Monograph on
Validation-Based Protocols
By Ms. Swati jain

Validation Based Protocol is a concurrency control method applied to transactional systems such as relational database management systems and software transactional memory. IT assumes that multiple transactions can frequently complete without interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted. Validation Based Protocol was first proposed by H.T. Kung and John T. Robinson.

Validation Based Protocol is generally used in environments with low data contention. When conflicts are rare, transactions can complete without the expense of managing locks and without having transactions wait for other transactions' locks to clear, leading to higher throughput than other concurrency control methods. However, if contention for data resources is frequent, the cost of repeatedly restarting transactions hurts performance significantly; that other concurrency control methods have better performance under these conditions. However, locking-based ("pessimistic") methods also can deliver poor performance because locking can drastically limit effective concurrency even when deadlocks are avoided.

Phases of Validation Based Protocol

The validation based protocols require that each transaction *Ti* executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order,

1. Read phase. During this phase, the system executes transaction *Ti*. It reads the values of the various data items and stores them in variables local to *Ti*. It performs all write operations on temporary local variables, without updates of the actual database.

2. Validation phase. Transaction *Ti* performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serializability.

3. Write phase. If transaction *Ti* succeeds in validation (step 2), then the system applies the actual updates to the database. Otherwise, the system rolls back *Ti*.

To perform the validation test, we need to know when the various phases of transactions *Ti* took place. We shall, therefore, associate three different timestamps with transaction *Ti*:

1. Start(*Ti*), the time when *Ti* started its execution.
2. Validation(*Ti*), the time when *Ti* finished its read phase and started its validation phase.
3. Finish(*Ti*), the time when *Ti* finished its write phase.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation(*Ti*). Thus, the value TS(*Ti*) = Validation(*Ti*) and, if TS(*Tj*) < TS(*Tk*), then any produced schedule must be equivalent to a serial schedule in which transaction *Tj* appears before transaction *Tk*. The reason we have chosen Validation(*Ti*),

rather than Start(*Ti*), as the timestamp of transaction *Ti* is that we can expect faster response time provided that conflict rates among transactions are indeed low.

The **validation test** for transaction *Tj* requires that, for all transactions *Ti* with TS(*Ti*) < TS(*Tj*), one of the following two conditions must hold:

1. Finish(*Ti*) < Start(*Tj*). Since *Ti* completes its execution before *Tj* started, the serializability order is indeed maintained.

2. The set of data items written by *Ti* does not intersect with the set of data items read by *Tj*, and *Ti* completes its write phase before *Tj* starts its validation phase (Start(*Tj*) < Finish(*Ti*) < Validation(*Tj*)). This condition ensures that the writes of *Ti* and *Tj* do not overlap. Since the writes of *Ti* do not affect the read of *Tj* , and since *Tj* cannot affect the read of *Ti*, the serializability order is indeed maintained.

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
⟨*validate* ⟩	
display($A + B$)	
	⟨*validate* ⟩
	write(B)
	write(A)

Schedule-6, a schedule produced by validation

Suppose that TS(*T*14) < TS(*T*15). Then, the validation phase succeeds in the schedule 6. Note
that the writes to the actual variables are performed only after the validation phase of *T*15. Thus, *T*14 reads the old values of *B* and *A*, and this schedule is serializable.

The validation scheme automatically guards against cascading rollbacks. However, there is a possibility of starvation of long transactions.

This validation scheme is called the **optimistic concurrency control (OCC)** scheme since transactions execute optimistically.

Advantages of Optimistic Concurrency Control:

i. This technique is very efficient when conflicts are rare. The occasional conflicts result in the transaction roll back.
ii. The rollback involves only the local copy of data, the database is not involved and thus there will not be any cascading rollbacks.

Problems of Optimistic Concurrency Control:

i. Conflicts are expensive to deal with, since the conflicting transaction must be rolled back.
ii. Longer transactions are more likely to have conflicts and may be repeatedly rolled back because of conflicts with short transactions.

Distributed Optimistic Concurrency Control Algorithm

Distributed optimistic concurrency control algorithm extends optimistic concurrency control algorithm. For this extension, two rules are applied −

Rule 1 − According to this rule, a transaction must be validated locally at all sites when it executes. If a transaction is found to be invalid at any site, it is aborted. Local validation guarantees that the transaction maintains serializability at the sites where it has been executed. After a transaction passes local validation test, it is globally validated.

Rule 2 − According to this rule, after a transaction passes local validation test, it should be globally validated. Global validation ensures that if two conflicting transactions run together at more than one site, they should commit in the same relative order at all the sites they run together. This may require a transaction to wait for the other conflicting transaction, after validation before commit. This requirement makes the algorithm less optimistic since a transaction may not be able to commit as soon as it is validated at a site.

Web Usage

The stateless nature of HTTP makes locking infeasible for web user interfaces. It's common for a user to start editing a record, then leave without following a "cancel" or "logout" link. If locking is used, other users who attempt to edit the same record must wait until the first user's lock times out.

HTTP does provide a form of built-in OCC: The GET method returns an ETag for a resource and subsequent PUTs use the ETag value in the If-Match headers; while the first PUT will succeed, the second will not, as the value in If-Match is based on the first version of the resource.

Some database management systems offer OCC natively - without requiring special application code. For others, the application can implement an OCC layer outside of the database, and avoid waiting or silently overwriting records. In such cases, the form includes a hidden field with the record's original content, a timestamp, a sequence number, or an opaque token. On submit, this is compared against the database. If it differs, the conflict resolution algorithm is invoked.

Examples include:

- MediaWiki's edit pages use OCC.
- Bugzilla uses OCC; edit conflicts are called "mid-air collisions".
- The Ruby on Rails framework has an API for OCC.
- The Grails framework uses OCC in its default conventions.
- The GT.M database engine uses OCC for managing transactions (even single updates are treated as mini-transactions).
- Microsoft's Entity Framework (including Code-First) has built-in support for OCC based on a binary timestamp value.

- Mimer SQL is a DBMS that only implements optimistic concurrency control.
- Pyrrho is a DBMS that uses optimistic concurrency control.
- Google App Engine data store uses OCC.
- The Apache Solr search engine supports OCC via the _version_ field.
- The Elasticsearch search engine supports OCC via the version attribute.
- CouchDB implements OCC through document revisions.
- The MonetDB column-oriented database management system's transaction management scheme is based on OCC.
- Most implementations of software transactional memory use OCC.
- Redis provides OCC through WATCH command.
- MySQL implements OCC in Group Replication configuration.
- Firebird uses Multi-generational architecture as implementation of OCC for data management.
- DynamoDB uses Conditional Update as implementation of OCC.
- Kubernetes uses OCC when updating resource.