Most of the database transactions in today's time are geo-distributed, as the organizations serve a global user base. Thus organizations are using new cloud-based systems, which can scale OLTP workloads. One of the solutions to this is CockroachDB. It is the scalable SQL database management system, supporting global OLTP workloads, along with maintaining strong consistency and high availability. It is also resilient to disasters using automatic recovery mechanisms and replication. It provides a new novel transaction model supporting consistent geo-distributed transactions.

It uses a shared-nothing architecture, where nodes can be used for data storage as well as computation. Each of the nodes has a layered architecture. SQL Layer is an interface for database interactions. The SQL engine in this layer, converts high-level SQL statements into low-level write and read requests for the underlying key-value stores. Transactional layer ensures atomicity across different KV pairs, and also guarantees isolation. Distribution layer provides an ordered keyspace, which can be queried by the key for the system and user data. Range-partitioning on the keys divides the data into contiguous ordered chunks with size of approximately 64 megabytes, called Ranges, allowing it to move between different nodes. The distribution layer routes the query subsets to appropriate Ranges. The replication layer replicates each of the range at 3 times for each of the partitions, and ensures the modifications durability under a consensus-based replication. The storage layer represents a local disk based KV store, ensuring performance efficient SQL execution. RockDB is used for this storage layer.

Raft consensus algorithm aids in the consistent replication, where a group of replicas form a Raft group. Each of the replicas can be a leader or a follower. Also, one of the replicas can act as LeaseHolder, in a Raft group. Load distribution or redistribution happens in case of the addition or removal of nodes, or on the temporary or permanent failure of the nodes, using heuristics. Replica placement happens automatically or manually using configurations. Thus these leaseholder and replica placement allow for different possibilities for policies of data placement.

Serializable Isolation is guaranteed by variation of the multi-version concurrency control (MVCC). For minimum latency, the nodes are connected which are closer geographically. Each of the operations are issued, after the previous operation has received a response. Thus, to prevent the operations from being stalled while replication is going on, two important optimizations are employed by the coordinator: Parallel Commits and Write Pipelining, which combined allow multiple multi-statement SQL transactions to complete with just one round of replication latency. Here, the operations are tracked which are not fully replicated. For atomicity of the transactions, all of the writes are considered as provisional until committed, which are known as write intents. The intent consists of meta-deta of the key-value pair, along with the status of the key-value pair such as pending, committed, aborted.etc. Hybrid scheme is used for clock synchronization which uses hardware level and software level clock synchronization, which are used for timestamp ordering. CockroachDB employs different mechanisms for that in

cases of clock skew, the transactional isolation is not violated. For concurrency control, it resolves different conflicts such as read-write, write-read, write-write, here write-write conflict is resolved using a distributed deadlock detection algorithm.
The SQL layer consists of the parser, query optimizer, query planning and execution engine. CockroachDB uses Optgen, a domain specific language, for writing the transformation rules, which is compiled to Go language. The SQL query execution is handled in two ways: one is by gateway-only mode, in which the query processing for the SQL is done by the node that planned that query; the second is distributed mode, where other nodes also do the query processing, but during write queries, only read-only queries are executed. Two different execution modes are used depending on the plan complexity and the cardinality of the input, which are vectorized engine and row-at-a-time engine.

Strong Points:
1). Vertical and Horizontal scaling does not affect the efficiency of the CockroachDB, as much as it does to Amazon Aurora.
2). In comparison with some of the metrics with Spanner DB, the CockroachDB shows much better performance.
3). Seamless integration between different layers of the database, as all of the layers except the storage layer is written in one language that is Go.
4). The paper presents use cases as well, to realize the usefulness of the system, which not most of the other papers have covered this semester.
5). The lessons learned section provides a good insight into different mistakes which can be avoided by other developing systems.
6). The language of the paper is simple, and easy to grasp different concepts and algorithms used in the database design.

Weak Points:
1). The paper compares the performance of the CockroachDB mainly with Amazon Aurora, rather than a number of other widely used OLTP databases used in the industry.
2). The database does not support Snapshot isolation, and provides only Serializable isolation.
3). Raft currently has few challenges, thus making the system live with it, can result in few unexpected performance impacts.
4). From the learned lessons section, it can be inferred that few things which were part of the design of the system, lead to few performance hindrances of the system, and at times, proved to be incompatible with the goals of the database.

Question:
1). Does removing snapshot isolation in the database such as CockroachDB lead to much lower concurrency?

2). What can be the implications of the duplicated indexes on the database performance? I believe it can lead to longer write times.