WedgeDB: transaction processing for edge databases

Abhishek A. Singh
Univesity of California, Santa Cruz
Santa Cruz, California 95064
Email: abasingh@ucsc.edu

Summary

WedgeDB is an attempt to create an Edge database optimized for read only transactions. The current implementation of WedgeDB is built as a key-value store which supports distributed transaction processing. The network is divided into two specific types of nodes: Edge and Cloud. Cloud nodes host applications and perform view changes for distributed consensus. WedgeDB's design goal is to enable efficient execution of serializable transactions on the edge nodes. In WedgeDB Data is partitioned into clusters. A small number of closely connected (locality aware) Edge nodes are added to the cluster to ensure data replication and byzantine consensus on any transaction affecting the data in the partition. The nodes in the cluster are assigned by the Cloud node which is a trusted node in the WedgeDB network model. An edge node in the cluster is also designated as a leader to coordinate transaction execution. The rest of this paper discusses transaction processing on the Edge nodes.

Transactions are divided into types: read-only and read-write transactions and are processed differently. Read-only transactions can be processed by any of the nodes in a partition cluster whereas read-write transactions are processed by the leader of the partition using a deterministic transaction processing algorithm. Transactions are buffered by the client during execution. Clients create a transaction object which executes read-only transactions whenever data needs to be retrieved. The results of the read-only transaction are buffered as read-history in the transaction object. Any write operation performed using the transaction object is buffered. A call to commit the transaction results in the transaction object sending the transaction read history and write operations to one of the WedgeDB servers. The database, therefore supports two operations: read and commit. Every transaction response (either to read-only or read-write) sent back to the client contains a verifiable proof of its execution by every node that took part in the consensus.

When a client sends a read-write transaction to a cluster, the transaction is forwarded to the leader of the cluster. The transaction is added to a transaction batch and is verified to be serializable when added to a batch. If the transaction is serializable in the batch and consistent with the current history of the leader's data history, it can be committed. Transactions received by a cluster may also have operations which need to be executed by other clusters. These transactions are called Distributed transactions and are handled differently from transactions that affect data only on one partition - called "Local" transactions as they affect data local to one partition. When processing "Local" transactions, batches formed by the partition leader are committed via PBFT [1]. The prepare message contains the transactions in the entire batch. Once the other nodes in the cluster receive the transaction they verify if the transactions in the batch are serializable with their local history of the database. The normal phases of the PBFT consensus then continue. During the commit phase, each node also maintains and updates its local merkle tree with the updated keys committed during the transaction and forms a new merkle tree for the transaction batch [2]. The root of the tree along with nodes that affected the construction of the root of the merkle tree is signed and added to each transaction response by the nodes in the cluster as a proof of the transactions committed in the batch. This proof is stored by each node and allows read-only transactions to provide a proof of execution for every read-only transaction received by a node.

Distributed transactions span multiple partition clusters and are executed using two phase commit (2PC). Once cluster C's leader receives a distributed read-write transaction T, it creates a prepare request P_T for each cluster K_i whose keys are referenced in T. Each cluster K_i's leader then verifies the transaction and responds with a prepare response. Once the transaction has been prepared by all cluster K_i, a commit is issued by C. Within each cluster the transaction processing algorithm described previously is executed.

Additionally, we maintain a vector clock for distributed transactions to support consistent reads across data clusters [3]. The vector clock is formed using the batch numbers of the during which the transaction was prepared by the remote cluster. This allows data to be read from a consistent checkpoint.

In conclusion, we have briefly described WedgeDB, a distributed key-value store which supports efficient read-only transactions, serializable transaction processing, consistent reads across partitions and byzantine fault tolerance.

References

[1] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design*

and Implementation, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[2] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 529–540, Apr. 2013.

[3] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM. event-place: Copper Mountain, Colorado, USA.