver them to the message queue. There, the message is queued and stored until the consumer (another application) collects them.

1. What is load balancer? Difference between Hardware and Software load balancer?

Load balancer is a device that distributes network traffic across a number of servers to ensure no single server bears too much load. Hardware load balancer are deployed between servers and clients and implemented on transport and application layer. While software load balancer are implemented on virtual machines that consumes the processor and memory of the servers.

1. What is caching? Different types of caching? Why is caching needed? What happens when caching is not present?

Caching is a high-speed data storage layer that stores data so that future request related to that data can be served up faster. It’s used to effeciently retrieve previously computed data which will improve the overall performance of the application, precalculating results, and pre-generating expensive indexes. There are few different types of caching such as browser cache, server-side cache, database cache, and object cache. When caching is not present, then there will be no cache saved that might impact performance and increase load on the server.

1. What is platform as service , infrastructure as service and software as service?

**PaaS:** Vendor provides platform for developer to develop application such as hardware and software tools over the internet.

**IaaS:** Vendor provides pay as you go storage, networking and virtualization. The nature of it are flixible and scalable.

**SaaS:** No longer to engage with an IT specialist as the software can be accessible from any device, anytime as long as there is an internet connection. Everything else is taken care by the vendor.

1. What is CDN? Please explain in detail. Where is CDN used in apps you use on daily basis on your smart phone or TV or Computer?

Content Distribution Network is a layer in the internet ecosystem that geographically distributed network of servers and data centers to provide high availability and performance. CDN make sure content is delivered faster to internet users. An example is Banking Application, having CDN provide a fast, secure and reliable infrastructure to deliver sensitive data to consumers.

1. What is Map Reduce? What is distributed computing? Explain data gravity?

**MapReduce:** programming model or pattern within Hadoop to access big data stored in Hadoop File System.

**Distributed computing**: a model where components of a software system are shared among multiple computers to improve efficiency and performance based on geographic area.

**Data gravity:** a relationship between data and application. When data are getting larger, it’s harder to manage and attracted other things like application and processing power to it.

**Question 2. Read the following case study on designing News Feed System like Facebook and answer following questions:**

1. What is news feed system and why is it important business like Facebook?

It’s an application that allows a user to post content that can be subscribed by others. News feed system is important to Facebook as that is the feature that generates user base for facebook.

1. Please describe your understanding of Redis and Cassandra distributed database systems. Explain purpose and salient features of each database system. Explain creative use of these system in Newsfeed system

**Redis:** a key-value store that allows each user to have a dedicated individual message box or more instead of one message box for all users that is stored in-memory datasets. This allows Redis to be more versatile, such as adding another massage boxes for a user so they are able to get different logic.

**Cassandra**: is a key value store that can handle data across machines and data that will not fit into memory. It has a bigger capability in scaling the application than Redis.

1. Describe traditional approach and message box based approach to design News feed system like Facebook.

**Traditional:** to create a content management system where a user creates a post that goes into the database. With a feed update, a database query will fetch the list of all subscriptions and show the latest content sorted by publication date.

**Message Box:** When a content is published by a user, it should broadcast it to all subscribed user. Message box approach is by having a message box for all user so that when a post is created, it goes to the subscribed user message box. Then the post are fetched from the message box and show to the user.

1. How do we scale news feed system with Cassandra.

Because it allows multiple hardware as required. It has no single point of failure so it’s continuously available with reliable performance. Cassandra supports replication acroess multiple geographic locations and provides lower latency for users.

**Question 3: Read following article on Google Architecture and answer following questions.**

1. After reading following article please describe how Google is different and why google is king of scalability

Google is able to scale well because they have a reliable scalable storage which they build theirselves to ensure control. Then using MapReduce to process and generate large data sets, they are able to partition tasks across lots of machine, handle machine failure and make it works across different application types. Their data are stored in BigTable that is able to handle millions of reads per second.

1. Summarize Google File System and how is it different from Windows or Linux file system

It’s a build in storage platform with large distributed log structured file system that is capable in handling huge amount of data. It allows google to control everything, unlike Windows or Linux file system.

1. What is Map Reduce w.r.t. Google Architecture. What is Google Big Table?

MapReduce is a programming model and an associated implementation for processing and generating large data sets. It talks to Google File System in coordinating scheduling, failure handling and data transporting automatically.

Google Big Table is a self managing system that has terabytes of memory and petabytes of storage. It is a distributed hash mechanism built on top of Google File System.

1. Please write your perspective and most effective lessons learned from Google Architecture.

To build a reliable and scalable infrastructure, build it on our own. However, if it’s not align with the company’s goal, take a look at Hadoop, which is an open-sourced alternatives.

**Question 4. Answer following questions after reading “Designing Instagram” Article:**

1. Explain Instagram Architecture and draw stack diagram. Please refer to techstacks.io (<https://techstacks.io/stacks/instagram/)> an explain different technology stack components utilized and salient features and importance of each component of the stack.
   * React : JSX.

Components.

One-way Data Binding.

Virtual DOM.

Simplicity.

Performance.

* + NGINX: Reverse proxy with caching.

IPv6.

Load balancing.

FastCGI support with caching.

WebSockets.

Handling of static files, index files, and auto-indexing.

TLS/SSL with SNI.

* + Phyton: Easy to Learn and Use.

Expressive Language.

Interpreted Language.

Cross-platform Language.

Free and Open Source.

Object-Oriented Language.

Extensible.

Large Standard Library.

* + Gearman:
  + Apache Solr: Full text search

Hit highlighting

Faceted search

Real-time indexing

Dynamic Clustering

Database Integration

* + Gunicron: Natively supports WSGI, web2py, Django and Paster.

Automatic worker process management.

Simple Python configuration.

Multiple worker configurations.

Various server hooks for extensibility.

Compatible with Python 2.6+ and Python 3.2+

* + PostgreSQL: Compatible with various platforms using all major languages and middleware.

It offers a most sophisticated locking mechanism.

Support for multi-version concurrency control.

Mature Server-Side Programming Functionality.

Compliant with the ANSI SQL standard.

* + Redis: Transactions

Pub/Sub

Lua scripting

Keys with a limited time-to-live

LRU eviction of keys

Automatic failover

* + AWS: Mobile friendly access

Serverless cloud function

Completely managed databases

Flexible and easy to use storage

Secure and compliance

AWS Marketplace for add ons

* + Ubuntu: Hardware autoconfiguration

Multiple desktop

Software repositories

Ssh client

No antivirus

A screenshot of a cell phone

Description automatically generated

1. Explain briefly capacity estimation for Instagram

If there’s 1 Billion total users, with 500M daily active users.

100M new photos every day, 70 new photos every second.

Average photo file size => 500KB

Total space required for 1 day of photos

100M \* 500KB => 50 TB

Total space required for 10 years:

50TB \* 365 (days a year) \* 10 (years) ~= 182,5 Petabyte

1. What is data sharding w.r.t. Instagram

Data sharding is a term that is used to make a very large database more managable by breaking it up to a many smaller databases that share nothing and can be spread across multiple servers.

1. Explain Ranking of Newsfeed and newsfeed generation methods associated with it

To create a ranking of newsfeed, Instagram application server will first get a list of people the user follows and then fetch metadata info of latest 100 photos from each user. In the final step, the server will submit all these photos to our ranking algorithm which will determine the top 100 photos (based on recency, likeness, etc.) and return them to the user. A possible problem with this approach would be higher latency as we have to query multiple tables and perform sorting/merging/ranking on the results. To improve the efficiency, it can pre-generate the News Feed and store it in a separate table.

1. Explain Load Balancing and Caching @ Instagram. Why is it important? (use black Friday deals for example. Caching is to put addictive habit in the program. Read book: habit)

**Load Balancing**: Used to ensure performace and loadtime of the application. Every request to Instagram servers goes through load balancing machines; we moved to using Amazon’s Elastic Load Balancer, with 3 NGINX instances behind it that can be swapped in and out (and are automatically taken out of rotation if they fail a health check). We use Amazon’s Route53 for DNS, which they’ve recently added a pretty good GUI tool for in the AWS console.

**Caching**: Used to lessen the load on server by saving data locally on user side. Instagram use Memcached for caching, and currently have 6 Memcached instances, which we connect to using pylibmc & libmemcached. Amazon has an Elastic Cache service they’ve recently launched, but it’s not any cheaper than running our instances, so we haven’t pushed ourselves to switch quite yet.

**Question 5. Read the following article on Different type of Software Architecture and summarize each architecture in your own words. Give few examples of products/apps using following each architecture in use in industry today**.

1. Two Tier: An architecture where the interface runs on a client and a data layer gets stored in server. Example: Contact management system
2. Domain Driven: Object-oriented analysis that focuses on the plan to solve business problems. With concepts of entities, value objects, domain modeling, ubiquitous language, bounded context and anti-corruption layer, it’s creating a more requirement-oriented software.
3. Service Oriented: Collection of services that communicate with each other and can be reusable. The nature is that it’s modular and provide a plug-and-play environment.
4. Three Tier: An architecture where the user interface, business logic, and computer data are stored and managed as separate platforms. Example: Guru99.com
5. Microservice Architecture: This architecture is a collection of small, self-contained, autonomous services that sums up the business capability. These independent services allow developers to develop, test, deploy, scale and switch on single service. Example: Yelp.
6. Onion: Layers of application consists of UI, Business, Layer and Data Access Layer depend tightly on each other.The basic idea of onion architecture is so that inner layers of the application shouldn’t depend on the outer layer, but the outer layer can depend on inner layer. Example: Shein
7. N-Tier: Multi layer architecture that process, manage and present data both physically and logically separated. Different functions are hosted on several machines to ensure services are provided on separate resources. Example: Amazon.
8. Aspect Oriented: Architecture that focuses on creating a modular software that helps them keep their code maintanable and reusable.
9. Event based: This architecture has a producer that creates an event and pushes the events to be queued up for processing. Once it’s completed, it will be forwarded to the consumers. Example: Google Pay

