6.814 - Lecture 14 Notes

Long Nguyen

October 31, 2017

1 Topics

Today, we'll talk about locking granularity, degrees of isolation, and an introduction to recovery.

How do we build a system with transactional abstraction that's durable—in the sense that when the system crashes, it can return to its old state?

2 Locking granularity

So far, we've talked of RX/WX transactions. What is X? It can be a record, page, table, etc. These are granularities.

Why does this matter? Record-level locking carries overhead, while page/table locking potentially blocks readers or writers that might not even have conflicts (false sharing).

In SimpleDB, we use page-level locking.

In real systems, we'd like to work with more clever policies. Perhaps we can support multiple granularities in our implementation. This isn't trivial in practice, of course.

2.1 Intention locks

Locking hierarchy: tables $\rightarrow$ pages $\rightarrow$ tuples.

We create new types of locks, separate from S and X locks. The idea is to acquire (at least attempt to, otherwise block) an IS or IX "intention" lock on all levels in the hierarchy above Object Z. Suppose we want to read some tuple t. Then, we have to acquire IS locks on both the page containing t and the table containing that page.

Consider the following table, which details all new lock compatibilities.

Table 1: *Lock compatibilities.*

	S	***X***	***IX***	***IS***
S	Y	N	N	Y
X	N	N	N	N
IX	N	N	Y	Y
IS	Y	N	Y	Y

3 Degrees of isolation (relaxed consistency)

We don't always need a system that enforces serializability (rigid consistency).

3.1 Read-committed

Something we might like is the **read-committed** property, meaning that all reads see committed data, but some values can change. This buys us more concurrency, since we don't necessarily have to hold some S locks. The disadvantage is violation of serializability.

3.2 Repeatable read

This property states that all reads of an object have the same value. All reads are committed. This is actually *not* identical to serializability.

What if an object appears in between transactions? Suppose a transaction scans a range and returns the value read. However, suppose afterwards, another tuple is added. Then, the first transaction's function call, if run again, gives a different value.

Problem: With fine-granularity locking, new objects can appear—"phantom problem".

To solve the phantom problem, we can do a few things:

- lock the tables,
- lock a range, e.g. employees with some salary threshold,
- next-key locking.

Range locks are difficult. Next-key locking usually makes use of B-trees and prevents transactions from inserting any new pages (possible values) between two leaf nodes/pages of the B-tree (essentially a placeholder for missing values).

3.3 Snapshot isolation

This is an optimistic concurrency control variant where read sets are not tracked (only checking for write conflicts). It does lead to anomalies, but interestingly enough, is usually the default implementation in most databases (older Postgres versions, Oracle, SQL Server).

4 Introduction to recovery (or crash recovery)

During recovery, we'd like to ensure that neither of these is an issue:

1. aborted/uncommitted actions aren't visible after a crash,
2. committed actions are durable, e.g. are visible after a crash.

We care that transactions are still consistent after a system failure.

Normally, when a crash happens, all volatile memory where operations were happening is wiped. Disk, as it is nonvolatile, remains as is. How can an aborted action still be seen? *An uncommitted state was flushed to disk.* Conversely, if we come back after a crash and committed actions aren't durable, then *when we committed, we didn't push these writes back to disk.*

Let's introduce the notion of a **log**, in which we keep track of transaction states.

4.1 Write-ahead logging

In WAL, we guarantee that log records for operations are written to the log before updating database pages on disk.

At commit time for some transaction T, we write log records for T. We can think of the log as a record of *how we plan to change the database.*

WAL allows us to, given sufficiently written states, address both issues presented above for recovery. We can use info from the log records to make changes as necessary.

A log is append-only and is sequential.

4.1.1 What to log

We need to log

- all writes,
- updates, with object and its changed value (REDO state)
- old values of object (UNDO state),
- start, commit, abort.

Next time, we'll discuss the ARIES property and how it works.