Lecture 18: Index Concurrency Control

15-445/645 Database Systems (Fall 2017)
Carnegie Mellon University
Prof. Andy Pavlo

The Phantom Problem

1. An update in one transaction can change the result of serial scans in another transaction (**a tuple can appear or disappear like a phantom**).

2. Conflict serializability on reads and writes of individual items guarantees serializability **only** if the set of objects is fixed.

3. One Solution: **Predicate Locking**

   (a) Lock records that satisfy a logical predicate.

   (b) Originally proposed as a fix to the Phantom Problem by IBM in the 1970s.

   (c) Predicate locking has a lot of locking overhead. In practice, nobody does this.

Index Locking

Index locking is a special case of predicate locking that is potentially more efficient for preventing the phantom problem.

1. If there is a dense index on the rating field then the transaction can lock index page containing the data with some predicate.

2. If there are no records with some predicate, the transaction must lock the index page where such a data entry would be, if it existed.

3. If there is no suitable index, the transaction must obtain:

   (a) A lock on every page in the table to prevent a record's data to being changed to satisfy or unsatisfy the predicate.

   (b) The lock for the table itself to prevent records with some predicate from being added or deleted.

4. Repeating scans is another alternative:

   (a) An alternative is to just re-execute every scan again when the transaction commits and check whether it gets the same result.

   (b) If you get the same result, then there were no phantoms and you can commit the transaction.

   (c) Drawback: you must retain the scan set for every range query in a transaction.

Weaker Levels of Isolation

Serializability is useful because it allows programmers to ignore concurrency issues. But enforcing it may allow too little concurrency and limit performance. Thus, the application may want to use a weaker level of consistency to improve scalability.

Isolation Level: Controls the extent that a transaction is exposed to the actions of other concurrent transactions. Provides for greater concurrency at the cost of exposing transactions to uncommitted changes. Not all DBMS support all isolation levels in all execution scenarios. You can also provide hints to the DBMS about whether a transaction will modify the database using the modes (`READ WRITE` and `READ ONLY`).

The SQL-92 standard defines the following isolation levels:

1. **`SERIALIZABLE`**
   (a) No phantoms, all read repeatable, no dirty reads.
   (b) Implementation: Obtain all locks first, plus index locks, plus 2PL.
2. **`REPEATABLE READS`**
   (a) Phantoms may happen.
   (b) Implementation: Same as above, but no index locks.
3. **`READ COMMITTED`**
   (a) Phantoms and unrepeatable reads may happen.
   (b) Implementation: Same as above, but `S` locks are released immediately.
4. **`READ UNCOMMITTED`**
   (a) Phantoms, unrepeatable reads, and dirty reads may happen.
   (b) Implementation: Same as above, but allows dirty reads (no `S` locks).

Locking in B+Trees

The logical contents of the index is the only thing we care about. They are not quite like other database elements so we can treat them differently. **It's okay to have non-serializable concurrent access to an index as long as the accuracy of the index is maintained**. Using 2PL on an index prevents other transactions from using the index because a lock on the B+tree root is acquired.

Lock crabbing is a protocol to allow multiple threads to access/modify B+Tree at the same time: (1) Get lock for parent. (2) Get lock for child. (3) Release lock for parent if "safe" A **safe node** is one that will not split or merge when updated (not full on insertion or more than half full on deletion).

1. **Basic Lock Crabbing**
   (a) **Search:** Start at root and go down, repeatedly acquire lock on child and then unlock parent.
   (b) **Insert/Delete:** Start at root and go down, obtaining `X` locks as needed. Once child is locked, check if it is safe. If the child is safe, release locks on all its ancestors.

2. **Improved Lock Crabbing (Optimistic Lock Coupling):** The problem with the basic lock crabbing algorithm is that transactions always acquire an exclusive lock on the root for every insert/delete operation. This greatly limits parallelism. Instead, we can assume that having to resize (i.e., split/merge nodes) is rare, and thus transactions can acquire shared locks down to the leaf nodes. Each transaction will assume that the path to the target leaf node is safe, and use `S-locks` and crabbing to reach it, and verify. If any node in the path is not safe, then do previous algorithm (i.e., acquire `X-locks`).

   (a) **Search:** Same algorithm as before.

   (b) **Insert/Delete:** Set `S-locks` as if for search, go to leaf, and set `X-lock` on leaf. If leaf is not safe, release all previous locks, and restart transaction using previous Insert/Delete protocol.

Conclusion

1. Indexes make concurrency control hard(er) because it's essentially a second copy of the data.

2. Most applications do not execute with `SERIALIZABLE` isolation.