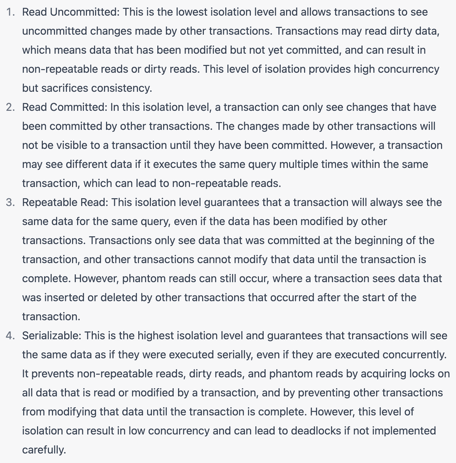
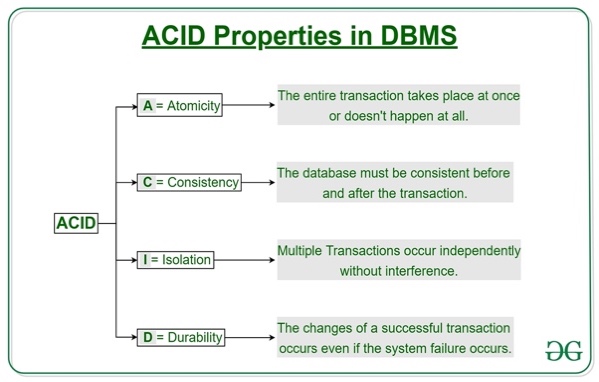
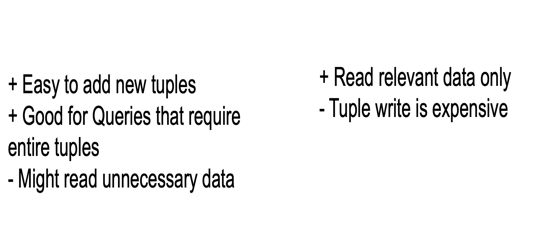
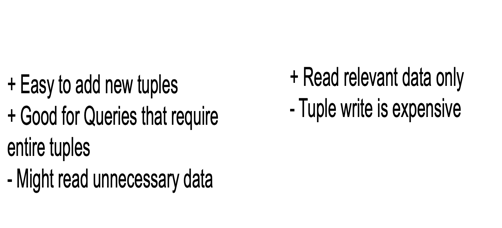
**Nested loop** join algorithm, for each row in the outer relation, the entire inner relation must be scanned. This means that each row of the outer relation results in a full scan of the inner relation, resulting in a large number of disk I/O operations. B(S) + B(S)B(R) | B(R) + T(R) B(S) when S is clustered| B(R) + T(R) T(S) when S is unclustered

**Block nested loop join** reduces the no. of disk I/O operations by processing multiple tuples from the inner relation at once. Instead of scanning the entire inner relation for each tuple in the outer relation, a block of tuples from the inner relation is loaded into memory, & each tuple in the outer relation is joined with the tuples in the block. Reduces the no.of disk I/O operations required since the block can be read into mem once & used to process multiple tuples from the outer relation B(S) + B(S)B(R)/(M-2)

**Hash-based algorithms** are commonly used in db systems to process queries that involve joining two or more tables. The basic idea behind a hash-based join is to partition the data from each table into multiple buckets based on a hash function applied to the join column(s), &then join only the tuples that fall into the same bucket(s). [hash based index – efficient selection on =lity predicated] 3B(R)

**Grace Hash Join algorithm**, there are two hash functions used to partition the input relations into multiple buckets. The first hash function is used to partition each input relation into in-memory buckets, &the second hash function is used to partition the disk-based buckets. 3B(R) + 3B(S) Assumption: B(R) and B(S) <= M2

**External merge sort** is an algorithm used to sort large amounts of data that cannot fit in memory. It is a variation of the merge sort algorithm that operates on disk files instead of in-memory arrays. Read+write+read = 3B(R) | Sort-Merge join B(R) + B(S)

The external merge sort algorithm works as follows:

1. Divide the data to be sorted into smaller chunks that can fit in mem, called runs.
2. Sort each run in memory using an efficient in-memory sorting algorithm, such as quicksort or heapsort. | Hash Join B(R) + B(S) if B(R) <= M
3. Write each sorted run to a separate file on disk.
4. Merge (M-1) the sorted runs into larger sorted runs until all the data is sorted.

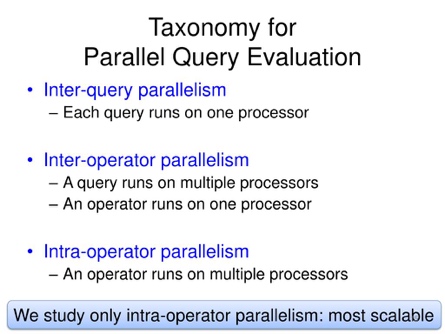
The merging step is the key to the efficiency of external merge sort. It works by reading a block of data from each run into memory &merging them into a larger sorted block, which is then written to disk. This process is repeated until all the runs have been merged into a single sorted file.

**Index-Based Selection** is a query optimization technique that involves using an index to identify the qualifying records in a table, instead of scanning the entire table. The idea is to use an index on the search key of a table to quickly locate the records that match a given search condition. By doing so, the query can avoid scanning the entire table, which can be a time-consuming process for large tables.

**Index nested loop join** is a type of join operation in a db where an index is used to access the data instead of scanning the entire table. In this join method, one table is accessed using an index while the other table is scanned sequentially.

**Index-based join** is a type of join operation used in DBMS to combine data from two or more tables based on matching values in their indexes. This type of join is commonly used when one or more of the tables are very large, &indexing can be used to speed up the join operation. Assume both R & S have sorted index. B(r) +B(s)

**Non-Repeatable Reads:** occur when a txn reads the same data multiple times & gets different results because another txn has updated the data in the meantime.

**Phantom Reads:** when a txn reads a set of rows that meet a particular condition, but while the txn is still running, another txn inserts new rows that meet the same condition, causing the first txn to see additional rows that were not there before.

**Lost Updates**: when two or more txn’s update the same data simultaneously.T he updates made by one txn can be lost, &the final result is incorrect.

**Dirty Reads**: one txn reads uncommitted data from another transaction. This can lead to incorrect or inconsistent results.

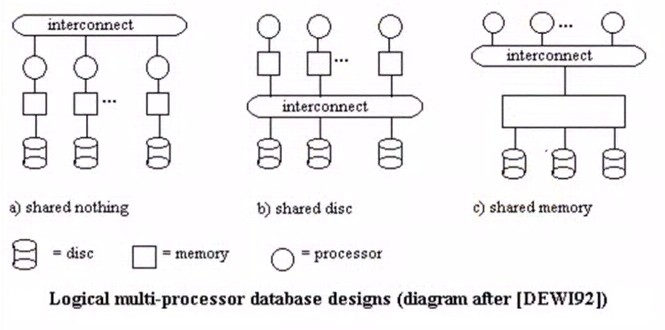
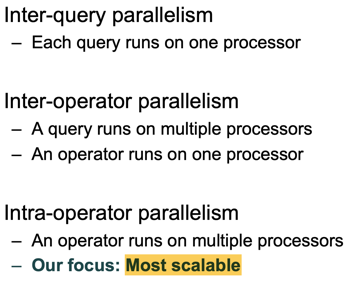
**Deadlocks**: occur when 2 or more transactions wait for each other to release resources that they need to complete their execution. Deadlocks avoidance & detection: acquire locks in pre-defined order; acquire all before

**Serializable execution** is a type of txn execution in a db management system that guarantees that transactions are executed as if they were executed serially (i.e., one at a time), even if they are executed concurrently. I.e, it ensures that the outcome of concurrent transactions is the same as the outcome of executing the same transactions serially in some order.(2 Phase Locking)

**Phantom problem** occurs when two transactions concurrently access the same data set & one txn adds or removes a row from the data set, causing the other txn to see an inconsistent view of the data.

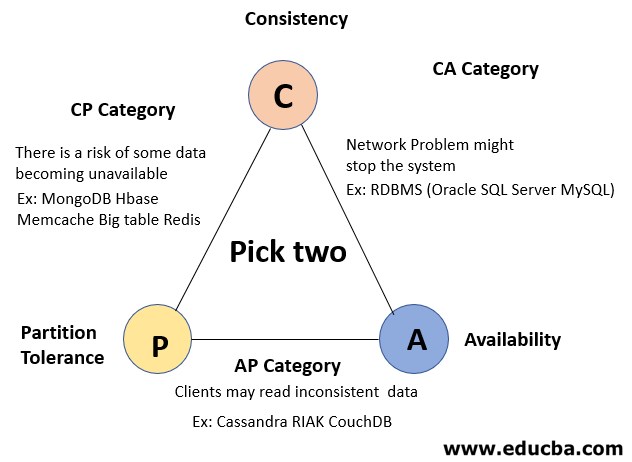
**Predicate locking**, a lock is acquired on a set of data based on a specific predicate or condition, rather than on the entire data set.

**Eventual consistency** is a concept in distributed systems that allows updates to be propagated asynchronously between different nodes in a system, with the goal of eventually reaching a consistent state. It allows for updates to be made to a distributed system without requiring immediate consistency between all nodes, but rather allowing for consistency to be achieved over time. Different nodes may have different views of the data at any given time, but eventually all nodes will converge to the same state. This approach is often used in distributed systems that require high availability and scalability, as it allows for updates to be made without requiring immediate synchronization across all nodes, which can cause performance and availability issues.



**Shared-memory architecture:** In this architecture, multiple processors or nodes share a common memory space, &can access &modify data directly. This approach can provide high-performance parallel processing, but may be limited by the size of the shared memory &the complexity of managing concurrent access.

**Shared-disk architecture**: In this architecture, multiple nodes are connected to a shared disk subsystem, which stores the db files. Each node has its own processors &memory, but can access the shared disk to read or write data. This approach can provide good performance &scalability, but may be limited by the bandwidth of the disk subsystem &the need for effective caching &synchronization.

**Shared-nothing architecture:** In this architecture, each node has its own processors, memory, &disk subsystem, &is responsible for storing &processing a subset of the data. The nodes communicate with each other over a network, &can coordinate their activities to process queries or transactions in parallel. This approach can provide excellent scalability &fault tolerance, but may be limited by the need for effective data partitioning &load balancing. Availability is a must; compromise on consistency

**Data portioning**

**Round robin** - Good load balance but always needs to read all the data

**Hash based partitioning** - Good load balance but works only for equality predicates & full scans

**Range based** – Great for range predicates but can suffer from data skew

PACELC Extends CAP &builds on eventual consistency:

*If Partition, trade A for C; Else, trade Latency for C*

**Consistent hashing** is a technique used in distributed computing to efficiently distribute &retrieve data from a large number of nodes or servers. Basically, to map each node or server in the system to a unique identifier or hash value, &then map each piece of data to a corresponding hash value using a hash function. By doing so, each piece of data can be easily &efficiently routed to the appropriate node or server in the system. key benefits of consistent hashing is its ability to handle node failures &changes in the system topology. When a node fails or is added to the system, only a small no. of keys needs to be remapped, since most keys will still be mapped to the same node as before

**V clock:** If all 1st object’s counters are **strictly less than or equal** to the 2nd clock, the 1st object happened before the second. Else, conflicts detected!

**Write operations**- Coordinator generates vector clock & stores locally

– Coordinator forwards new version to all N replicas ; If at least W replica nodes respond then success!

**Read** **operations** - initial request sent to coordinator; Coordinator requests data from all N replicas; Once gets R responses, returns data

**Handling (temporary) Failures: Hinted Handoff -** Do not enforce strict quorum membership; Sloppy quorum: Involve first N healthy nodes from prefrefence list**;** Data returned from healthy but not 1st N nodes, contain a hint that this is temp;Responsibility is sent back when failed node recovers

**Memcached** is a distributed caching system that is commonly used to speed up dynamic web applications by caching frequently accessed data in memory. It is an open-source, high-performance, key-value caching system that can store &retrieve large amounts of data very quickly. LRU

Uses “slab” memory allocation to deal with memory fragmentation

**Gossip protocol** is a decentralized communication protocol used to disseminate information or state updates among a group of nodes in a distributed system. The basic idea is that each node periodically exchanges information with a randomly selected subset of its neighbors or peers, spreading updates throughout the system in a probabilistic manner.

**Quorum** is the min number of votes that a distributed txn has to obtain in order to be allowed to perform an operation in a distributed syste

**Partition key** is used to determine the partition or shard where the data will be stored. In Cassandra, data is typically partitioned across multiple nodes based on the partition key value. The partition key value is hashed to determine which node(s) will be responsible for storing the data. It is often chosen based on the application's access patterns or data distribution.

**Clustering key**, is used to determine the order of the data within a partition. Clustering keys are only used when data is stored within a partition, &allow data to be sorted or grouped by the clustering key value. In Cassandra, data within a partition is typically stored in sorted order based on the clustering key value. Clustering keys are often chosen based on the application's access patterns or query requirements.

**S**orted **S**tring **Tables** are **immutable data:** Performance, Data consistency, Durability

**Bloom filters** are a probabilistic data structure used to test whether an element is a member of a set. They work by using a bit array &a set of hash functions to represent the set of elements. When an element is added to the set, it is hashed using multiple hash functions, &the resulting indices in the bit array are set to 1. When testing for membership of an element in the set, the element is hashed using the same hash functions, &the corresponding bits in the bit array are checked. If all of the bits are set to 1, the element is considered to be a member of the set with a certain probability. If any of the bits are not set to 1, the element is definitely not a member of the set.

**Object Relational Mapping (ORM)** is a technique used in creating a "bridge" between object-oriented programs and, in most cases, relational dbs.

**Document Store Features**

-  Explicit locks

-  Strong consistency guarantees

-  Transactions

-  Joins

- *Literally* No SQL

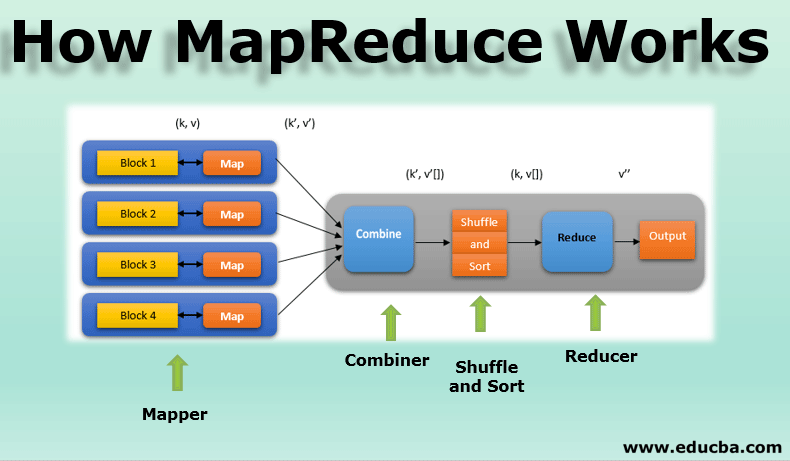
+  RESTful API

+  Data aggregation API (e.g., Map/Reduce)

+  Full text search

+  Secondary indexes

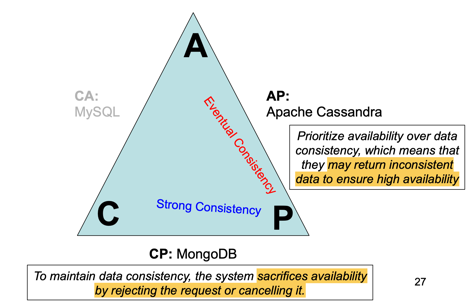
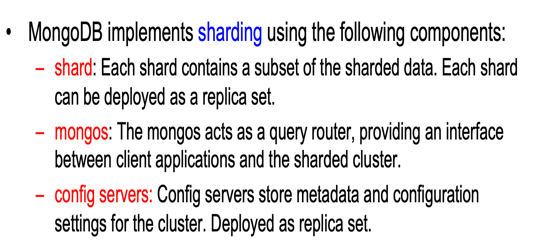
+  Automatic sharding (scale writes)

Commit log/Journal/**W**rite **A**head **L**og

**Write Concern** – { w: <value>, j: <written to journal>, wtimeout: <timeout> }

**Read Concern** “Local” (most recent data); “Majority” replicated most to 2nd

**Journal** is used to provide **durability** for write op. When a write op is performed, it is first written to an in-memory buffer called the write *op log* before it is applied to the db. The journal is *a file that is used to store a copy of the write op log.* It is used to provide durability for write op by ensuring that any changes made to the db are recorded on disk before they are applied to the db. This means that if there is a power failure or other system failure, it can recover the db by replaying the write oplog from the journal.

**Hash sharding** uses a hash function to determine which shard a document should be stored on. The hash value of the sharding key is used to distribute documents evenly across all the shards, which ensures that there are no hotspots. However, since the distribution is random, it can be difficult to query based on the sharding key. (some field is the shard key to be hashed) |THESE ARE SHARDING STRATS IN MONGODB|

**Range sharding** is based on dividing the sharding key range into non-overlapping intervals, with each interval assigned to a specific shard. This ensures that data is stored in a way that preserves the natural ordering of the keys, making it easier to query based on the sharding key. Also useful for partitioning data geographically or by time, where the range corresponds to a physical location or a time period. (Managed by config server)

**Optimistic concurrency control (OCC)** is a technique used in db sys to allow multiple txn to access the same data simultaneously while ensuring that conflicts are handled in a consistent manner. Each txn is allowed to read &modify the data independently, without acquiring any locks on the data. Before committing the changes, the sys checks if any other txn has modified the same data. If no conflicts are detected, the txn is committed. If there is a conflict, the txn is rolled back &the user is notified to retry the txn.

A **virtual node** is a concept used in distributed systems to allow for dynamic &efficient load balancing &partitioning of data. Where each physical node is divided into multiple virtual nodes, each responsible for a portion of the data or workload. This allows the system to easily add or remove nodes without having to rebalance the entire dataset, as each node only needs to manage a subset of the virtual nodes.

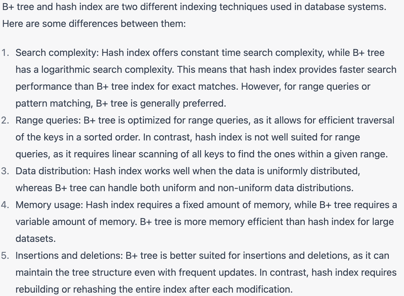
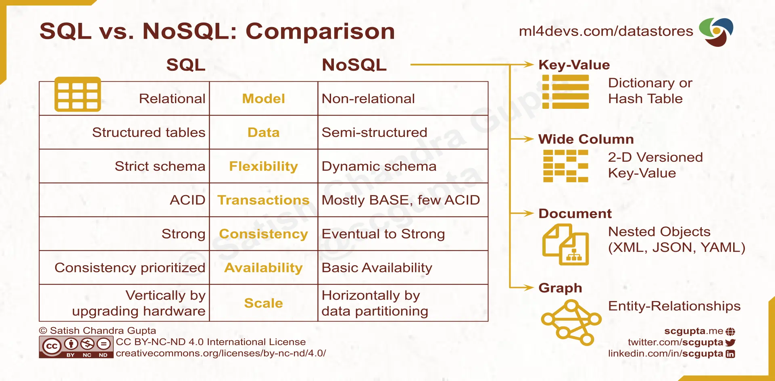
**Late tuple materialization** is deferring the materialization of tuples until they are actually required to produce the query result. This is done by delaying the projection operation, which selects the attributes to be included in the query result, until the tuples have been filtered and joined as necessary. Thus, the system can avoid the unnecessary materialization of tuples that are not included in the final query result.

**Partitioning** refers to the process of splitting a database into smaller, more manageable pieces called partitions. Each partition is assigned to a different node, and the data is stored and processed locally. It is typically done based on a range of values or a hash function applied to a partition key. Commonly used in shared-nothing

**Sharding**, is a specific type of (horizontal data) partitioning that involves dividing data based on a specific shard key. The shard key is a field or set of fields that are used to distribute data across different shards or nodes. Often used in shared-everything, where all nodes have access to the same storage and can access any piece of data.

**Two-pass algorithm** based on sorting is a data processing algorithm that operates on large datasets by first sorting the data (Duplicate elimination or Grouping 3B(R))

In the first pass, the algorithm reads the data and sorts it based on a specified key or set of keys. This can be done using a distributed sorting algorithm such as MapReduce or Hadoop. In the second pass, the algorithm performs the desired operation on the sorted data. Join (can’t fit in mem) 5B(R)+5B(S) Assumption: B(R) <= M2, B(S) <= M2.

Else if both fit in memory, do merging as part of the sort ) 3B(R)+3B(S)

**Lock Granularity**

**Fine granularity locking** (e.g., tuples) – High concurrency – High overhead in managing locks

**Coarse grain locking** (e.g., tables) – Less overhead in managing locks– Many false conflicts

**Normalizing** a database involves removing redundancy which means there will be no duplicate entries of the same data. **Denormalization** is a process that adds some redundant data into a normalized database to enhance its functionality and minimize the running time of specific database queries.

**Parallel join** is a technique used in database systems to execute a join operation between two large tables more efficiently by distributing the processing across multiple processors or nodes. It works by dividing the join operation into smaller sub-tasks and assigning each sub-task to a separate processor or node. These sub-tasks can be executed in parallel, resulting in a significant reduction in the overall execution time of the join operation. It can be implemented using various techniques such as hash join, sort-merge join, and parallel nested loop join.

**Cassandra**- Hashmpas of hashmpinternally, no join, Very Scalable, Eventual c ( quorum)

**MongoDB**- Replica set is max at 50, bottleneck is config server, Strong C

