**Part 1**

While designing systems, there are three primary concerns that should be addressed — reliability, scalability and maintainability. These terms are tossed around quite frequently and, in this post,, I want to provide expositions for each of them.

**Reliability**

Reliability means the ability of a system to tolerate faults or problems to prevent failures or complete shutdowns. Large systems are built using fault intolerant components. The beauty and art of system design is to build fault tolerant systems using fault intolerant components.

Faults can be categorized as hardware or software. For example, a large data center, with hard disks with MTTF of 50–100 years, will witness disks going bust every day. Memory will be corrupted on a regular basis. Hardware faults can be addressed by adding redundancies. For example, disks can be stored in RAID configuration, data centers can have multiple power backups, CPUs can have [hot-swapping](https://en.wikipedia.org/wiki/Hot_swapping) support.

Software faults can happen due to a variety of reasons. One runaway process can hog your system resources and cause a systematic crash across all nodes or the operating assumptions of the applications can change and result in crashes. Software faults can be handled by understanding business requirements and building resiliency to handle deviations from the same, better monitoring to publish warnings early on, better unit testing and finally by designing better abstractions and interfaces to easily isolate problems.

**Scalability**

Scalability is the system’s ability to deliver reasonable performance in face of increased load. System load can be described in terms of parameters that best translate an application’s raison d’être technically. For example, for a social networking website, the expected number of the writes (posts) per second or reads (posts in timeline view) per second can be used to describe load. You can also consider peak reads/writes instead of average, for describing load on the system.

Performance can be thought of as the system’s operating characteristic when system’s load parameter is changed. For example, you might measure performance in terms of system average response time. You can also measure performance in terms of distribution of response times. So, you might consider the 99th percentile of response time to be under 1 sec and average response time to be 300 Ms. Performance metrics quite often are a part of your SLA with customers. There are many ways to achieve system scalability which I plan to cover in the subsequent posts in this series.

**Maintainability**

Maintainability means writing code that can easily be understood, refactored and upgraded by someone who is not the original author of the code. Any piece of spaghetti confusing code will ultimately be understood by machines. Good code should be readable and easily understood so that teams can collaborate. Good code should also have the right level of abstractions, clean APIs and interfaces so that new functionality can be easily built on top of existing codebase.

**Part 2**

Applications need to have permanent storage for user or applications specific data. In memory data structures like linked list, arrays are optimized for access by CPU via pointers. Permanent storage is optimized for read/write access by clients/processes connecting to database server. A very important aspect of permanent persistence is data modelling. I will devote this post on how to choose a good data model for your application.

**Relational Database**

The most famous and prevalent data modelling technique is using relational tables. In relational tables, data is organized into records of a table. Tables are related to one another using primary key foreign key. There are a lot of reasons why you should choose relational table.

1. You are just building out v1 of your app and data access patterns are not quite clear yet. Relational schemas are always a good first default choice. v1 of all apps are generally shit (rightly told to me by a senior backend developer at my office :) ), and no one will fault you for starting with relational tables, instead of fancier NoSQL or Graph DB.
2. You need to enforce strict schema on write constraints.
3. You want to maintain zero data redundancy. Normalization of schema in relational models has the effect of shredding information into many tables
4. Your data model has many to one and many to many relationships. In other words you know which joins will be performed beforehand. Joining and querying relational databases using a declarative language like SQL is one of the greatest secret sauce of relational databases. A lot of research and effort has gone into making relational queries super-fast. An application developer just must specify the expected data pattern of the query. The query engine will convert the SQL query into an optimized code to fetch/write data.

**NoSQL Database**

Application development is done using object-oriented programming. However, when data storage is done using tables, there is a translation required from objects to “shredded” relational tables. ORM frameworks provide boilerplate code to reduce the effort required for this translation, but still there is work to be done.

NoSQL solves this problem by representing a record as a self-contained JSON document. For example, let’s say we need to store patient demographic information along with his current conditions. One way to represent this record in a NoSQL database can be:

{ “first\_name”: “John”, “last\_name”: “Doe”, “conditions”:[ {“name” : “T2DM”, “onset”: “12–12–1990”}]}

If you need read profile data for a patient in your app, you need not issue multiple joins as all data is inside one NoSQL document. Typically, if your data model exhibits a tree like, one to many relationships, using a NoSQL database might make more sense.

In the patient example above, what if we wanted to store ICD10 standard codes for conditions instead of name of condition. This is a little troublesome in document-based NoSQL databases as they have little support for joins. You can still do a join at the application layer, but this will be always suboptimal compared to the joins done at a typical relational database layer. NoSQL databases become less desirable in this case.

Lastly in case your data model does not have a fixed schema, going the NoSQL route might make more sense. Consider, the patient example above and say we need to also store patient’s date of birth. In NoSQL case, we can add a new field, ‘dob’ to new documents. At the application level we can also add code to handle reading old documents without dob field. In a relational database, the solution to handle dob would be to alter schema and make data migrations. Data migrations are slow and require downtime and consequently generally avoided.

**Graph Database**

Graph database makes a lot of sense when your application’s data model needs to support many ‘many-many’ relationships. The relational model can handle a few many to many relationships, but beyond a point all the relational joins become messy and slow. Using graph databases also provides an added advantage of easily extending relationships between heterogeneous objects.

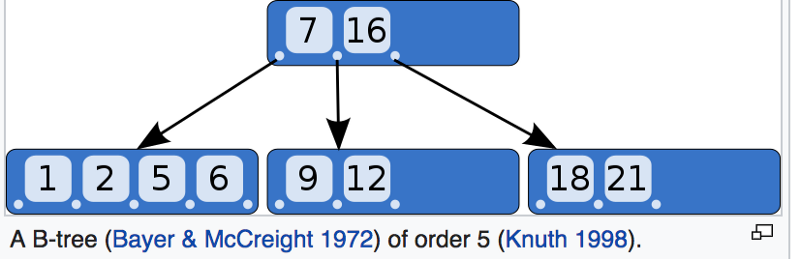
A graph consists of two kinds of objects — nodes and edges. Nodes contain description of objects or entities. Edges contain description of relationships between nodes. For example, say a person suffers from an allergic reaction because of exposure to a substance. You can model person, allergic reaction and substance as nodes. You can also model relationships between nodes in a graph database. For example, a relationship between person and allergy can be a unidirectional relationship “person-has-allergy” from person node to allergy node. Relationship between allergy and substance can be a unidirectional relationship “triggered-by-exposure-to” from allergy to substance.

Why not do this in relational tables? Well you can. You can create three tables — person, allergy and substance and set up the appropriate PK-FK constraints. Like I said earlier, using graph databases makes sense when we have a lot of many-many relationships. For example, let’s say you introduce a location object in the overall scheme of things so that you can capture location of the person where the allergy reaction occurred. Location can be neighborhood, city, state, country, continent or hemisphere. Basically, location information can be available at various levels of granularity. Using SQL to create a declarative query will be messy. SQL needs to know in advance which joins will be part of the query. In graph database, on the other hand, you can traverse many nodes and edges before arriving at the target node. You can express the fact of traversing a graph once or many time quite concisely using a graph database declarative query language like Cypher, for Neo4j graph database.

**Part 3**

All databases implement indexes. Indexes are additional abstraction implemented in databases to support high read throughput. So why not index everything in database? Because indexes are additional abstractions and make writes slower. Hence it is the role of application developer, to define which columns/keys require indexing based on the read/write workloads of the application.

The most prevalent way of indexing is using B-trees. B-trees divvy up the database in pages or blocks of few Kbs. Each page can be located by its unique address on disk. This way pages can reference each other.



For example, in the diagram above, if you are looking for say key 10, start by looking at the root node. 10 lies between 7 and 16, so follow the pointer between 7 and 16 and you will land at the middle node in second row. 10 lies between 9 and 12 so follow the pointer the next page referenced by the on-disk pointer between 9 and 12. Keep on searching till you reach at the leaf node containing the key and its corresponding value. The value will typically be the byte offset of the location where the record is stored.

Another indexing technique which is used in newer databases like Elastic Search, Hbase, Cassandra, Riak is based on the [BigTable paper by Google](https://static.googleusercontent.com/media/research.google.com/en/archive/bigtable-osdi06.pdf). Here is the short summary of how it works under the hood:

1. A new write is added to an in-memory sorted balanced tree (like a red black tree or an AVL tree). This in-memory tree is known as memtable.
2. When the memtable becomes big, it is flushed to the disk as an SSTable file. Think of SSTable as a sorted key-value storage on disk of the in-memory tree i.e. memtable which is already sorted. While the SSTable is being written on disk, new write can continue in memtable.
3. Reads are first directed towards memtable. If the keys are not found in memtable, it’s searched in the most recent SSTable and then the next most recent SSTable and so on and so forth. SSTables are sorted so it is easy to do range queries on them.
4. In the background, duplicate key is removed from SSTable and the most recent key value is retained. This process is known as compaction.
5. Compacted SSTable are merged into new SSTable. Since SSTables are sorted, merging them using a merge sort like algorithm is quite fast.

This indexing scheme is known LSM Tree or Log Structured Merge Tree.

As a thumb rule, LSM trees can handle higher write workloads and B-trees are good for high read workload. This is because writes in SSTables are always sequential unlike B trees, where random writes take place (B-tree pages need not be sequentially arranged on disk).

In order to bake durability, writes in B-trees are also preceded with writes to an additional file known as Write Ahead Logs (WAL). WALs are append only files, that help to restore B-tree to a consistent state in case of a crash. This implies that writing to a B-tree means, first writing to WAL, which in turn means additional work and slower writes.

Typically, the merge and compaction process in an LSM tree is very fast but sometimes it can lag the writes. This can happen when database is experiencing very high written e workload. Slow compaction and merging adversely affects reads as many more SSTables need to be read now. This is a big disadvantage of LSM trees that is in very high write throughputs scenarios, its performance can get unstable, unlike that of B-trees which exudes generally stable performance.

Finally, if transactional semantics are of paramount importance, then B-trees are preferable. In LSM trees, same key can be present in multiple SSTable. In B-tree, one key is present at only place and it’s value is updated in-place. As a result, transactional isolation is easily achieved in Btrees.

**Part 4**

Modern applications can handle thousands of transactions per minute. One of the things that you will do to achieve scalability, is to do replication of your database. There are several reasons for doing so:

1. You want to keep data geographically close to users
2. Balance read/write across multiple machines to prevent throttling of a single database server
3. Build redundancies within your system

There are a lot of interesting subtleties involved in replications that are concerned with durability guarantees and eventual consistencies. As a product person or application developer, it’s very important to cut through the vagueness of jargons and understand how replication works.

A copy of your data on another machine or node is known as a replica. There are three ways to achieve replication of data.

1. Single master slave replication: All writes go to one node, called master. The changes in master node are replicated to other nodes, called slaves. Read requests can go to either master or slave.
2. Many master replication: Instead of writes going to one master, they can go to many masters. Masters can then replicate changes made to their data stores to other slave’s nodes. Read requests can go to either master or slave.
3. No master replication: There are no master and slaves in this setup. Writes and reads can be sent to any node.

I will cover single master slave replication in some detail before talking about the other configurations.

An important detail worth mentioning is the way replication happens. It can either be synchronous or asynchronous.

Synchronous replication means that master sends its writes to all slaves and waits for write confirmation from ‘all’ slaves. Once the master receives write confirmations, it then returns a successful write message to client. After all this, the writes are made visible to other clients. This mode provides the highest durability guarantee. Every write will be propagated to all nodes and no client connecting to any of the slaves will ever see stale data. If these guarantees make sense for your application, you should choose synchronous replication.

The biggest drawback with synchronous replication is that it is very slow. If the network connectivity to some nodes is choppy, it can take forever to make one successful write. In practice, people always invariably choose ‘asynchronous’ replication. In asynchronous replication, a master confirms write to the client, after a successful write on master node. It also sends changes to other slave replicas but does not wait for a successful write confirmation from other slaves. This effectively means that durability guarantees are weakened as some of the slave nodes update their data stores later and in the meantime a client connecting to one of the slave nodes will be presented with old data. However, using asynchronous replication makes up for this by providing significant performance gains. As a result, asynchronous replication is the dominant replication strategy.

Most applications have read/write ratio skewed heavily in favor of reads i.e. writes will be fewer. In order to take advantage of this fact, reads are handled by slaves. In an asynchronous master slave configuration, this presents a problem because of replication lags. Let me go through a few examples of problems due to replication lags and steps to save them.

1. Reading writes: User submits a write on master and then decides to read his/her write. The read request can be serviced by any slave and the slave servicing read request might not have the most recent write because of replication lag. This can clearly cause user frustration, not able to see his/her most recent post. You can resolve this by ensuring all reads go to master after 1 minutes of any write. Another approach can be client remembers the last timestamp of the write made and then uses this information to read only from those replicas which have a more recent timestamp than the client’s timestamp
2. Moving backwards in time: User submits a write that gets replicated on one of the slaves, s1 and not on the others due to replication delays. User then reads from slave s1 and sees the write made by him/her earlier. However, he/she reads again, and this time say the read is handled by a slave which is not s1. User can be quite frustrated to watch his/her writes disappearing. Such problems can be resolved by handling reads for one client only from one replica.

There can be other subtle but operationally annoying issues, especially if you are operating in a choppy internet or full capacity. It’s important to consider replication lags in your overall system design for replicated data stores.

Now on to many master configurations. The most common use case for this configuration is when you have multiple data centers. Each data center can have one master and rest of the nodes can be slaves.

The biggest advantage of many master configuration is performance. Your writes are distributed and no longer throttled by the capacity of single master. However, there are no free lunches in this world. What you have gained in performance is made up by handling additional complexity in case of concurrent writes.

In case of many master configuration, you can tie users editing their own data, to one master. Essentially from a user’s perspective, the configuration becomes single master. This is neat as now you can avoid all merge conflicts due to concurrent writes on many masters. However, if you cannot do this, then you again need to resolve conflicts by using one of the following strategies:

1. Make the most recent write win
2. Write custom code on detecting conflicts in replication logs
3. Present the user with a list of values, and let the user decide the merge and discard strategy in case of a conflict

Clearly the right strategy is dependent on the nature of your application and user expectations.

Finally let me introduce no master replication. In this setup, every node can accept both read and write request. You can immediately see a problem with this approach. Not only your reads can be stale because of replication lags, your writes can also be inconsistent.

To solve this problem reads and writes are sent in parallel to many nodes. Let’s say r reads are made in parallel, w writes are made in parallel and there is a total of n nodes. As long as r + w > n, you can be sure that at least one of the r nodes must be up to date and consequently reads will not be stale.

**Part 5**

Partitioning or sharding is the technique of breaking down a large dataset and distributing it across many disks. Reads/writes, after partitioning can be parallelized across many nodes. Partitioning is combined with replication to achieve scalability.

One easy way to achieve partitioning is to divvy up the entire dataset based on some fixed criteria. For example, for an e-commerce website, you can put all transactions for mobile phones on node 1, shoes on node 2 and so on. The problem with this approach is that this can result in skewed distribution. If mobile phones are the most popular items, the node handling mobile phone transactions will always be overloaded. One easy workaround is to assign partitions randomly. This ensures all your data gets spread evenly across nodes, but you must have already guessed the problem with this approach. When you issue a read query, it needs to be sent to all the nodes, making reads costlier.

Another way to handle this problem is to design key based range partitions. You can first create a hash of key and then assign a range of resultant hash values to certain partitions. If you use a 32 bit hash function, your keys can be mapped to one of 2^32 – 1 hash values. You can then choose to assign keys 1–10 to say partition 1 and so on. The partition boundaries can be chosen randomly or manually.

While hash-based partition, ranges help in reducing asymmetric loads, there are no easy ways, at least at database level, to completely solve this problem. If for the said e-commerce website, iPhone X is the highest grossing product for one month, node partitions hosting iPhone X transactions will be stressed. The application developer in this case can add a 2-digit random key to original key hash and consequently distribute load for one key across 100 partitions. The challenge will again be reading as 100 parallel reads now need to be fired to find one transaction for iPhone X. Hence it makes sense to add random digits to only a few number of “hot” keys.