Transaction Concurrency Methods

Lars Frank
Copenhagen Business School, Denmark

INTRODUCTION

In this article, I will evaluate the most important methods for increasing concurrency between transactions. Because different transactions have different needs, it is important that the evaluation gives an overview of the advantages and disadvantages of the different optimization methods.

It is possible to increase concurrency between database transactions and minimize the time in which conflicts may occur by splitting major transactions into minor transactions performing the same operations. For short, I will call this concurrency optimization technique short-duration locking, even though not all concurrency control methods use locking. Concurrency may also be increased by using by using low isolation levels, in which certain types of conflicts between transactions are accepted. In practice, either two-phase locking or optimistic concurrency control is used for concurrency control. When one of these concurrency control methods is chosen, the only way to increase concurrency further is to relax the isolation property. In practice, the isolation property is relaxed by using low isolation levels, short duration locks, or both. Therefore, these optimization methods are analyzed, too.

By using the results of my analyses, it should be easier to increase concurrency between database transactions. This is especially important in long-lived transactions, hotspots, and multidatabases in which the traditional concurrency control methods often have to give up in practice. However, when either short duration locking or low isolation levels optimize concurrency, the traditional atomicity, consistency, isolation, and durability (ACID) properties are normally lost. The problems caused by a lack of ACID properties may be managed by using approximated ACID properties (i.e., from an application point of view the system should function as if all the traditional ACID properties had been implemented).

When the isolation property is relaxed, the following isolation anomalies may occur (Berenson et al., 1995; Breibart, 1992):

- **The lost update anomaly** is by definition a situation in which a first transaction reads a record for update without using locks. After this, the record is updated by another transaction. Later, the update is overwritten by the first transaction. In practice, the read and the update of a record is often executed in different subtransactions to prevent locking the record across a dialogue with the user. In this situation, it is possible for a second transaction to update the record between the read and the update of the first transaction. If countermeasures are not used, the update of the second transaction may be lost.
- **The dirty read anomaly** is by definition a situation in which a first transaction updates a record without committing the update. After this, a second transaction reads the record. Later, the first update is aborted (i.e., the second transaction may have read a nonexistent version of the record). In extended transaction models, this may happen when the first transaction updates a record by using a compensatable subtransaction and later aborts the update by using a compensating subtransaction. If a second transaction reads the record before it is compensated, the data read will be “dirty.”
- **The nonrepeatable read anomaly** or fuzzy read is by definition a situation in which a first transaction reads a record without using long duration locks. This record is later updated and committed by a second transaction before the first transaction is committed or aborted. In other words, one cannot rely on what one has read. In extended transaction models, this may happen when the first transaction reads a record updated by a second transaction, which commits the record locally before the first transaction commits it globally.
- **The phantom anomaly** is normally not important and, therefore, I do not deal with it in this article.

BACKGROUND

Different concurrency methods and isolation levels have been analyzed and evaluated (e.g., Gray & Reuter, 1993), and short duration locks have been used in practice for many years. However, to my knowledge, short duration locks have not been evaluated and compared to other methods in terms of increasing concurrency between transactions.
The transaction model described in the next section is the countermeasure transaction model described by Frank and Zahle (1998), Frank (1999, 2004), and Frank and Kofod (2002). This model owes many of its properties to, for example, Garcia-Molina and Salem (1987); Mehrotra, Rastogi, Korth, and Silberschatz (1992); Weikum and Schek (1992); and Zhang, Nodine, Bhargava, and Bukhres (1994). The countermeasure transaction model (Frank & Zahle, 1998) is important because it is the first systematic attempt to find countermeasures against isolation anomalies caused by using short duration locking. In this article, I have generalized this idea and included new countermeasures that especially apply to isolation level anomalies.

Kempster, Stirling, and Thanisch (1999) have described isolation anomalies in detail. The objective of this article is not to evaluate concurrency control methods as such, as this has been done by Thomasian (1998), for example.

In the following, I will first describe how the atomicity property can be implemented by using extended transaction models. This is important if short duration locks are used. Next, I will describe and evaluate some of the methods most used to increase concurrency between transactions. The results of the evaluation will either be explained or adopted from references.

**ATOMICITY IMPLEMENTATION FOR TRANSACTIONS THAT USE SHORT DURATION LOCKS**

An updating transaction has the atomicity property and is called atomic if either all or none of its updates are executed. If short duration locks are used, the atomicity property cannot be implemented automatically by the DBMS, and therefore extended transaction models are recommended. In extended transaction models, a global transaction consists of a root transaction (client transaction) and several single-site subtransactions (server transactions). The subtransactions can be nested transactions (i.e., a subtransaction may be a parent transaction for other subtransactions). In the following, I will give a broad outline of how the atomicity property is implemented in extended transaction models.

In extended transaction models, a global transaction is partitioned into the following types of subtransactions that may be executed in different locations:

- **The Pivot** subtransaction manages the atomicity of the global transaction (i.e., the global transaction is committed when the pivot subtransaction is committed locally). If the pivot subtransaction aborts, all the updates of the other subtransactions must be compensated or not be executed.
- **Compensatable** subtransactions, that all may be compensated. Compensatable subtransactions must always be executed *before* the pivot subtransaction, to make it possible to compensate them if the pivot subtransaction cannot be committed. Compensation is achieved by executing a compensating subtransaction.
- **Retriable** subtransactions are designed so that the execution is guaranteed to commit locally (sooner or later) if the pivot subtransaction is committed. An update propagation tool is used to automatically resubmit the request for execution until the subtransaction has been committed locally (i.e., the update propagation tool is used to force the retrievable subtransaction to be executed).

Executing compensatable, pivot, and retrievable subtransactions, in that order, implements the global atomicity and atomicity property of a transaction (Breibart et al., 1992).

**METHODS TO INCREASE CONCURRENCY**

In this section, I will analyze and evaluate pessimistic concurrency control, optimistic concurrency control, low isolation levels, and short duration locks as different methods to increase concurrency. Table 1 gives an overview of the properties of the methods used to increase concurrency.

**Pessimistic Concurrency Control**

Locking protocols implements serializability by using locks (i.e., each transaction locks the data it uses to prevent other transactions from accessing its data). Locking protocols can be described as pessimistic, because they assume that data used by one transaction may be needed by other transactions and, therefore, it is better to lock the data.

Pessimistic concurrency control can be optimized by using shared and exclusive locks for reading and writing/updating, respectively. All locks should have long duration (i.e., they must be held until the transaction has been committed). However, the designer of the transaction may divide it into subtransactions that manage the locks independently of the parent transaction. In this way, the programmer can force short duration locks to be used. If short duration locks are used, approximated ACID properties should be used, as described previ-