Concurrency control

In information technology and computer science, especially in the fields of computer programming, operating systems, multiprocessors, and databases, **concurrency control** ensures that correct results for concurrent operations are generated, while getting those results as quickly as possible.

Computer systems, both software and hardware, consist of modules, or components. Each component is designed to operate correctly, i.e., to obey or to meet certain consistency rules. When components that operate concurrently interact by messaging or by sharing accessed data (in memory or storage), a certain component's consistency may be violated by another component. The general area of concurrency control provides rules, methods, design methodologies, and theories to maintain the consistency of components operating concurrently while interacting, and thus the consistency and correctness of the whole system. Introducing concurrency control into a system means applying operation constraints which typically result in some performance reduction. Operation consistency and correctness should be achieved with as good as possible efficiency, without reducing performance below reasonable levels. Concurrency control can require significant additional complexity and overhead in a concurrent algorithm compared to the simpler sequential algorithm.

For example, a failure in concurrency control can result in data corruption from torn read or write operations.

Contents

Concurrency control in databases
- Database transaction and the ACID rules
- Why is concurrency control needed?
- Concurrency control mechanisms
  - Categories
  - Methods
- Major goals of concurrency control mechanisms
  - Correctness
    - Serializability
    - Recoverability
  - Distribution
    - Distributed serializability and commitment ordering
    - Distributed recoverability
  - Other major subjects of attention
    - Recovery
    - Replication
- See also
- References
  - Citations

Concurrency control in operating systems
- See also
- References

Concurrency control in databases

Comments:

1. This section is applicable to all transactional systems, i.e., to all systems that use *database transactions* (*atomic transactions*; e.g., transactional objects in Systems management and in networks of smartphones which typically implement private, dedicated database systems), not only general-purpose database management systems (DBMSs).
2. DBMSs need to deal also with concurrency control issues not typical just to database transactions but rather to operating systems in general. These issues (e.g., see *Concurrency control in operating systems* below) are out of the scope of this section.

Concurrency control in Database management systems (DBMS; e.g., Bernstein et al. 1987, Weikum and Vossen 2001), other transactional objects, and related distributed applications (e.g., Grid computing and Cloud computing) ensures that *database transactions* are performed concurrently without violating the data integrity of the respective databases. Thus concurrency control is an essential element for correctness in any system where two database transactions or more, executed with time overlap, can access the same data, e.g., virtually in any general-purpose database system. Consequently, a vast body of related research has been accumulated since database systems emerged in the early 1970s. A well established concurrency control theory for database systems is outlined in the references mentioned above: serializability theory, which allows to effectively design and analyze concurrency control methods and mechanisms. An alternative theory for concurrency control of atomic transactions over abstract data types is presented in (Lynch et al. 1993), and not utilized below. This theory is more refined, complex, with a wider scope, and has been less utilized in the Database literature than the classical theory above. Each theory has its pros and cons, emphasis and insight. To some extent they are complementary, and their merging may be useful.

To ensure correctness, a DBMS usually guarantees that only *serializable* transaction schedules are generated, unless *serializability* is intentionally relaxed to increase performance, but only in cases where application correctness is not harmed. For maintaining correctness in cases of failed (aborted) transactions (which can always happen for many reasons) schedules also need to have the *recoverability* (from abort) property. A DBMS also guarantees that no effect of *committed* transactions is lost, and no effect of *aborted* (rolled back) transactions remains in the related database. Overall transaction characterization is usually summarized by the ACID rules below. As databases have become distributed, or needed to cooperate in distributed environments (e.g., Federated databases in the early 1990, and Cloud computing currently), the effective distribution of concurrency control mechanisms has received special attention.

Database transaction and the ACID rules

The concept of a *database transaction* (or *atomic transaction*) has evolved in order to enable both a well understood database system behavior in a faulty environment where crashes can happen any time, and *recovery* from a crash to a well understood database state. A database transaction is a unit of work, typically encapsulating a number of operations over a database (e.g., reading a database object, writing, acquiring lock, etc.), an abstraction supported in database and also other systems. Each transaction has well defined boundaries in terms of which program/code executions are included in that transaction (determined by the transaction's programmer via special transaction commands). Every database transaction obeys the following rules (by support in the database system; i.e., a database system is designed to guarantee them for the transactions it runs):

- **Atomicity** - Either the effects of all or none of its operations remain ("all or nothing" semantics) when a transaction is completed (*committed* or *aborted* respectively). In other words, to the outside world a committed transaction appears (by its effects on the database) to be indivisible

(atomic), and an aborted transaction does not affect the database at all. Either all the operations are done or none of them are.

- **Consistency** - Every transaction must leave the database in a consistent (correct) state, i.e., maintain the predetermined integrity rules of the database (constraints upon and among the database's objects). A transaction must transform a database from one consistent state to another consistent state (however, it is the responsibility of the transaction's programmer to make sure that the transaction itself is correct, i.e., performs correctly what it intends to perform (from the application's point of view) while the predefined integrity rules are enforced by the DBMS). Thus since a database can be normally changed only by transactions, all the database's states are consistent.
- **Isolation** - Transactions cannot interfere with each other (as an end result of their executions). Moreover, usually (depending on concurrency control method) the effects of an incomplete transaction are not even visible to another transaction. Providing isolation is the main goal of concurrency control.
- **Durability** - Effects of successful (committed) transactions must persist through crashes (typically by recording the transaction's effects and its commit event in a non-volatile memory).

The concept of atomic transaction has been extended during the years to what has become Business transactions which actually implement types of Workflow and are not atomic. However also such enhanced transactions typically utilize atomic transactions as components.

Why is concurrency control needed?

If transactions are executed *serially*, i.e., sequentially with no overlap in time, no transaction concurrency exists. However, if concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected, undesirable results may occur, such as:

1. The lost update problem: A second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is lost to other transactions running concurrently which need, by their precedence, to read the first value. The transactions that have read the wrong value end with incorrect results.
2. The dirty read problem: Transactions read a value written by a transaction that has been later aborted. This value disappears from the database upon abort, and should not have been read by any transaction ("dirty read"). The reading transactions end with incorrect results.
3. The incorrect summary problem: While one transaction takes a summary over the values of all the instances of a repeated data-item, a second transaction updates some instances of that data-item. The resulting summary does not reflect a correct result for any (usually needed for correctness) precedence order between the two transactions (if one is executed before the other), but rather some random result, depending on the timing of the updates, and whether certain update results have been included in the summary or not.

Most high-performance transactional systems need to run transactions concurrently to meet their performance requirements. Thus, without concurrency control such systems can neither provide correct results nor maintain their databases consistently.

Concurrency control mechanisms

Categories

The main categories of concurrency control mechanisms are:

- **Optimistic** - Delay the checking of whether a transaction meets the isolation and other integrity rules (e.g., serializability and recoverability) until its end, without blocking any of its (read, write) operations ("...and be optimistic about the rules being met..."), and then abort a transaction to prevent the violation, if the desired rules are to be violated upon its commit. An aborted transaction is immediately restarted and re-executed, which incurs an obvious overhead (versus executing it to the end only once). If not too many transactions are aborted, then being optimistic is usually a good strategy.
- **Pessimistic** - Block an operation of a transaction, if it may cause violation of the rules, until the possibility of violation disappears. Blocking operations is typically involved with performance reduction.
- **Semi-optimistic** - Block operations in some situations, if they may cause violation of some rules, and do not block in other situations while delaying rules checking (if needed) to transaction's end, as done with optimistic.

Different categories provide different performance, i.e., different average transaction completion rates (*throughput*), depending on transaction types mix, computing level of parallelism, and other factors. If selection and knowledge about trade-offs are available, then category and method should be chosen to provide the highest performance.

The mutual blocking between two transactions (where each one blocks the other) or more results in a deadlock, where the transactions involved are stalled and cannot reach completion. Most non-optimistic mechanisms (with blocking) are prone to deadlocks which are resolved by an intentional abort of a stalled transaction (which releases the other transactions in that deadlock), and its immediate restart and re-execution. The likelihood of a deadlock is typically low.

Blocking, deadlocks, and aborts all result in performance reduction, and hence the trade-offs between the categories.

Methods

Many methods for concurrency control exist. Most of them can be implemented within either main category above. The major methods,[1] which have each many variants, and in some cases may overlap or be combined, are:

1. Locking (e.g., **Two-phase locking** - 2PL) - Controlling access to data by locks assigned to the data. Access of a transaction to a data item (database object) locked by another transaction may be blocked (depending on lock type and access operation type) until lock release.
2. **Serialization graph checking** (also called Serializability, or Conflict, or Precedence graph checking) - Checking for cycles in the schedule's graph and breaking them by aborts.
3. **Timestamp ordering** (TO) - Assigning timestamps to transactions, and controlling or checking access to data by timestamp order.
4. **Commitment ordering** (or Commit ordering; CO) - Controlling or checking transactions' chronological order of commit events to be compatible with their respective precedence order.

Other major concurrency control types that are utilized in conjunction with the methods above include:

- **Multiversion concurrency control** (MVCC) - Increasing concurrency and performance by generating a new version of a database object each time the object is written, and allowing transactions' read operations of several last relevant versions (of each object) depending on scheduling method.
- **Index concurrency control** - Synchronizing access operations to indexes, rather than to user data. Specialized methods provide substantial performance gains.

- **Private workspace model** (**Deferred update**) - Each transaction maintains a private workspace for its accessed data, and its changed data become visible outside the transaction only upon its commit (e.g., Weikum and Vossen 2001). This model provides a different concurrency control behavior with benefits in many cases.

The most common mechanism type in database systems since their early days in the 1970s has been *Strong strict Two-phase locking* (SS2PL; also called *Rigorous scheduling* or *Rigorous 2PL*) which is a special case (variant) of both Two-phase locking (2PL) and Commitment ordering (CO). It is pessimistic. In spite of its long name (for historical reasons) the idea of the **SS2PL** mechanism is simple: "Release all locks applied by a transaction only after the transaction has ended." SS2PL (or Rigorousness) is also the name of the set of all schedules that can be generated by this mechanism, i.e., these are SS2PL (or Rigorous) schedules, have the SS2PL (or Rigorousness) property.

Major goals of concurrency control mechanisms

Concurrency control mechanisms firstly need to operate correctly, i.e., to maintain each transaction's integrity rules (as related to concurrency; application-specific integrity rule are out of the scope here) while transactions are running concurrently, and thus the integrity of the entire transactional system. Correctness needs to be achieved with as good performance as possible. In addition, increasingly a need exists to operate effectively while transactions are distributed over processes, computers, and computer networks. Other subjects that may affect concurrency control are recovery and replication.

Correctness

Serializability

For correctness, a common major goal of most concurrency control mechanisms is generating schedules with the *Serializability* property. Without serializability undesirable phenomena may occur, e.g., money may disappear from accounts, or be generated from nowhere. **Serializability** of a schedule means equivalence (in the resulting database values) to some *serial* schedule with the same transactions (i.e., in which transactions are sequential with no overlap in time, and thus completely isolated from each other: No concurrent access by any two transactions to the same data is possible). Serializability is considered the highest level of isolation among database transactions, and the major correctness criterion for concurrent transactions. In some cases compromised, relaxed forms of serializability are allowed for better performance (e.g., the popular *Snapshot isolation* mechanism) or to meet availability requirements in highly distributed systems (see *Eventual consistency*), but only if application's correctness is not violated by the relaxation (e.g., no relaxation is allowed for money transactions, since by relaxation money can disappear, or appear from nowhere).

Almost all implemented concurrency control mechanisms achieve serializability by providing *Conflict serializablity*, a broad special case of serializability (i.e., it covers, enables most serializable schedules, and does not impose significant additional delay-causing constraints) which can be implemented efficiently.

Recoverability

See *Recoverability* in *Serializability*

Comment: While in the general area of systems the term "recoverability" may refer to the ability of a system to recover from failure or from an incorrect/forbidden state, within concurrency control of database systems this term has received a specific meaning.

Concurrency control typically also ensures the *Recoverability* property of schedules for maintaining correctness in cases of aborted transactions (which can always happen for many reasons). **Recoverability** (from abort) means that no committed transaction in a schedule has read data written by an aborted transaction. Such data disappear from the database (upon the abort) and are parts of an incorrect database state. Reading such data violates the consistency rule of ACID. Unlike Serializability, Recoverability cannot be compromised, relaxed at any case, since any relaxation results in quick database integrity violation upon aborts. The major methods listed above provide serializability mechanisms. None of them in its general form automatically provides recoverability, and special considerations and mechanism enhancements are needed to support recoverability. A commonly utilized special case of recoverability is *Strictness*, which allows efficient database recovery from failure (but excludes optimistic implementations; e.g., Strict CO (SCO) cannot have an optimistic implementation, but has semi-optimistic ones).

Comment: Note that the *Recoverability* property is needed even if no database failure occurs and no database *recovery* from failure is needed. It is rather needed to correctly automatically handle transaction aborts, which may be unrelated to database failure and recovery from it.

Distribution

With the fast technological development of computing the difference between local and distributed computing over low latency networks or buses is blurring. Thus the quite effective utilization of local techniques in such distributed environments is common, e.g., in computer clusters and multi-core processors. However the local techniques have their limitations and use multi-processes (or threads) supported by multi-processors (or multi-cores) to scale. This often turns transactions into distributed ones, if they themselves need to span multi-processes. In these cases most local concurrency control techniques do not scale well.

Distributed serializability and commitment ordering

See *Distributed serializability* in *Serializability*

As database systems have become distributed, or started to cooperate in distributed environments (e.g., Federated databases in the early 1990s, and nowadays Grid computing, Cloud computing, and networks with smartphones), some transactions have become distributed. A distributed transaction means that the transaction spans processes, and may span computers and geographical sites. This generates a need in effective distributed concurrency control mechanisms. Achieving the Serializability property of a distributed system's schedule (see *Distributed serializability* and *Global serializability* (*Modular serializability*)) effectively poses special challenges typically not met by most of the regular serializability mechanisms, originally designed to operate locally. This is especially due to a need in costly distribution of concurrency control information amid communication and computer latency. The only known general effective technique for distribution is Commitment ordering, which was disclosed publicly in 1991 (after being patented). **Commitment ordering** (Commit ordering, CO; Raz 1992) means that transactions' chronological order of commit events is kept compatible with their respective precedence order. CO does not require the distribution of concurrency control information and provides a general effective solution (reliable, high-performance, and scalable) for both distributed and global serializability, also in a heterogeneous environment with database systems (or other transactional objects) with different (any) concurrency control mechanisms.[1] CO is indifferent to which mechanism is utilized, since it does not interfere with any transaction operation scheduling (which most mechanisms control), and only determines the order of commit events. Thus, CO enables the efficient distribution of all other mechanisms, and also the distribution of a mix of different (any) local mechanisms, for achieving distributed and global serializability. The existence of such a solution has been considered "unlikely" until 1991, and by many experts also later, due to misunderstanding of the CO solution (see Quotations in *Global serializability*). An important side-benefit of CO is automatic distributed deadlock resolution. Contrary to CO, virtually all other techniques (when not combined with CO) are prone to distributed deadlocks (also

called global deadlocks) which need special handling. CO is also the name of the resulting schedule property: A schedule has the CO property if the chronological order of its transactions' commit events is compatible with the respective transactions' precedence (partial) order.

SS2PL mentioned above is a variant (special case) of CO and thus also effective to achieve distributed and global serializability. It also provides automatic distributed deadlock resolution (a fact overlooked in the research literature even after CO's publication), as well as Strictness and thus Recoverability. Possessing these desired properties together with known efficient locking based implementations explains SS2PL's popularity. SS2PL has been utilized to efficiently achieve Distributed and Global serializability since the 1980, and has become the de facto standard for it. However, SS2PL is blocking and constraining (pessimistic), and with the proliferation of distribution and utilization of systems different from traditional database systems (e.g., as in Cloud computing), less constraining types of CO (e.g., Optimistic CO) may be needed for better performance.

Comments:

1. The *Distributed conflict serializability* property in its general form is difficult to achieve efficiently, but it is achieved efficiently via its special case *Distributed CO*: Each local component (e.g., a local DBMS) needs both to provide some form of CO, and enforce a special *vote ordering strategy* for the *Two-phase commit protocol* (2PC: utilized to commit distributed transactions). Differently from the general Distributed CO, *Distributed SS2PL* exists automatically when all local components are SS2PL based (in each component CO exists, implied, and the vote ordering strategy is now met automatically). This fact has been known and utilized since the 1980s (i.e., that SS2PL exists globally, without knowing about CO) for efficient Distributed SS2PL, which implies Distributed serializability and strictness (e.g., see Raz 1992, page 293; it is also implied in Bernstein et al. 1987, page 78). Less constrained Distributed serializability and strictness can be efficiently achieved by Distributed Strict CO (SCO), or by a mix of SS2PL based and SCO based local components.
2. About the references and Commitment ordering: (Bernstein et al. 1987) was published before the discovery of CO in 1990. The CO schedule property is called *Dynamic atomicity* in (Lynch et al. 1993, page 201). CO is described in (Weikum and Vossen 2001, pages 102, 700), but the description is partial and misses CO's essence. (Raz 1992) was the first refereed and accepted for publication article about CO algorithms (however, publications about an equivalent Dynamic atomicity property can be traced to 1988). Other CO articles followed. (Bernstein and Newcomer 2009)[1] note CO as one of the four major concurrency control methods, and CO's ability to provide interoperability among other methods.

Distributed recoverability

Unlike Serializability, *Distributed recoverability* and *Distributed strictness* can be achieved efficiently in a straightforward way, similarly to the way Distributed CO is achieved: In each database system they have to be applied locally, and employ a vote ordering strategy for the Two-phase commit protocol (2PC; Raz 1992, page 307).

As has been mentioned above, Distributed SS2PL, including Distributed strictness (recoverability) and Distributed commitment ordering (serializability), automatically employs the needed vote ordering strategy, and is achieved (globally) when employed locally in each (local) database system (as has been known and utilized for many years; as a matter of fact locality is defined by the boundary of a 2PC participant (Raz 1992)).

Other major subjects of attention

The design of concurrency control mechanisms is often influenced by the following subjects:

Recovery

All systems are prone to failures, and handling *recovery* from failure is a must. The properties of the generated schedules, which are dictated by the concurrency control mechanism, may affect the effectiveness and efficiency of recovery. For example, the Strictness property (mentioned in the section Recoverability above) is often desirable for an efficient recovery.

Replication

For high availability database objects are often *replicated*. Updates of replicas of a same database object need to be kept synchronized. This may affect the way concurrency control is done (e.g., Gray et al. 1996[2]).

See also

- Schedule
- Isolation (computer science)
- Distributed concurrency control

References

- Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman (1987): *Concurrency Control and Recovery in Database Systems* (http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx) (free PDF download), Addison Wesley Publishing Company, 1987, ISBN 0-201-10715-5
- Gerhard Weikum, Gottfried Vossen (2001): *Transactional Information Systems* (https://www.elsevier.com/books/transactional-information-systems/weikum/978-1-55860-508-4), Elsevier, ISBN 1-55860-508-8
- Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete (1993): *Atomic Transactions in Concurrent and Distributed Systems* (https://web.archive.org/web/20120715080630/http://www.elsevier.com/wps/find/bookdescription.cws_home/680521/description#description), Morgan Kaufmann (Elsevier), August 1993, ISBN 978-1-55860-104-8, ISBN 1-55860-104-X
- Yoav Raz (1992): "The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment." (https://web.archive.org/web/20070523182950/http://www.informatik.uni-trier.de/~ley/db/conf/vldb/Raz92.html) (PDF (http://www.vldb.org/conf/1992/P292.PDF)), *Proceedings of the Eighteenth International Conference on Very Large Data Bases* (VLDB), pp. 292-312, Vancouver, Canada, August 1992. (also DEC-TR 841, Digital Equipment Corporation, November 1990)

Citations

1. Philip A. Bernstein, Eric Newcomer (2009): *Principles of Transaction Processing*, 2nd Edition (http://www.elsevierdirect.com/product.jsp?isbn=9781558606234) Archived (https://web.archive.org/web/20100807151625/http://www.elsevierdirect.com/product.jsp?isbn=9781558606234) 2010-08-07 at the Wayback Machine, Morgan Kaufmann (Elsevier), June 2009, ISBN 978-1-55860-623-4 (page 145)
2. Gray, J.; Helland, P.; O'Neil, P.; Shasha, D. (1996). *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. The dangers of replication and a solution (ftp://ftp.research.microsoft.com/pub/tr/tr-96-17.pdf) (PDF). pp. 173–182. doi:10.1145/233269.233330 (https://doi.org/10.1145%2F233269.233330).

Concurrency control in operating systems

Multitasking operating systems, especially real-time operating systems, need to maintain the illusion that all tasks running on top of them are all running at the same time, even though only one or a few tasks really are running at any given moment due to the limitations of the hardware the operating system is running on. Such multitasking is fairly simple when all tasks are independent from each other. However, when several tasks try to use the same resource, or when tasks try to share information, it can lead to confusion and inconsistency. The task of concurrent computing is to solve that problem. Some solutions involve "locks" similar to the locks used in databases, but they risk causing problems of their own such as deadlock. Other solutions are Non-blocking algorithms and Read-copy-update.

See also

- Linearizability
- Lock (computer science)
- Mutual exclusion
- Semaphore (programming)
- Software transactional memory
- Transactional Synchronization Extensions

References

- Andrew S. Tanenbaum, Albert S Woodhull (2006): *Operating Systems Design and Implementation, 3rd Edition*, Prentice Hall, ISBN 0-13-142938-8
- Silberschatz, Avi; Galvin, Peter; Gagne, Greg (2008). *Operating Systems Concepts, 8th edition*. John Wiley & Sons. ISBN 978-0-470-12872-5.