# Different Locks

Locking can be done on below

* **Database Level**
* **Table Level**
* **Page-Level**
* **Row Level**
* Shared Lock->It used while read operation is done to the database row,any number of shared locks can be applied.
* Exclusive Lock->It used when write operation is done on the database row.Once exclusive lock is present no other lock can be applied on the database row
* **Intent exclusive (IX) Lock –**  
  This method of locking explicit locking at a lower level with exclusive or shared locks. This means that if a transaction has used this type of lock, then it must be a case of modifying the lower level of resources by imposing exclusive lock separately.
* **Intent shared (IS) Lock –**   
  This method of locking is explicit locking at a lower level of the tree but only with shared locks. This means that if a transaction has used this type of lock, then it must be a case of reading the lower level of resources by imposing shared lock separately.
* **Shared intent exclusive (SIX) Lock –**   
  This method of locking states that the transaction is used to read the resources at a lower level. Here in SIX, we impose the shared lock on all the resources that are available at the lower level. The subtree rooted by that node is locked explicitly in shared mode and explicit locking is done ata lower level with exclusive mode locks. In this method, only one SIX can be acquired on a relation at a time and if there are any other transactions for updating any change, then it will block those transactions.
* **Update (U) Lock –**    
  This method of locking can be imposed on a record that already consists of a shared lock and if it has a shared lock already, then the update lock will impose another shared lock on the target row or page of relation. This is the same as an exclusive lock and also in some ways flexible. Here in this lock, after checking that the transaction holds the update lock for modifying the data, then the update lock will be modified into an exclusive lock.

# Isolation Levels

**Level**

**Dirty Read**

**Nonrepeatable Read**

**Phantom Read**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dirty Read | Non Repeatable Read | Phontom Read |
| Read uncommitted:No locks are applied | y | y | y |
| Read committed:Shared lock is applied on read operation.Exclusive lock is applied for write operation | n | y | y |
| Repeatable Read :Shared and Exclusive lock is applied through out transaction | n | n | y |
| Serializable: Shared and Exclusive lock is applied through out transaction additionally range lock is applied on the rows | n | n | n |

A **phantom** read occurs when a transaction retrieves a set of rows twice and new rows are inserted into or removed from that set by another transaction that is committed in between.

**Nonrepeatable** Reads A nonrepeatable read occurs when a transaction reads the same row twice but gets different data each time. For example, suppose transaction 1 reads a row. Transaction 2 updates or deletes that row and commits the update or delete.

# concurrency level locking

**Optimistic Lock** :Read committed and Read Uncommitted this level concurrency is high

**Pessimistic Lock**: Read Repeatable and Serializable

Every Database has it own default locking strategy mostly they have Read Repeatable Locking

# Transaction Management in Distributed Systems

2PC : It is serializable process,A transaction co Ordinator is present to process the transaction.It is suitable when there is small transaction.

Two phase

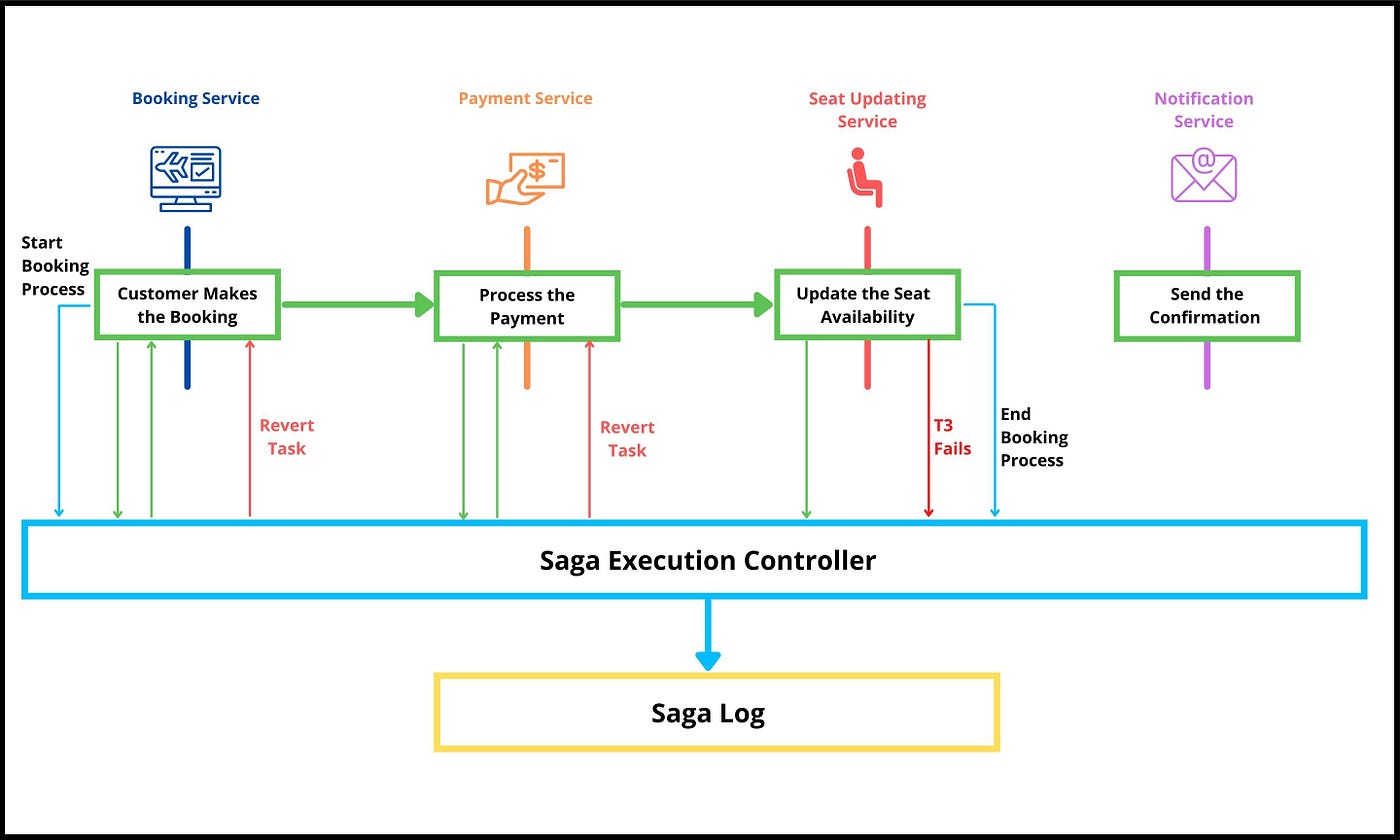
Prepare Phase

Commit Phase

# Patterns for distributed transactions within a microservices architecture | Red Hat Developer

**3pc**:It is extension of 2pc.extra phase pre commit phase is present.it is not much used as configurations is little bit confused

**Saga pattern**:It is asynchronous process.Queues are used for this implantation.It is mostly used in large transaction is involved



# Transaction Propagation Levels

|  |  |
| --- | --- |
| **REQUIRED** | **Always executes in a transaction.** If there is any existing transaction it uses it. If none exists then only a new one is created  Eg:registering new user |
| SUPPORTS | It may or may not run in a transaction. If current transaction exists then it is supported. If none exists then gets executed with out transaction.  **if calling service addNoteToSpecficUser having REQUIRED propagation, and registerUser and addNote having SUPPORT propagation then registerUser and addNote will make use of existing transaction created from addNoteToSpecficUser** |
| **NOT\_SUPPORTED** | **Always executes without a transaction.** If there is any existing transaction it gets suspended  if calling service addNoteToSpecficUser having REQUIRED propagation, and registerUser and addNote having NOT\_SUPPORTED propagation then registerUser and addNote will not make use of existing transaction created from addNoteToSpecficUser and nor it will creates it's own, rather it runs without any transaction. |
| **REQUIRES\_NEW** | **Always executes in a new transaction.** If there is any existing transaction it gets suspended  In this case, if calling service addNoteToSpecficUser having REQUIRED propagation, and registerUser and addNote having REQUIRES\_NEW propagation then registerUser and addNote will always creates it's own transaction and doesn't utilizes the existing/calling service transaction. |
| **NEVER** | **Always executes with out any transaction. It throws an exception if there is an existing transaction**  In this case, if calling service addNoteToSpecficUser having REQUIRED propagation, and registerUser and addNote having NEVER propagation then registerUser and addNote will not make use of existing transaction rather it will throw EXECEPTION. And if calling service addNoteToSpecficUser doesn't have any transaction, then registerUser and addNote will not create it's own transaction and it'll run without transaction. |
| **MANDATORY** | **Always executes in a transaction. If there is any existing transaction it is used. If there is no existing transaction it will throw an exception.**  In this case, if calling service addNoteToSpecficUser having REQUIRED propagation, and registerUser and addNote having MANDATORY propagation then registerUser and addNote will make use of existing transaction. And if calling service (addNoteToSpecficUser) doesn't have trsation then registerUser and addNote having MANDATORY propagation, will throw EXCEPTION.  So i short, calling service (addNoteToSpecficUser) should transaction else service which calls calling service will throw exception. |