How many persistent connections can a machine hold:

- Depending on the workload 100k to 1M

System Design Artefacts

- Consistent hashing for caching, cache db memcached, redis
- In CAP the world is converging to availability and partition resistant over consistency.
- The client developer is an important part of the system, they need to be aware of what the system is offering. Consistency in ACID is slightly different, relates to unique, primary key guarantees. In NoSql Consistency is consistency between replicas.
- Database scaling, availability, replication, geographical closeness, durability
    - Leaderless Tunable consistency quorum consistency, sync replication to some nodes. Cassandra and dynamodb
    - Single leader - linearized using consensum, Strong consistency zookeeper, etcD, consensus problem, leader replication etcd raft paxos. Also writes are nearly ‘atomic’ across all the nodes.
    - Single leader - synchronous or asynchronous replication, potentially linearizable if reads are also from leader or synchronous followers.
    - Multi leader asynchronous replication, conflict resolution required, waitforreplication may still cause conflicts. Durability is an issue if node fails before replicating to other nodes. Lower write latency. Ex couchdb
    - https://www.brianstorti.com/replication/
- Replication
    - Reduce read latency but may not reduce response time as the query may be inefficient
    - Depending on type of replication write latency may increase
    - Sandra uses semi-synchronous replication (grid cluster vs ring and waitForReplication)
    - Issues with async replication or eventual consistency
        - Read your own writes - simple usually at data store level
            - Always read from leader
            - Use commit token (token of the last write), send token when reading and nodes will decide if its updated
        - Non-monotonic reads - read 1,2,3 and another node read 1,2
            - Always read from same replica
    - Quorum based consistency are strongly consistent but do not offer linearizability. Paxos, Raft and RDBS offer that.
- Partitioning and or sharding is important for scaling
    - Key mapped to partition which are in turn mapped to nodes
        - Indirection helps with load balancing and hot keys
        - Nodes can be split when full, partition are remapped to nodes
    - Several ways to query, usually the mapping is saved in intermediary in zookeeper like server to route read request.

        - Cassandra how nodes are selected for read and write (without zookeeper)
            - https://stackoverflow.com/questions/31669991/how-does-cassandra-find-the-node-that-contains-the-data
- Conflict Resolution Techniques
    - Last Write Wins (clock skew)
    - Application resolution
    - Conflict handler
    - Return all values
- API Gateway
    - Load balancing
    - Rate limiters
    - SSL termination
    - Security, DDoS
- Configuration consistency: Zookeeper, etcD
- Object storage, Amazon S3
- Notification service
- Key generator

Issues with sharding?

- Hot keys, increased data requires rebalancing
- Difficult to manage relational data
- Range queries requires visiting all nodes
-

DynamoDb

(Riak is based on the same principles as dynamo but open source)

- Primary key access only interface
- Data is partitioned, replicated using consistent hashing to N-1 nodes clockwise. Meaning, let's say DB has 100 nodes with a replication factor of N. Using consistent hashing, the node will be mapped to a co-ordinator node which will replicate to the next N-1 nodes represented in the circle or a preference list that is associated with each node.
- B-Tree storage mechanism is used
- Consistency is facilitated by object versioning like vector clock (node, counter)
- Use quorum like technique but is not linearizable ex: n=3, r=2, w=2: C write A,B then A,C reads from B, C more details in Chapter 9
- The decentralized replica synchronization protocol Merkle tree
- Gossip based distributed failure detection and membership protocol
- Automatic node addition and removal with auto partitioning and distribution
- Optimized for reading and modifying single objects - distributed hash table
- It uses sloppy quorum with hinted handoff, it selects the first N nodes from the preference list. In strict quorum, those N nodes would be fixed in sloppy the N is the first N healthy nodes. So two consecutive writes may have different N nodes. The new node

using hint metadata transfers the data back to the preferred node once it is back up. This increases availability but could cause conflicts.
- When conflict occurs, dynamodb returns all versions of the objects.
- Clients can connect directly to the coordinator role using a client api that is aware of the database structure. Or connect to any node (with or without a load balancer) which will in turn forward to the coordinator. This increases latency due to additional hop.
- The coordinator does a read repair on read requests when stale information is received from any node.

Cassandra

- Mostly the same as Dynamodb except that it provides a Wide-Column type of architecture and uses LSM tree unlike B-Tree in dynamodb
- Operation within a row-key is atomic
- Cassandra does consistent hashing as well but also moves nodes around in the circle to manage load balancing. Dynamo uses virtual nodes and is load unaware.
- Replication is configurable to build the preferred list instead of blindly using the next N-1 nodes in the ring. Configurations are, rack aware, rack unaware, datacenter aware, unaware, etc.
- A zookeeper is used to build the preference list The preference list is stored locally in each node as well as the zookeeper. The preference list is exchanged using gossip so that requests can be routed to the appropriate coordinator node.
- Like in dynamo, the addition or removal of a node is explicit to prevent unintentional rebalance which is expensive.
- Columns act like secondary keys. For example, inbox search. Row-id is userId and Super columns are words that occur in messages of the user, and columns within a super column are the message-ids where that word was found.
- Cassandra key is two-part: partition key and clustering key. The clustering key is the sort order of keys within a node.
- Cassandra optimizes read operation by getting a full object from the fastest node (a snitch server provides that info) and getting digest (version vector, hashed value) from other quorum nodes.
- For failure detection, Cassandra uses non-boolean status which helps in managing the load or declaring the node dead.

Kafka

- White paper http://notes.stephenholiday.com/Kafka.pdf
- Combines features of distributed database and message delivery (MQ) to a new category ‘Streaming platform’. One way to think about the relationship between messaging systems, storage systems, and Kafka is the following. Messaging systems are all about propagating future messages: when you connect to the one you are waiting for new messages to arrive. Storage systems such as a filesystem or a database are all

about storing past writes: when you query or read from them you get results based on the updates you did in the past. This is why Kafka's core abstraction is a **continuous time-ordered log**.

- IMHO - Kafka provides durability aspects of databases but not other features like update, delete, transactions, etc.
- A topic is published into multiple partitions, messages are ordered only within a partition. A producer can add a 'key' to any message it publishes. Kafka guarantees that messages with the same key are written to the same partition
- Kafka maintains the offset in Zookeeper for each consumer group, per topic, per partition. Within a consumer group, each consumer gets message from a partition. If there are more consumers in consumer group than the number of partitions then the excess consumers are idle. If less, then certain consumers are assigned more than 1 partition. This means that messages are distributed by partition, i think this help in reducing the number of open sockets in each broker.
- Kafka CAP replication model is CA.
- Kafka does not use Quorum because quorum nodes increase proportionally with replicated nodes. Instead it uses ISR (in sync replica) that is stored in Zookeeper to manage recovery.
  https://www.confluent.io/blog/distributed-consensus-reloaded-apache-zookeeper-and-replication-in-kafka/
- Kafka leader detects slow replica by monitoring the number of messages a ISR is behind and crashed replica by detecting the how long ago replica made a sync request.
- MessageIds in a partition is essentially an offset from the beginning. Offset of next message is calculated as current offset + message length.
- Kafka brokers maintains in-memory offset to segment mapping, for efficiency it maintains a sorted sparse list of message id to segment id in disk. Each segment contains a continuous sequence of message ids. Unlike LSM tree where key can arrive in any order, message ids are always increasing so it is simpler to maintain mapping.
- The client API **pulls** several messages at a time but provides 1 message at a time to application. The client pulls periodically rather than being pushed by broker.
- In addition we optimize the network access for consumers. Kafka is a multi-subscriber system and a single message may be consumed multiple times by different consumer applications. A typical approach to sending bytes from a local file to a remote socket involves the following steps: (1) read data from the storage media to the page cache in an OS, (2) copy data in the page cache to an application buffer, (3) copy application buffer to another kernel buffer, (4) send the kernel buffer to the socket. This includes 4 data copying and 2 system calls. On Linux and other Unix operating systems, there exists a sendfile API [5] that can directly transfer bytes from a file channel to a socket channel. This typically avoids 2 of the copies and 1 system call introduced in steps (2) and (3). Kafka exploits the sendfile API to efficiently deliver bytes in a log segment file from a broker to a consumer.
- For writes, producers have option to fire and forget, committed to leader and committed to leader and syncd to ISR.

- Kafka writes everything to disk, even with spinning disk its throughput is very high as read and writes are very sequential due to streaming nature of the data it handles.

BigTable

- Good for storing lots of small objects. Youtube uses it to save thumbnails. Not sure why.
-

Redis

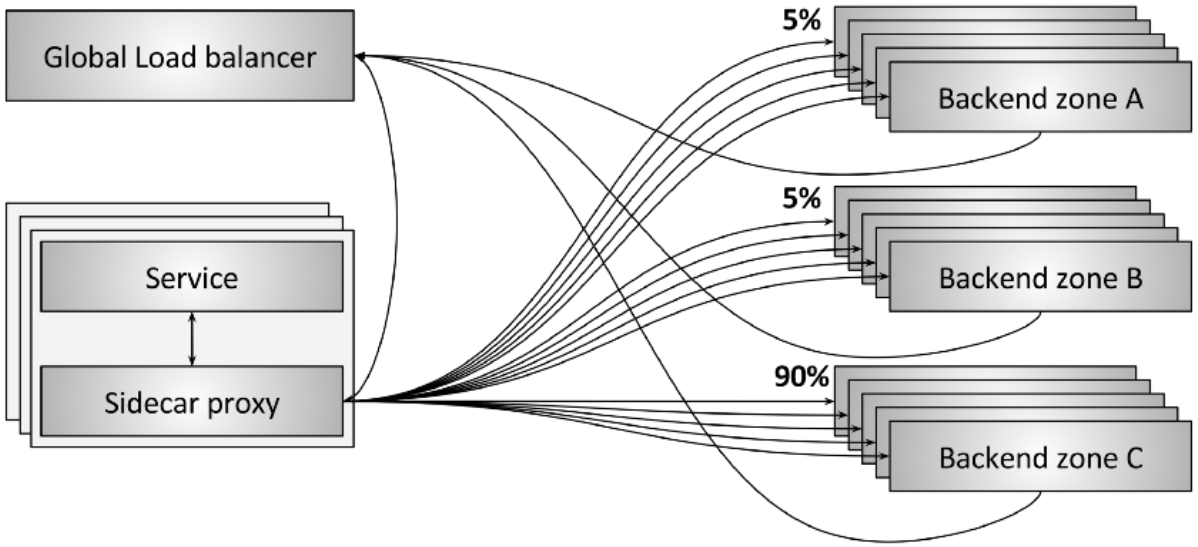
ElastiCache
MemCacche
Kafka
etcD, Zookeeper

Consistent Hashing

- To hash keys to a particular node or a bucket. Consistent hashing means that if a bucket is removed or added only *k/n* number of keys are remapped. This is an important aspect of distributed hash tables.
- One implementation of consistent hashing is to assign nodes on a circle and the keys will then belong to a node that is nearest to it in a clockwise (or anticlockwise) manner.
- An important aspect is that the mapping of keys to the server is deterministic given a configured list of servers.
- Generally, for auto-scaling, the IP addresses of servers are hashed and assigned to the circle. Hashing of IP address is not required and servers can be arbitrarily assigned to the circle but this is one popular implementation.
- Also using some deterministic logic, the servers may be assigned multiple times on the circle to counter the non-uniform distribution of the keys. Another advantage is the servers that are more powerful can be assigned more virtual spots on the circle (proportional hashing). 64 virtual nodes are popular.
- Each client keeps the hashed nodes in a binary tree so a lookup of the closest largest neighbor can be done in log(n) time.
- Domino effect, when one node dies the load is transferred to the neighboring node that may cause it to fail and so on. Note with virtual nodes and the proportional hashing domino effect is avoided as keys of the failed node will be distributed to multiple nodes.
- A couple of other alternative implementations (instead of a circle) are:
    - Google Jump algorithm, uses some sort of reservoir sampling logic and is faster than circle hashing. Works well when the range of keys a tightly grouped, like, sequential IDs.
    - Vimeo implementation to manage hot-keys, where a server might be overloaded was to find the closest server and if that server is overloaded go the next closest. To implement that load count and max load count also need to be tracked.
- Server configuration is generally available through configuration services such as zookeeper.

Load Balancing

- https://blog.envoyproxy.io/introduction-to-modern-network-load-balancing-and-proxying-a57f6ff80236
- For blocking/non-blocking/asynchronous Unix IO model https://notes.shichao.io/unp/ch6/
- Load balancing can be done at OSI L4 (network layer) or OSI L7 (application layer). At the network layer, the balancer simply uses NAT translation to route packets indiscriminately. It was used earlier when CPU and memory were expensive. These days all routing happens at the application layer where the balancer inspects the entire packet. Generally, network layer balancing was done by specialized hardware (cisco), and software wasn't required. L4/L7 combination still very popular today.
- Load balancers also function as reverse proxies as they hide the real IP of the application server. Pedantically, not all load balancers are reverse proxy as you can have a load balancer on the client-side as a library.
- Load balancers can also serve as service discovery.
- Side note: HTTP/2 supports TCP multiplexing, that is, open a single TCP connection and fire off multiple non-blocking requests. HTTP/1 doesn't multiplex and as a result, opens multiple connections. Point being that, in L4 load balancing won't behave well with HTTP2 request as it direct all requests on that connect connection to a single server. So if there are dedicated servers to fetch images/videos L4 will fail miserably. Whereas L7 load balancers 'knows' the type of request being that it sits at the application layer. Continuation of side node, websockets and SSE (server send events) are also bidirectional multiplexing protocols. SSE are primarily used to real time events like stock ticker prices.
- SSL termination, rate limiters, network stats, security, and DDoS. These features are called 'API Gateway'.
- Different load balancer topology:
    - Edge/middle where load balancers sit in the middle of client and server. This is the most typical implementation.
    - Client library implementation, generally only used in internal applications, must also provide various language binding plus all other headaches of upgrades and version control.
    - Sidecar proxy is the load balancer running as a separate process on the client-side. Can auto-update etc. Expected to be the most popular implementation in the future. There is a global loan balancer talking to all sidecar proxies directing the overall load.



    -
    - L4/L7 combined, L4 sits in the front to distribute the load over L7 and do simple network level checks such as DDOS, rate limits.
    - Edge routers talk to multiple L4 balancers and using consistent hashing spread the load to ensure high availability of load balancers themselves.
- L4 balancing can be termination or pass-thru. In termination type, the TCP connection from the client is terminated and a connection to the app server is established. In pass thru the source and destination address is replaced using NAT and can vertically scale quite well as it doesn't need to do many of the TCP functions like throttling, packet assembly, etc. As state information is present it needs to be shared with other balancers for high availability.
- L7 load balancers must understand the protocol such as HTTP, SOAP, Dynamodb, Mongodb protocols, etc.
- There are different algorithms to implement load balancing:
    - Round robin or weighted round-robin
    - Random allocation
    - Least connection - fewest active connection - most popular
    - Least response time
    - Least bandwidth - NIC usage
    - Hashed allocation
    - Power of 2 random choices - gist - when using stale or cached load information, the least busy server is immediately overloaded and then underloaded. Thereby causing oscillation in the load graph. The other approach is to choose 2 hosts randomly and then choose the less busy one. Simulations have shown this method to be effective and is thus gaining popularity, should be useful in client-side load balancing like sidecar proxy. https://brooker.co.za/blog/2012/01/17/two-random.html

Unique Id Generator

- https://www.callicoder.com/distributed-unique-ID-sequence-number-generator/

Rate Limiting and Load shedding

- Load shedding drops low priority requests even if the rate is not limited due to overall system outages or performance
- Absolute rate-limiting - up to 20 requests per second
- Concurrent rate limiters - up to 20 requests will be processed concurrently (useful for CPU intensive endpoints
- Fleet usage load shedder - reserve a certain amount of capacity to serve critical requests. Ex: if 80% of capacity is being currently used, drop all low priority requests to allow critical requests to be processed
- When traffic is dropped, slowly allow to traffic to be brought back in to prevent flip-flop
- Stripe uses Redis and token bucket algorithm
- Catch exception, issue with algo or Redis should allow requests to go through
- Clear exception, HTTP has errors defined 429 Too many requests, 503 services unavailable.
- Build safeguards turn on/off rate limiters, adjust limit, set up alerts and metrics. How do you know requests are being limited or whether the thresholds are good?
- Code dark launch to understand behavior to test limits.

NoSQL vs SQL

- NoSQL primarily means - non-relational at DB level - no joins, no foreign key, search is generally on primary key though you can build secondary indices as well.
- NoSQL scales horizontally whereas SQL DBs scale vertically
- NoSQL sacrifices ACID properties to achieve horizontal scalability
- Tunable Consistency in leaderless clusters you can set r + w > n, i.e quorum consistency. Each distributed system works on the principle of the CAP theorem. The CAP theorem states that any distributed system can strongly deliver any two out of the three properties: Consistency, Availability, and Partition-tolerance. Cassandra provides flexibility for choosing between consistency and availability while querying data. In other words, data can be highly available with a low consistency guarantee, or it can be highly consistent with lower availability. For example, if there are three data replicas, a query reading or writing data can ask for acknowledgments from one, two, or all three replicas to mark the completion of the request. For a read request, Cassandra requests the data from the required number of replicas and compares their write-timestamp. The replica with the latest write-timestamp is considered to be the correct version of the data. Hence, the more replicas involved in a read operation adds to the data consistency guarantee. For write requests, the requested number is considered for replicas acknowledging the write.
- RDBM choose consistency over availability and require expensive hardware to perform complex query over just key-value lookup.
- NoSQL generally does not work well when a query modifies multiple objects

- Naturally, the time required to get the acknowledgement from replicas is directly proportional to the number of replicas requests for acknowledgement. Hence, consistency and availability are exchangeable. The concept of requesting a certain number of acknowledgements is called tunable consistency and it can be applied at the individual query level.
- There are a few considerations related to data availability and consistency:
- The replication factor should ideally be an odd number. The common replication factor used is three, which provides a balance between replication overhead, data distribution, and consistency for most workloads.
- The number of racks in a data center should be in multiples of the replication factor. The common number used for nodes is in multiples of three.
- There are various terms used to refer to the consistency levels –
- One, two, three: Specified number of replicas must acknowledge the operation.
- Quorum: The strict majority of nodes is called a quorum. The majority is one more than half of the nodes. This consistency level ensures that most of the replicas confirm the operation without having to wait for all replicas. It balances the operation efficiency and good consistency. e.g.Quorum for a replication factor of three is (3/2)+1=2; For replication factor five it is (5/2)+1=3.
- Local_*: This is a consistency level for a local data center in a multi-data center cluster. A local data center is where the client is connected to a coordinator node. The * takes a value of any specific number specified above or quorum, e.g. local_three, local_quorum.
- Each_*: This level is also related to multi data center setup. It denotes the consistency to be achieved in each of the data centers independently, e.g. each_quorum means quorum consistency in each data center.
- The data written and read at a low consistency level does not mean it misses the advantage of replication. The data is kept consistent across all replicas by Cassandra, but it happens in the background. This concept is referred to as eventual consistency. In the three replica example, if a user queries data at consistency level one, the query will be acknowledged when the read/write happens for a single replica. In the case of a read operation, this could mean relying on a single data replica as a source of truth for the data. In case of a write operation, the remainder replicas receive the data later on and are made consistent eventually. In case of failure of replication, the replicas might not get the data. Cassandra handles replication shortcomings with a mechanism called anti-entropy which is covered later in the post.

- Various types of NoSQL:
    - Key-value: simplest, could be in-mem or persistent. MemCached, Riak, dynamodb, Redis
    - Wide-Column: Key->multiple vals (columns). Cassandra, BigTable see https://indexoutofrange.com/What-is-the-problem-with-key-value-databases-and-how-wide-column-databases-solve-it/
    - Document DB -> mongo db, couch db
    - Graph db ->

Containerization

- https://platform.sh/blog/containers-are-the-new-static-binaries/
- Dynamic link vs static link: dynamic link faster to load the app if shared library is already cached by OS, otherwise could be slower as fetch time of small individual libraries could be slower than a single larger statically linked binary.

Timeseries Database

- Timescale blog
- Timeseries data are generally append only data streams. There is minimal updates
- The data generally comes in time ordered fashion though some data may arrive out of order.
- There is another primary key like device id, metric id
- Couple of different ways to save it:
    - Using columnar storage as data is highly compressible, increasing the time series; the difference between consecutive rows can be represented in fewer bits. Use snapshotting etc.
    - Timescale uses B+ tree, it internally does horizontal partitioning and the query optimizer is aware of it. It calls it chunks, each chunk contains one metric (therefore physically a separate table) and the most recent chunk is always in memory for new inserts.
    - Advantage for using SQL db like timescale is richer query language and support for secondary index.
    - Also since updates is not needed, you can use a stripped down version (like no version vectors etc) of a noSql db which generally spends a lot of resources in maintaining update consistency.

Kleppmann: Chapter 3: Storage and Retrieval

1) Database types and storage engines.
    a) Relational DB and NoSql are DB types
    b) Log-structured like LSM trees and page-oriented like B-Trees are storage engines.
2) Index is additional structure derived from primary data
3) Append only log segments advantage:
    a) Appending and segment merging are sequential write, therefore advantageous on spinning disk
    b) As appending doesn't overwrite, partial writes are safer
4) Hash index limitation
    a) On disk hash maps are cumbersome because of random seek request so only useful if all keys fits into memory
    b) Range-key queries are not possible

5) LSM Tree - Log structured merge tree, log segments, compacted and merged in a tree like manner?
    a) Each log segment is a key-value pair, which are written in the order key-value pairs arrive. So a key in a segment can be present several times with the later key-value taking precedence. Of course during compaction, duplicate keys are eliminated. Merging of keys is inefficient because they either first need to be sorted to be merged or it becomes 0(n^2) operation to merge two unsorted segments.
    b) This is where SST (sorted string table) comes in. SSTs are kept in sorted order unlike append-only log that is kept in arrival order. The keys in a single SST segment are represented only once but the key may be duplicated in multiple segments. The merge steps use the key from the latest segment. SSTs need to be kept in memory as you cannot maintain SSTs on disk though it is possible. For crash recovery purposes, you write a log to disk first then update SST. Discard, log once the SST is written to disk. The written SST then becomes a segment.
    c) SSTs in memory are generally represented as balanced trees (AVL, Red-Black). Another name for in-memory SST is **memtable**.
    d) On disk SSTs are sorted key/value pairs stored in an array-like fashion. Because disks are continuous blocks of memory. There is an index in the front, possibly a sparse index, that provides an offset on the disk for keys so that the entire segment doesn't need to be read.
    e) The index of SSTs is always loaded into the memory.
    f) The in-memory SSTs are called memtable. There is usually only one memtable and several in-memory SSTable indices.
    g) So writes are done to memtable.
    h) Reads are done from memtable and in-memory SST indices.. Followed by appropriate reading of the SST block.
    i) Bloom filter is used to quickly eliminate SST indices that does not need to be scanned
    j) The above is called LSM tree which provided the following storage characteristics:
        i) Quick read/write of key-value pairs.
        ii) Data can be streamed in sorted order (after merging and elimination).
    k) On-disk, there are several components of SSTables
        i) Data.db (actual data)
        ii) Index.db (index to keys with points to position in datafile offset)
        iii) Filter.db (bloom filter)
        iv) Compressioninfo.db (any info required for decompressing)
        v) Statistics.db, crc.db etc
        vi) https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmlHowDataWritten.html
    l) There are 2 main compaction strategies, size tiered and level tiered.
        i) In Size tiered, 4 SSTables from the previous levels are merged. Memtable is 4 mb, then each SSTable in Level 1 is 16 mb, 64mb in Level 2 and so

on. Each level also contains 4. The last level is HUGE (level 7 is 64 GB), so when compacting Level 6 you need additional 64 GB of space and thus causing “space amplification” even though it is temporary. Generally it is recommended that LSM db keep 50% free space.

ii) In level tiered compaction, each level is called a “run” and a run contains a specific number of SSTables of fixed size, and each SSTable in a run contains a specific range of keys that do not overlap with each other. Level 1 contains 10 SSTable of 160mb each, Level 2 contains 100 tables of 160 mb each. When a SSTable in a run exceeds 160 mb it is pushed to the next level which impacts 10 SSTable in that level. So at max 11*160mb ~= 2 gb of extra space is needed. Thus significantly less space amplification than STCS. However, much larger write amplification. Level 0 is straight dump from memtable, 4 Level 0 goes into Level 1, then 10 Level 1 and the 4 Level 0 is combined and merged and partitioned into 10 different key ranges and replacing the entire Level 1. If any Level 1 SST exceeds 160 mb, it is taken and merged with 10 Level 2 and those 10 level 2 are replaced.

iii) Since each level can hold 10x data of the previous level, in an almost full level, 90% of data will be at the last level and since there are no overlaps in a SSTable in a level, read performance is very good.

iv) So with lots of reads and less write, LTCS wins, comparatively lots of writes, STCS wins but at the cost of space amplification (temporary).

v) https://www.scylladb.com/2018/01/31/compaction-series-leveled-compaction/

6) B-Tree is popular for relational and is also an on-disk indexing system. The root page contains a range of keys for each of the children and so forth. The b-tree is always kept balanced. There are several hundred children per node and therefore the depth is usually very small (like less than 10). Each node is a page in disk, usually 4kb or 8kb.
    a) Since the keys are modified in place unlike immutable SST, there are several variants to make write more reliable like write-ahead-log or copy-on write.
    b) Other optimizations are abbreviating the key to save space as the parent key can be used as a prefix.
    c) In a B+ tree, values are stored in leaf nodes as opposed to B tree, where the key and value are stored in all nodes. This allows for more keys to fit in a page and thus the entire tree takes less memory. Therefore it is easier to fit B+ tree in memory than B tree, generally most DBMS use B+ tree, MongoDB version older than 3.2 used B tree and Discord saw performance degradation once the DB size exceeded a certain size.
7) Comparing B+ tree to LSM
    a) Generally B+ tree for higher read throughput and less write intensive. LSM for high write, frequently changing data. But different benchmarks provide different results, so best test for a particular load characteristics.
    b) Typically LSM trees have smaller write amplification (??? not clear why).
    c) LSM tree also compresses data better (not sure why)

    d) LSM tree does sequential writes which gives it a better performance on spinning disks.
    e) Background compaction and merge may impact performance due to sharing of disk throughput, cpu resources for compression, decompression, merging.
    f) In a B-tree, keys are stored only once and therefore they provide strong transactional semantics.
8) Multidimensional index
    a) The common variety of concatenated keys
    b) R-trees convert two dimensional location (latitude/longitude) into a single number. Used by many GIS systems.
    c) This type of index narrows down the search space simultaneously on multiple dimensions rather than intersecting 2 searches.
9) In-memory database though it writes to disk for durability serves most of its read requests from memory. Some in-memory implementations use virtual-memory-like semantics and are able to serve more data than actual physical memory. The performance of in-memory databases is better than regular db with full data in memory because it does not have the overhead of encoding data in form that can be written to disk (example, compaction merging of lsm trees)
10) Columnar databases:
    a) Instead of row wise storage, values of a column are stored together.
    b) All columns are saved in the same row order.
    c) Very helpful for data that doesn't change often and used for analytics.
    d) Column compression is a huge advantage and overall disk space requirement is less. Various methods of compression are used for example, bit mapping for country codes. The column which is sorted sees even more advantage from sorting.
    e) Selection on a few columns requires significantly less disk reads.
    f) Since the database is stored column-wise, they are automatically vertically partitioned. The tables are also horizontally partitioned on some keys. Very common to vertical partition by dates since many use cases of data warehouses are to save historical data.
    g) Also Martin Kleppmann says it makes efficient use of CPU cycles due to better prediction and vectorization.
    h) Some implementations (ex Vertica) take advantage of database redundancy by choosing a different sort order in each replica.
    i) When not batch loading, in memory tables (like memtables in LSM tree) are used for the batch writing. Recall that each write needs to modify several disk pages.
    j) Since columnar db doesn't see many writes, it is common to save materialized aggregate views of data (like a data cube, the 3 axis being aggregate).
    k) Read this when you get a chance [The design and implementation of modern column-oriented database systems | the morning paper]

Kleppmann: Chapter 4: Encoding

1) JSON, XML, CSV popular text encoding. JSON, popular in Rest services, is verbose and does not carry binary data very well. To pass binary data, it needs to be encoded in base64 which is 33% larger. These are generally backward and forward compatible as they are text based and new fields can be ignored.
2) Language supported encoding, like python pickle, Java serializable are generally not a good idea:
    a) producer/consumer of data objects are bound to the same language and stuck on it for a long time
    b) Security issue as arbitrary objects can be instantiated
    c) Inefficient encoding/decoding.. Example java.serializable
    d) Versioning is often ignored and schema evolution forward/backward compatibility is a challenge
3) BSON or binary json encodes attribute names which takes a lot more space than using tags as in protobuf.
4) Schema based encoding, Google protocol buffers, Facebook Thrift uses tags for variable names. Encoded are the tag, type and length (for variable length types only like string). Thrift offers a couple of different encoding like compact and binary.
5) The schema is called IDL (interface definition language) and comes with code generators for reading and writing the objects.
6) Forward compatibility is maintained by ignoring new tags (and thus new fields cannot be required attribute). And new fields should be previously unused tags. Also a change in types may result in loss of precision for example int64 when read by int32 code will get truncated.
7) Apache Avro takes a different approach, tags are not encoded thereby saving space. This means that the reading code needs to know the schema with which the encoding was done. The reader still has its own schema with which the application code was built, the Avro library uses the writer schema to map into reader schema. With just writer schema, code generation for readers is not required as Avro library can provide a getattr style of semantics. Though for compiled languages it is still helpful (for IDE popups) but not required for code generators. Writer schema can be passed in multiple ways:
    a) In a large file, schema can be written in the beginning of the file
    b) For DB records, writer schema can be read once for an entire set of records. This does not preclude using multiple versions of writer schema, as a set of records can refer to one schema
    c) During a network session the schema can be exchanged in the beginning for that session, just like SSL certificates.
8) One big advantage of Avro-like writer schema is that as data outlives code, code can read very old data that have been archived.
9) REST vs SOAP vs RPC vs Message Broker
    a) Rest uses http and JSON and is simple to use and popular for cross-organization.
    b) Can test Rest services with a browser (http and uri are web addresses)
    c) SOAP is a protocol and uses XML based encoding

d) SOAP uses XML based WSDL (web services description language)
e) SOAP is complex, requires tooling, code generators, IDE to test. Need SOAP vendors. Interoperability is troublesome due to its complex nature.
f) RPC design philosophy is to make it look like similar to function call which is bad philosophy in itself as there are intrinsic differences:
    i) Unlike function calls which will succeed or fail for a given input and operation, RPC may fail due to network unreliability.
    ii) If there is a timeout, you don't know the underlying cause
    iii) Network may be broken in one direction, request is applied but response doesn't come to user, so RPC should always be idempotent
    iv) RPC parameters needs to be serializable
    v) Any service has other parts in the ecosystem like caches, load balancers, proxies, firewalls, testing tools that may not be readily available
    vi) Message Brokers (MQ, Kafka etc) sit in-between client and server and provide buffering (q), caching, redelivery, one to many, many to many, one way requests (also acts as a proxy, load balancer).

Kleppmann: Chapter 5: Replication

1) Reduce latency (geographically close), horizontal scaling, high availability
2) Single leader replication
    a) Writes to leader, read from others
    b) Usually asynchronous replication, synchronous replication causes low availability.
    c) Writes may be lost if leader fails after confirming the write but before replication
    d) Usually there is one synchronous follower to fix above, this is called semi-synchronous
    e) To set up new followers, leaders need to maintain snapshot copy as data is always in flux.
    f) When a leader fails, a new leader needs to be elected or selected. Best candidate is one with the most up-to-date copy.
    g) Detection of failed leader, heartbeat, ping, crash. What is the right timeout, partial network failure.
    h) Lost writes are dangerous,for example, if auto incrementing ID are used, previously used ID might be issued again.
    i) Two node might act as leader - split brain
    j) Single leader causes high latency writes - geographically apart
    k) When new leader is elected, how to tell clients about new leader. This is called request routing problem. a) send to any node which forwards to the leader b) add a routing layer
3)
4) Implementation of replication logs:
    a) Statement Copy:

        i) Non-deterministic (now()) statements cause inaccurate replication. Replicators need to replace non-deterministic with deterministic statements at all levels including stored procs.
        ii) Statements with auto-incrementing, stored procs with side effects etc.
    b) WAL shipping
        i) Ship the same WAL log that is used to write to DB.
        ii) WAL is described at a very low level with block addresses, offsets and changes. This is problematic if the followers are not exactly the same, version upgrades, software, hardware changes becomes difficult if not impossible.
    c) Row-based logical replication
        i) Ship new, updated, deleted row in its entirety based on identifying information (rowid, primary key etc).
    d) Application level replication - useful if replica is an entirely different system. Can be done DB app level like triggers or even higher.
5) Replication Latency:
    a) Eventual consistency in async replication
    b) Reading your own writes is challenging (read after write)
        i) Read from the leader if chances of reading modified data is high, such as user own info rather than other people info.
        ii) Track last update time for each data set.
        iii) Let the client side track the update time and provide it when reading. This is not useful if the user can read the same data from multiple devices (clients). Also differences in clock can be an issue so use logical timestamps.
    c) If a user reads from different replicas, updates can out of order, so for monotonic reads always read from the same replica
    d) In a sharded database, reads will come from different replicas which may cause records to be arranged in a different order than the write order. Therefore if writes have a causal relationship then they should be written to the same shard.
6) Multi-leader replication
    a) Good for multi-datacenter application
        i) Intra-datacenter latency is hidden from user
        ii) Tolerates datacenter outages
        iii) Tolerates network outages
    b) Write conflicts are an issue and require conflict resolution. Example calendar on users personal device that needs to be synced with the cloud.
    c) Collaborative editing like google docs has some fancy algorithm CRTD, conflict free replicated data types. These are a family of data structures that merge automatically through some algo.
    d) All eventual consistent updates come converge otherwise write conflict, here are some techniques:

        i) Last write wins - LWW - clock synchronization is an issue. Or use UUID with higher uuid winning or always choose writes from higher number replicas.
        ii) Somehow merge it, rocksDb provides hook to implement custom merge methods for their LSM tree merge, perhaps something similar can be done here
        iii) Record conflict in a separate structure and resolve them later at an application level.
7) Leaderless Replication
    a) In this scheme, writes are done to multiple replicas before it is confirmed to a client. And reads are also done from multiple. The number of writes and the number of reads together must exceed the number of instances. The min number of reads and min number of writes is preconfigured and must exceed the instances. This is called Quorum consistency. Generally an ideal setting of r and w is n+1/2
    b) There is no failover, as the client doesn't care if one of the replicas is down since it tries to read from multiple replicas. Therefore there is no separate process to bring a node upto speed, it can be done dynamically or through a back ground process.
        i) Read-repair: When reading records, if client detects stale records on a replica then repair it then
        ii) Anti-entropy process: Through a back process that constantly looks for differences and fixes it.
8) Concurrent writes. Doesn't necessarily mean writing at the same time, all it means that when the second write happened it didn't know about the first write. To detect it, all records should have either a version number or transaction id. When writing an update the server will compare the version number with the modified record and if different, will either fail or save a version vector.The client will get a new version number.
    a) A version vector is a way to handle concurrent writes. It is a way of saying the database will maintain multiple versions (streams) of the same record. If there is a conflict it will generate a new stream, otherwise the appropriate record will be overwritten.
    b) It is upto the application or client on how to handle the vector version.

Kleppmann: Chapter 6: Partitioning

1) For horizontal scaling of databases. Issues with replication of each shard/partition is same as the one without partition
2) Partitioning can be done on keys directly or on hashed values of keys. With hashed values, distribution of keys is more uniform. Hashing algorithms should be chosen such that they produce the same hash across platforms. Range queries on keys cannot be done with a hashed partitioning scheme.
3) There is an in-between method, with compound primary key, the first part of the key is used for partition and the rest for sorting within a partition. For example, all updates by

user within a time range. Users mapped to a partition and within the partition index on time can be used to pull data.

4) Hot spots can exist, there is no real easy way to automatically identify and handle them. Special logic and bookkeeping is done. For example, a celebrity user with millions of followers, the celebrity user itself might need to be partitioned.
5) There are local indexes on secondary keys, clients will need to search multiple partitions for search on the secondary index. For that reason a global index is also built which adds to bookkeeping, and the secondary index itself can be partitioned. Index will point to primary-key. This is called term-partitioning.
6) In practice, global secondary indexes are updated asynchronously.
7) Rebalancing of partitioning needs to be done without bringing down the database and by minimizing data movement.
8) Mod by N is really bad, as each new and removed node impacts every key.
9) Have multiple partitions within a node. Keys are mapped to partitions and partitions to nodes. When a new node is added, partitions from several nodes are taken to create new nodes thereby reducing load on several nodes. Vice versa for removal of nodes. Also helps with mismatched hardware as weaker machines can host fewer partitions.
10) Partitions can be dynamically managed but as reallocating partitions is IO intensive it should be carefully managed. The dynamic management of partitions is similar to the LSM tree. Note that dynamic partitioning is easier with key-range partitioning but can also be done with has partitioning but would need additional bookkeeping.
11) Request routing:
    a) Client queries all nodes
    b) Client queries any node which then redirects the query
    c) Client goes through a routing intermediary
    d) Client has a copy of the mapping
12) Routing through an intermediary is a common approach, careful to not have a single point of failure. Zookeeper is often used to keep the routing tier up to date on node availability and key distribution.

Kleppmann: Chapter 7: Transactions

1) Atomicity: In multithreading atomicity is related to reading and writing of a particular record. In database, it is an extension of the same concept that extends to a group of records. For example, in a transaction, if part of the write fails, the earlier writes are rolled back.
2) Consistency: Certain statements about data that must always be True. Ex: credits and debits must be balanced across all accounts. Generally these fall under the responsibility of the application though databases provide some tools like unique constraints, foreign key constraints etc. Per Kleppmann, consistency really doesn't belong to databases but C was "tossed" in as a marketing tool.
3) Isolation: This is arguably the most complex and important feature. In essence, it means that concurrently executing transactions are isolated from each other. That is the end

result is such that those transactions were executed in serial. There are various interesting flavors of isolation that are worth reading again and again.

4) Durability: Written data is not forgotten. If isolation is not implemented properly, writes may still be lost. And despite replication, durability is still an issue (primarily due to conflicts) though in a disaster scenario all physical storage may be lost. Some corrupt data may be discovered so late that it has already been passed on to backups.
5) There are various flavors, specifically for multi-object operations. Single object writes also require atomicity and isolation, as some objects can be quite large and the database could crash writing midway. So write separately and then point to the new record.
6) Many transactions are multi-object: like updating records and indexes, and maintaining foreign keys. Even in a document model (which is likely denormalized) one update can impact many records.
7) Retrying an aborted transaction at an application level is tricky, as an already committed transaction may not be idempotent. Transactions may actually succeed at db but due to network failure may not inform clients so there would be no way to rollback on the client side. Clients would generally not know which parts of a multipart transaction actually completed.
8) Weak Isolation (isolation for a single DB statement vs serializable isolation)
    a) **Read committed**:
        i) Dirty read: Read only what has been committed. When a transaction runs on this isolation level, a SELECT query sees only data committed before the query began and never sees either uncommitted data or changes committed during query execution by concurrent transactions. (However, the SELECT does see the effects of previous updates executed within this same transaction, even though they are not yet committed.) Notice that two successive SELECTs can see different data, even though they are within a single transaction when other transactions commit changes during the execution of the first SELECT.
        ii) Dirty write: Overwrite only what has been committed. If a target row found by a query while executing an UPDATE statement (or DELETE or SELECT FOR UPDATE) has already been updated by a concurrent uncommitted transaction then the second transaction that tries to update this row will wait for the other transaction to commit or rollback. In the case of rollback, the waiting transaction can proceed to change the row. In the case of commit (and if the row still exists; i.e. was not deleted by the other transaction), the query will be re-executed for this row to check that the new row version still satisfies the query search condition. If the new row version satisfies the query search condition then the row will be updated (or deleted or marked for update). Note that the starting point for the update will be the new row version; moreover, after the update, the doubly-updated row is visible to subsequent SELECTs in the current transaction. Thus, the current transaction is able to see the effects of the other transaction for this specific row.

b) **Snapshot isolation** or read repeatability: When doing several reads within a transaction, the ability to freeze the db at a particular time and read as if no updates have been done to it. For example, when taking a database backup from a live database.
c) The way these isolation levels are implemented is by maintaining various versions of a row. Each version has a transaction-id and each read/update also has a transaction id and these transaction ids are monotonic for comparison purposes. Record with appropriate transaction id is considered. This is called Multi-version concurrency control. MVCC.
d) For example:
   09 AM : A1, B1,
   10 AM: (A1,A2*), B1 A2* means modified but not committed yet
   11 AM: (A1, A2), B1

   If a transaction starts at 9AM and tries to read record A after 11 AM
   * A2 will be returned for Read Committed isolation level
   * A1 will be returned for Snapshot Isolation level (also called read repeatability because in the same transaction you will always get the same value of a row despite concurrent updates by another transaction).

   After 10 AM but before 11 AM, A2* will be returned if no isolation level is used.
e) How are indexes maintained with MVCC? All versions are indexed and then filtered out later. Indexes are maintained asynchronously when old versions are purged. Deleted records are only marked as deleted.
f) Preventing lost updates. Concurrent read-modify-write operations.
   i) In simplest form, a single row atomic update. (update x=x+1 where y). This is done by taking an exclusive read lock on the row.
   ii) Another explicit option is the application locks the rows it is updating (like select *.. For update
   iii) Lazy detection, detect at time of update using version number/xact-id comparison.
   iv) If lazy detection is not provided by db, applications can compare and set. Update to new content if content is same as old content. But again as with multi-threading not safe unless DB implementation guarantees that compare and set is safe.
   v) With replication, there is also conflict. Detect conflict and save multiple versions to resolve later.
g) Write skew: is concurrent writes to non-overlapping records that breaks the constraint checked as part of the updates. As the constraint usually cannot be materialized in a lockable row, it is called a phantom. Solved using:
   i) Db level constraints through triggers or materialized views. Application level. Materialized views, for example at least one doctor on call, have a materialized table for number of doctors on call and update it as part of a

transaction with constraint that number > 0. Or db unique constraint for having a unique username

ii) Serializable isolation by taking locks on all potentially impacted rows

9) Serial isolation:
   a) Serial isolation solves all problems with concurrency but comes at a cost of performance. Example: single threaded will force all transactions to be serial. Others are 2 phase locking and serializable snapshot isolation.
   b) Single threaded feasible with in-memory db as I/O is negligible and OLTP transactions are short. Obviously not suitable for OLAP but as there are no updates you don't need isolation.
      i) Use stored proc where possible to reduce network I/O. Stored proc got a bad name:
         (1) Non-standard stored proc language
         (2) Code not in version control and therefore difficult to manage, debug, deploy
         (3) Badly written stored proc impact the entire db ecosystem
      ii) Only updates need to go through a single thread, single cpu. Reads can be executed through snapshot isolation. For scaling beyond single cpu, consider sharding or partitioning but cross partition queries or updates run into problems.
   c) Two phase locking.
      i) Read locks are acquired in shared mode
      ii) Write locks are acquired in exclusive mode
      iii) Deadlocks needs to be detected and one transaction aborted
      iv) There is huge performance penalty and is therefore least desired
      v) Doesn't really protect against phantoms so predicate locking or index-range locking is also needed
         (1) Predicate locking is similar for select for update
         (2) Alternative is lock a suitable range in a index
         (3) Or worst case lock the entire table
   d) Serializable Snapshot Isolation:
      i) This is optimistic locking whereas two phase locking is pessimistic.
      ii) During a transaction, version numbers (using MVCC) of all records that are read is noted and if during commit, any of the version number is not current then the transaction is aborted.
         (1) If during read you know that version is not current and if there is a write in the transaction, immediately abort. If the transaction is only read, continue.
         (2) If while reading the record the version is current but becomes stale before committing, then the transaction that committed the read must notify read transaction that their reads are stale.
      iii) Performance depends on implementation. Instead of tracking every read record, databases can do some optimization like tracking a range of records instead.

iv) One particular advantage that read only queries doesn't need any lock so it particularly appealing to read heavy workloads
v) One disadvantage is the optimistic locking increased workload due to frequent aborted transactions. Therefore it makes heavy load conditions even worse.

Kleppmann: Chapter 8: The trouble with distributed Systems

1. Detecting faults: As response times are unbounded, a timeout is generally the only way to detect fault. Too short timeout may cause poorly performing service even worse as the load will increase and some action to be performed twice. Too long timeout may cause poor user experience.
2. Choose timeout value experimentally, measure across hos, networks to determine expected variability of delays. Instead of using constant timeouts, continuously measure response times and their variability.
3. Network systems are packet switched to increase the utilization of bandwidth. Synchronous networks like ISDN circuit based provide bounded delay but make poor use of resources.
4. Clocks are unreliable and nodes are always out of sync and time stamps should not be relied on to solve concurrency. Clocks can even go back in time due to frequent synchronization of the clock. There are API's to get monotonic time and that time is only applicable to that node.
5. Clock synchronization is done through NTP servers and it is only accurate within certain limits and several NTP servers should be used as NTP server itself may go out of sync or excessive network delay volatility could cause poor synchronization.
6. Time stamps can be used for global ordering but only if an accurate confidence interval API is available. For example, Spanner implements snapshot isolation across datacenters, it uses TrueTime API that returns confidence intervals. It deliberately waits for the length of interval to guarantee that any reader will not see the data????
7. Process can be paused for several reasons:
    a. Global garbage collection
    b. In Virtualized env, processes can be suspended
    c. User suspends the process manually
8. There if lease.isvalid(): process(request) is bad programming paradigm because process could be paused after lease check and lease can expire before request is processed.
9. Fencing tokens: a token service that provides monotonic tokens, so get a token (or lease), use the token to write a record. If the record has already been updated/written with a higher token then reject the write. ZooKeep can be used to provide token or transaction ids.
10. Byzantine faults are faults were a node send malicious or knowingly sends bad information. Generally distributed systems considered in this book are not byzantine fault tolerant.

Kleppmann: Chapter 9: Consistency and Consensus

1. Various forms of consistency
    a. Weak (Quorum setting where r+w < n)
    b. Strong (quorum read/write) (Cassandra, dynamo). Even with r+w>=n, some reads can return newer values, and later reads can return older values. Take an example of a system where n=3, r=2, and w=2. For clarity let's assume the 3 nodes are named A, B, and C. Consider this scenario: a write is in progress; node A has been updated while B and C are still in process of receiving the updated value. Clients reading from A and B will see the newer value (resolved using version vectors or last write wins) and clients reading from B and C will see old values. See page 334.
    c. Linearizable -> all clients see the same value irrespective of node they read from (etcD, Paxos). Also, there are no conflicts. Linearizability is important for example in getting locks.
2. Linearizability vs Serializability
    a. Serializability refers to serializing of transactions. Guarantee that transactions were executed in serial order even if they were executing in parallel
    b. Linearizability refers to the recency guarantee of read-write operation.

Distributed Cache

1. **Functional**: put and get **Non-Functional:** Scalable, high-available, high-performance (durability possibly if designing database)
2. Start with a simple design, local cache design which becomes an algorithmic problem. Hash table is suitable for local cache but there is no policy to evict keys so if the hash table becomes full no more keys can be added. Hashtable with eviction policy are LRU and TTL. Implement LRU in leet code. A TTL cacheSo think about LRU cache. An implementation of TTL policy is to randomly inspect a set of keys and check for expiry

UDP Hole Punching

- Allows peer to peer connection between two clients behind NAT. A public host is used to discover and exchange public IP and port.