Cloud Spanner: TrueTime and external consistency

TrueTime's benefits for Cloud Spanner

TrueTime is a highly available, distributed clock that is provided to applications on all Google servers[1] (#1). TrueTime enables applications to generate monotonically increasing timestamps: an application can compute a timestamp T that is guaranteed to be greater than any timestamp T' if T' finished being generated before T started being generated. This guarantee holds across all servers and all timestamps.

This feature of TrueTime is used by Cloud Spanner to assign timestamps to transactions. Specifically, every transaction is assigned a timestamp that reflects the instant at which Cloud Spanner considers it to have occurred. Because Cloud Spanner uses multi-version concurrency control, the ordering guarantee on timestamps enables clients of Cloud Spanner to perform consistent reads across an entire database (even across multiple Cloud regions (https://cloud.google.com/about/locations)) without blocking writes.

External consistency

Cloud Spanner provides clients with the strictest concurrency-control guarantees for transactions, which is called external consistency[2] (#2). Under external consistency, the system behaves as if all transactions were executed sequentially, even though Cloud Spanner actually runs them across multiple servers (and possibly in multiple datacenters) for higher performance and availability. In addition if one transaction completes before another transaction starts to commit, the system guarantees that clients can never see a state that includes the effect of the second transaction but not the first. Intuitively, Cloud Spanner is semantically indistinguishable from a single-machine database. Even though it provides such strong guarantees, Cloud Spanner enables applications to achieve performance comparable to databases that provide weaker guarantees (in return for higher performance). For example, like databases that support snapshot isolation, Cloud Spanner allows writes to proceed without being blocked by read-only transactions, but without exhibiting the anomalies that snapshot isolation allows.

External consistency greatly simplifies application development. For example, suppose that you have created a banking application on Cloud Spanner and one of your customers starts with $50 in their checking account and $50 in their savings account. Your application then begins a workflow in which it first commits a transaction T_1 to deposit $200 into the savings account, and then issues a second transaction T_2 to debit $150 from the checking account. Further, assume that at the end of the day, negative balances in one account are covered automatically from other accounts, and a customer incurs a penalty if the total balance across all their accounts is negative at any time during that day. External consistency guarantees that because T_2 starts to commit after T_1 finishes, then all readers of the database will observe that the deposit T_1 occurred before the debit T_2. Put another way, external consistency guarantees that no one will ever see a state where T_2 occurs prior to T_1; in other words, the debit will never incur a penalty due to insufficient funds.

A traditional database that uses strict two-phase locking provides external consistency. Unfortunately, in such a system, every time your application wants to read the most current data (which we call a "strong read"), the system acquires a read lock on the data, which blocks writes to the data being read.

Timestamps and multi-version concurrency control (MVCC)

To read without blocking writes, Cloud Spanner and many other database systems keep multiple immutable versions of data (often called multi-version concurrency control). A write creates a new immutable version whose timestamp is that of the write's transaction. A "snapshot read" at a timestamp returns the value of the most recent version prior to that timestamp, and does not need to block writes. It is therefore important that the timestamps assigned to versions be consistent with the order in which transactions can be observed to commit. We call this property "proper timestamping"; note that the existence of a proper timestamping is equivalent to external consistency.

To see why proper timestamping is important, consider the banking example from the previous section. Without proper timestamping, T_2 could be assigned a timestamp that is earlier than the timestamp assigned to T_1 (for example, if a hypothetical system used local clocks instead of TrueTime, and the clock of the server that processes T_2 lagged slightly). A snapshot read could then reflect the debit from T_2 but not the deposit T_1, even though the customer saw the deposit finish before starting the debit.

Achieving proper timestamping is trivial for a single-machine database (for example, you can just assign timestamps from a global, monotonically increasing counter). Achieving it in a widely distributed system such as Cloud Spanner, in which servers all over the world need to assign timestamps, is much more difficult to do efficiently.

Cloud Spanner depends on TrueTime to generate monotonically increasing timestamps. Cloud Spanner uses these timestamps in two ways. First, it uses them as proper timestamps for write transactions without the need for global communication. Second, it uses them as timestamps for strong reads, which enables strong reads to execute in one round of communication, even strong reads that span multiple servers.

FAQ

What consistency guarantees does Cloud Spanner provide?

Cloud Spanner provides external consistency, which is the strictest consistency property for transaction-processing systems. All transactions in Cloud Spanner satisfy this consistency property, not just those within a partition. External consistency states that Cloud Spanner executes transactions in a manner that is indistinguishable from a system in which the transactions are executed serially, and furthermore, that the serial order is consistent with the order in which transactions can be observed to commit. Because the timestamps generated for transactions correspond to the serial order, if any client sees a transaction T_2 start to commit after another transaction T_1 finishes, the system will assign a timestamp to T_2 that is higher than T_1's timestamp.

Does Cloud Spanner provide linearizability?

Yes. In fact, Cloud Spanner provides external consistency, which is a stronger property than linearizability, because linearizability does not say anything about the behavior of transactions. Linearizability is a property of concurrent objects that support atomic read and write operations. In a database, an "object" would typically be a single row or even a single cell. External consistency is a property of transaction-processing systems, where clients dynamically synthesize transactions that contain multiple read and write operations on arbitrary objects. Linearizability can be viewed as a special case of external consistency, where a transaction can only contain a single read or write operation on a single object.

Does Cloud Spanner provide serializability?

Yes. In fact, Cloud Spanner provides external consistency, which is a stricter property than serializability. A transaction-processing system is serializable if it executes transactions in a manner that is indistinguishable from a system in which the transactions are executed serially. Cloud Spanner also guarantees that the serial order is consistent with the order in which the transactions can be observed to commit.

Consider again the banking example used earlier. In a system that provides serializability but not external consistency, even though the customer executed T_1 and then T_2 sequentially, the system would be permitted to reorder them, which could cause the debit to incur a penalty due to insufficient funds.

Does Cloud Spanner provide strong consistency?

Yes. In fact, Cloud Spanner provides external consistency, which is a stronger property than strong consistency. The default mode for reads in Cloud Spanner is "strong", which guarantees that they observe the effects of all transactions that committed before the start of the operation, independent of which replica receives the read.

What is the difference between strong consistency and external consistency?

A replication protocol exhibits "strong consistency" if the replicated objects are linearizable. Like linearizability, "strong consistency" is weaker than "external consistency", because it does not say anything about the behavior of transactions.

Does Cloud Spanner provide eventual (or lazy) consistency?

Cloud Spanner provides external consistency, which is a much stronger property than eventual consistency. Eventual consistency trades weaker guarantees for higher performance. Eventual consistency is problematic because it means that readers can observe the database in a state that it was never truly in (e.g., a read could observe a state where Transaction B is committed but Transaction A is not, even though A happened before B). Cloud Spanner provides stale reads (https://cloud.google.com/spanner/docs/timestamp-bounds#bounded_staleness), which offer similar performance benefits as eventual consistency but with much stronger consistency guarantees. A stale read returns data from an "old" timestamp, which cannot block writes because old versions of data are immutable.

Further reading

- Cloud Spanner transaction semantics (https://cloud.google.com/spanner/docs/transactions#rw_transaction_semantics)
- Spanner, TrueTime, and the CAP Theorem (https://research.google.com/pubs/pub45855.html)

Notes

- [1]*J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database* (https://research.google.com/archive/spanner.html)*. In Tenth USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), pp. 261–264, Hollywood, CA, Oct. 2012.*

- [2]*Gifford, D. K. Information Storage in a Decentralized Computer System. PhD thesis, Stanford University, 1981.*