Design and Properties of Transaction Processing System for Web Applications in the Cloud Computing

[1]Liladhar R. Rewatkar , [2]Ujwal A. Lanjewar

[1]Zulekha College of Commerce, Science and Technology, RTMNU Nagpur (MS)

[2]G. S. College of Commerce & Management Studies, Nagpur (MS)

leela_rewatkar@rediffmail.com, ualanjewar@gmail.com

Abstract: *A transaction is a set of queries to be executed atomically on a single consistent view of a database. The main challenge to support transactional guarantees in a cloud computing environment is to provide the ACID properties of Atomicity, Consistency, Isolation and Durability without compromising the scalability properties of the cloud. However, the underlying data storage services provide only eventual consistency. We address this problem by creating a secondary copy of the application data in the transaction managers that handle consistency. Cloud computing platforms provide scalability and high availability properties for web applications but they sacrifice data consistency at the same time. However, many applications cannot afford any data inconsistency. We present a scalable transaction manager for cloud database services to execute ACID transactions of web applications, even in the presence of server failures. We demonstrate the scalability of our system using a prototype implementation.*

We demonstrate the scalability of our transactional database service using a prototype implementation. Our system exploits two properties typical of Web applications to allow efficient and scalable operations. First, each transaction is encapsulated in the processing of a particular request from a user. Second, Web applications tend to issue transactions that span a relatively small number of well-

identified data items. This means that the two-phase commit protocol for any given transaction can be confined to a relatively small number of servers holding the accessed data items.

1. Introduction

Cloud computing offers the vision of a virtually infinite pool of computing, storage and networking resources where applications can be scalably deployed [1]. The scalability and high availability properties of Cloud platforms however come at a cost. First, the scalable database services offered by the cloud such as Amazon SimpleDB and Google BigTable allow data query only by primary key rather than supporting secondary-key or join queries [2, 3]. Second, these services provide only eventual data consistency: any data update becomes visible after a finite but undeterministic amount of time. As weak as this consistency property may seem, it does allow to build a wide range of useful applications, as demonstrated by the commercial success of Cloud computing platforms. However, many other applications such as payment services and online auction services cannot afford any data inconsistency.

Any centralized transaction manager would face two scalability problems: 1. A single transaction manager must execute all incoming transactions and would eventually become the performance bottleneck; 2) A single transaction manager must maintain a copy of all data accessed by transactions and would eventually run out of storage space. To support transactions in a scalable fashion, we propose to split the transaction manager into any number of Local Transaction Managers (LTMs) and to partition the application data and the load of transaction processing across LTMs.

A transactional system must maintain the ACID properties even in the case of server failures. For this, we replicate data items and transaction states to multiple LTMs, and periodically checkpoint consistent data snapshots to the cloud storage service. Consistency correctness relies on the eventual consistency and high availability properties of Cloud computing storage services: we need not worry about data loss or unavailability after a data update has been issued to the storage service.

2. System Design

Figure 1 shows the organization of our transactional system. Clients issue HTTP requests to a Web application, which in turn issues transactions to a Transaction Processing System (TPS). The TPS is composed of any number of LTMs, each of which is responsible for a subset of all data items. The Web application can submit a transaction to any LTM that is responsible for one of the accessed data items. This LTM then acts as the coordinator of the transaction across all LTMs in charge of the data items accessed by the transaction. The LTMs operate on an in-memory copy of the data items loaded from the cloud storage service. Data updates resulting from transactions are kept in memory of the LTMs and periodically checkpointed back to the cloud storage service.

We implement transactions using the two-phase commit protocol. In the first phase, the coordinator requests all involved LTMs and asks them to check that the operation can indeed been executed correctly. If all LTMs vote favorably, then the second phase actually commits the transaction. Otherwise, the transaction is aborted.

We assign data items to LTMs using consistent hashing [4]. To achieve a balanced assignment, we first cluster data items into virtual nodes, and then assign virtual nodes to LTMs. As shown in Figure 1, multiple virtual nodes can be assigned to the same LTM. To tolerate LTM failures, virtual nodes and transaction states are replicated to one or more LTMs. After an LTM server failure, the latest updates can then be recovered and affected transactions can continue execution while satisfying ACID properties.

We now detail the design of the TPS to guarantee the Atomicity, Consistency, Isolation and Durability properties of transactions. Each of the properties is discussed individually [5].

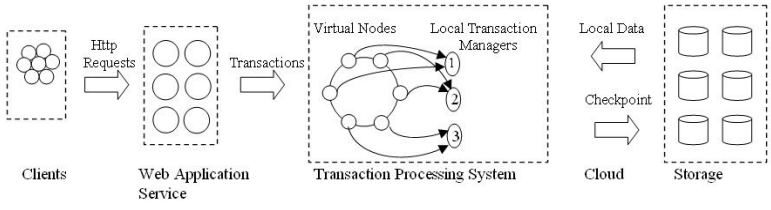

***Fig. 1**: System Model*

2.1 Atomicity

The Atomicity property requires that either all operations of a transaction complete successfully, or none of them do. To ensure Atomicity, for each transaction issued, our system performs two-phase commit (2PC) across all the LTMs responsible for the data items accessed. If an agreement of \COMMIT" is reached, the transaction coordinator can return the result to the web application without waiting for the completion of the second phase [6].

To ensure Atomicity in the presence of server failures, all transaction states and data items should be replicated to one or more LTMs. When an LTM fails, the transactions it was coordinating can be in two states. If a transaction has reached an agreement to "COMMIT," then it must eventually be committed; otherwise, the transaction can be aborted. Therefore, we replicate transaction states in two occasions: 1) When an LTM receives a new transaction, it must replicate the transaction state to other LTMs before confirming to the application that the transaction has been successfully submitted; 2) After all participant LTMs reach an agreement to "COMMIT" at the coordinator, the coordinator updates the transaction state at its backups. This creates in essence in-memory "redo logs" at the backup LTMs. The coordinator must finish this step before carrying out the second phase of the commit protocol. If the coordinator fails after this step, the backup LTM can then complete the second phase of the commit protocol. Otherwise, it can simply abort the transaction without violating the Atomicity property.

An LTM server failure also results in the inaccessibility of the data items it was responsible for. It is therefore necessary to re-replicate these data items to maintain N replicas. Once an LTM failure is detected, the failure detector issues a report to all LTMs so that they can carry out the recovery process and create a new consistent membership of the system. All incoming transactions that accessed the failed LTM are aborted during the recovery process. If a second LTM server failure happens during the recovery process of a previous LTM server failure, the system initiates the recovery of the second failure after the current recovery process has completed. The transactions that cannot recover from the first failure because they also

accessed the second failed LTM are left untouched until the second recovery process.

2.2 Consistency

The Consistency property requires that a transaction, which executes on a database that is internally consistent, will leave the database in an internally consistent state. Consistency is typically expressed as a set of declarative integrity constraints. We assume that the consistency rule is applied within the logic of transactions. Therefore, the Consistency property is satisfied as long as all transactions are executed correctly.

2.3 Isolation

The Isolation property requires that the behavior of a transaction is not impacted by the presence of other transactions that may be accessing the same data items concurrently. In the TPS, we decompose a transaction into a number of sub-transactions. Thus the Isolation property requires that if two transactions conflict on more than one data item, all of their conflicting sub-transactions must be executed sequentially, even though the sub-transactions are executed in multiple LTMs.

We apply timestamp ordering for globally ordering conflicting transactions across all LTMs. Each transaction has a globally unique timestamp, which is monotonically increasing with the time the transaction was submitted. All LTMs then order transactions as follows: a sub-transaction can execute only after all conflicting sub-transactions with a lower timestamp have committed. It may happen that a transaction is delayed (e.g., because of network delays) and that a conflicting sub-transaction with a younger timestamp has already committed. In this case, the older transaction should abort, obtain a new timestamp and restart the execution of all of its sub-transactions.

As each sub-transaction accesses only one data item by primary key, the implementation is straightforward. Each LTM maintains a list of sub-transactions for each data item it handles. The list is ordered by timestamp so LTMs can execute the sub-transactions sequentially in the timestamp order. The exception discussed before happens when an LTM inserts a sub-transaction to the list but finds its timestamp smaller than the one

currently being executed. It then reports the exception to the coordinator LTM of this transaction so that the whole transaction can be restarted. We extended the 2PC with an optional "RESTART" phase, which is triggered if any of the sub-transactions reports an ordering exception. After a transaction reached an agreement and enters the second phase of 2PC, it cannot be restarted any more.

2.4 Durability

The Durability property requires that the effects of committed transactions would not be undone and would survive server failures. In our case, it means that all the data updates of committed transactions must be successfully written back to the backend cloud storage service.

The main issue here is to support LTM failures without losing data. For performance reasons, the commit of a transaction does not directly update data in the cloud storage service but only updates the in-memory copy of data items in the LTMs. Instead, each LTM issues periodic updates to the cloud storage service. During the time between a transaction commit and the next checkpoint, durability is ensured by the replication of data items across several LTMs. After checkpoint, we can rely on the high availability and eventual consistency properties of the cloud storage service for durability. When an LTM server fails, all the data items stored in its memory that were not checkpointed yet are lost. However, as discussed in Section 2.1, all data items of the failed LTM can be recovered from the backup LTMs. The difficulty here is that the backups do not know which data items have already been checkpointed. One solution would be to checkpoint all recovered data items. However, this can cause a lot of unnecessary writes. One optimization is to record the latest checkpointed transaction timestamp of each data item and replicate these timestamps to the backup LTMs. We further cluster transactions into groups, then replicate timestamps only after a whole group of transactions has completed.

Another issue related to checkpointing is to avoid degrading the system performance at the time of a checkpoint. The checkpoint process must iterate through the latest updates of committed transactions and select the data items to be checkpointed. A naive implementation that would lock the

whole buffer during checkpointing would also block the concurrent execution of transactions. We address this problem by maintaining a bu_er in memory with the list of data items to be checkpointed. Transactions write to this buffer by sending updates to an unbounded non-blocking concurrent queue [6]. This data structure has the property of allowing multiple threads to write concurrently to the queue without blocking each other. Moreover, it orders elements in FIFO order, so old updates will not override younger ones.

2.5 Read-only Transactions

Our system supports read-write and read-only transactions indifferently. The only difference is that in read-only transactions no data item is updated during the second phase of 2PC. Read-only transactions have the same strong data consistency property, but also the same constraint: accessing well identified data items by primary key only. However, our system provides an additional feature for read-only transactions to support complex read queries such as range queries executed on a consistent snapshot of database.

We exploit the fact that many read queries can produce useful results by accessing an older but consistent data snapshot. For example, in e-commerce Web applications, a promotion service may identify the best seller items by aggregating recent orders information. However, it is not necessary to compute the result based on the absolute most recent orders. We therefore introduce the concept of Weakly-Consistent Read-only Transaction (WCRT), which is defined as follows: 1) A WCRT allows any type of read operations, including range queries; 2) WCRTs do not execute at the LTMs but access the latest checkpoint in the cloud storage service directly. A WCRT always executes on an internally consistent but possibly slightly outdated snapshot of the database. WCRTs are supported only if the underlying data storage service supports multi-versioning, as for example Bigtable [3]. To implement WCRTs, we introduce a snapshot mechanism in the checkpoint process, which marks each data update with a specific snapshot ID that is monotonically increasing. This ID is used as the version number of the new created version when it is written to the cloud storage service. A WCRT can thus access a specific snapshot by only reading the

latest version of any data item of which the timestamp is not larger than the snapshot ID.

3 Conclusion

Many Web applications need strong data consistency for their correct executions. However, although the high scalability and availability properties of the cloud make it a good platform to host Web content, scalable cloud database services only provide eventual consistency properties. This paper shows how one can support ACID transactions without compromising the scalability property of the cloud for web applications, even in the presence of server failures.

This work relies on few simple ideas. First, we load data from the cloud storage system into the transactional layer. Second, we split the data across any number of LTMs, and replicate them only for fault tolerance. Our system supports full ACID properties even in the presence of server failures, which only cause a temporary drop in throughput and a few aborted transactions.

References

[1] Hayes, B.: Cloud computing. Communications of the ACM 51, July 7, 2008, pp. 9-11

[2] Amazon.com: Amazon SimpleDB. http://aws.amazon.com/simpledb.

[3] Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable : a distributed storage system for structured data. In: Proc. OSDI., 2006 pp. 205-218

[4] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In: Proc. ACM Symposium on Theory of Computing., 1997, pp. 654-663

[5] Gray, J., Reuter, A. : Transaction Processing Concepts and Techniques. Morgan Kaufmann, 1993.

[6] Hvasshovd, S.O., Torbjornsen, O., Bratsberg, S.E., Holager, P.: The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response. In: Proc. VLDB. 1995, pp. 469-477

[7] Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. ACM symposium on Principles of distributed computing, 1996, pp. 267-275