Nested Transactions

CS3524 Distributed Systems

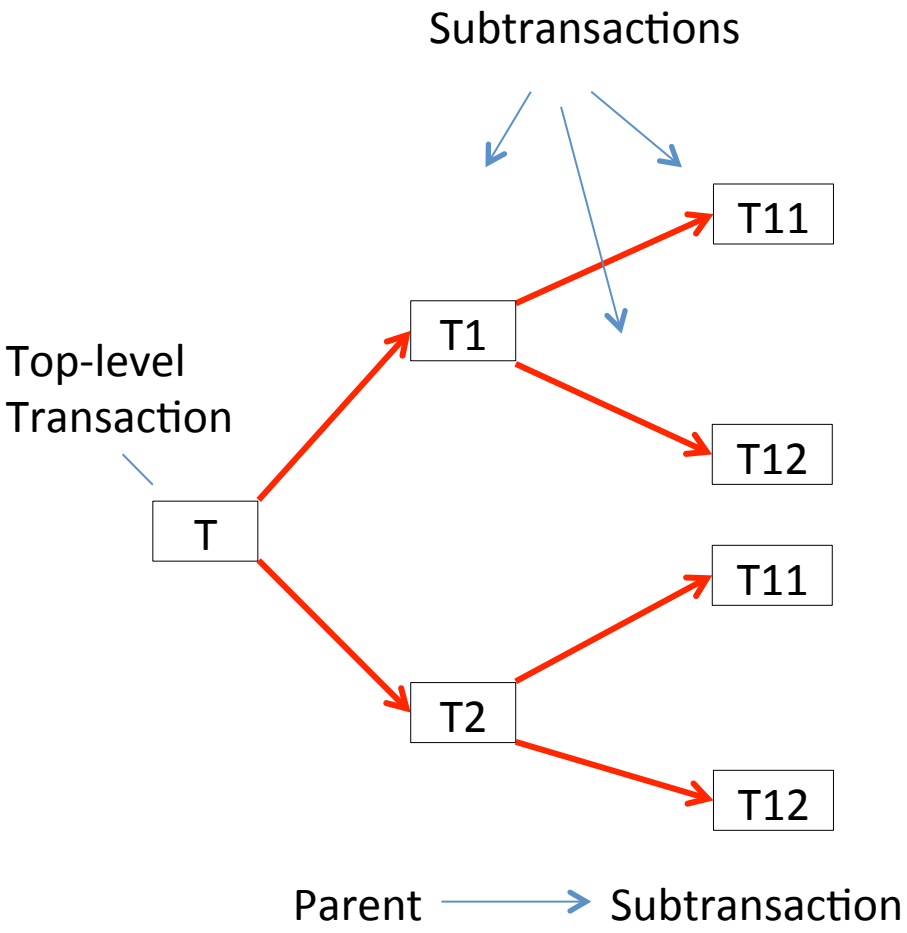
Lecture 10

Nested Transactions

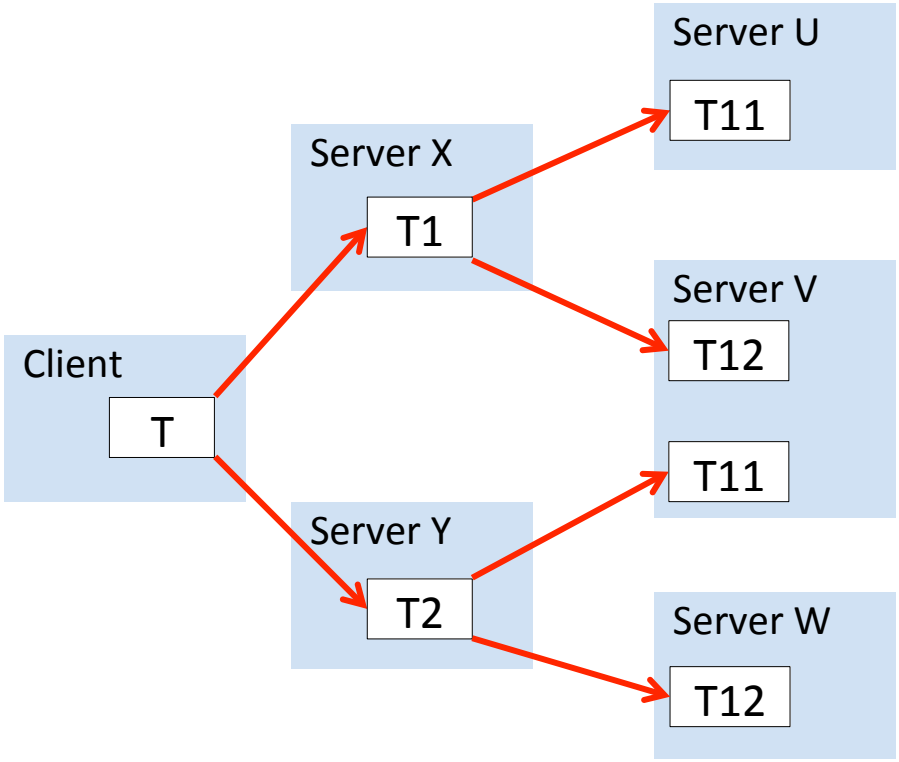
- So far, we only looked at “flat” transactions
  - A transaction has a defined beginning and an end, where commit/abort/unlock/validation actions take place
  - For long-lasting transactions, it may be beneficial to separate a transaction into sub-parts – transactions may consist of “sub-transactions”
- Transactions with sub-transactions are regarded as “Nested Transactions”

Nested Transactions

- Transactions may be composed of other transactions
  - A “parent” or “antecedent” transaction starts a set of “subtransactions”
- The outermost transaction of a set of nested transactions is called the **top-level transaction**
- All other transactions are called **sub-transactions**
- Nested transactions are related in terms of commit and abort behaviour



Nested Transactions

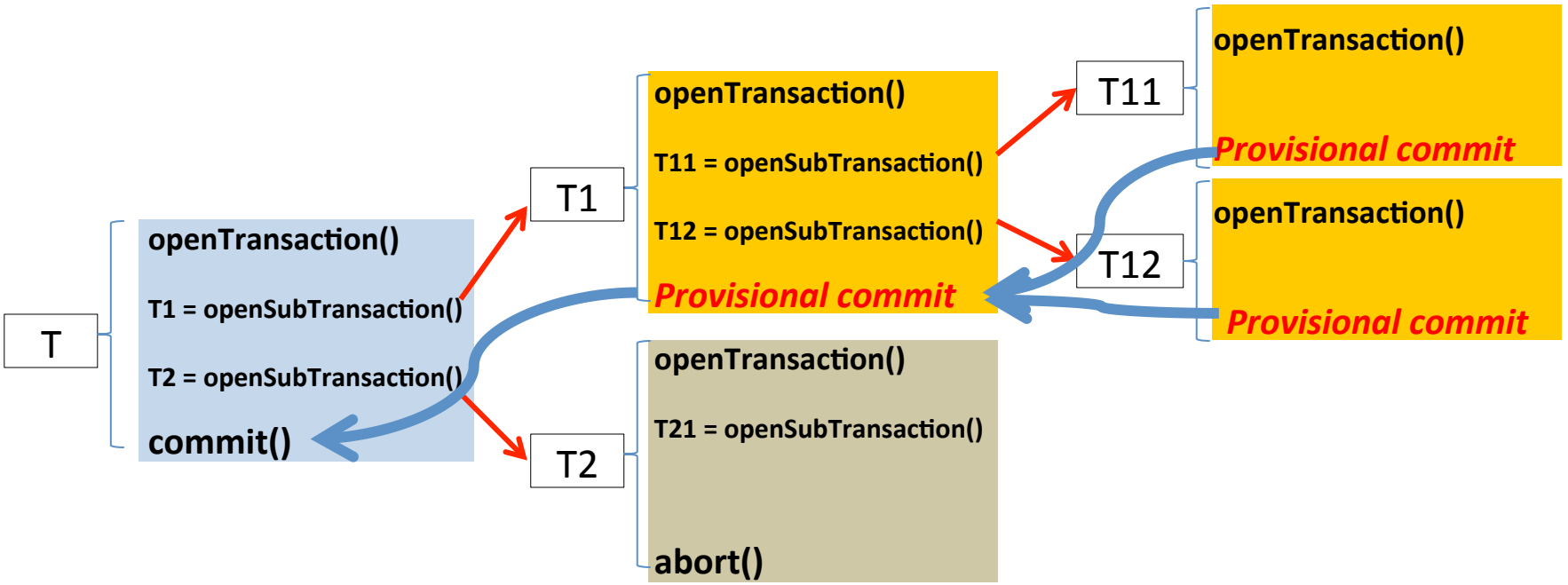


- Nested Transactions may be distributed across multiple physical servers

Subtransactions - Properties

- Atomicity
  - A subtransaction appears atomic to its parent in terms of transaction failures and concurrent access to data
- Concurrency among subtransactions
  - Subtransactions at the same level may run concurrently
  - If they access shared objects, concurrency control of their activities is needed
  - We assume that parent transactions are suspended when subtransactions are running – no concurrency between levels
- Commit / Abort Dependencies
  - Subtransactions can commit / abort independently of their parent transaction
  - If a subtransaction aborts, the parent transaction may decide whether to abort or not (e.g. restart the subtransaction)
  - The final commit of write operations is done by the top-level transaction
- Failure Robustness
  - A subtransaction can fail independently from its parent transaction – despite a subtransaction failing, the parent transaction can continue its execution
  - A parent transaction may decide to run a different transaction instead of the failed transaction

Provisional Commit of Subtransactions

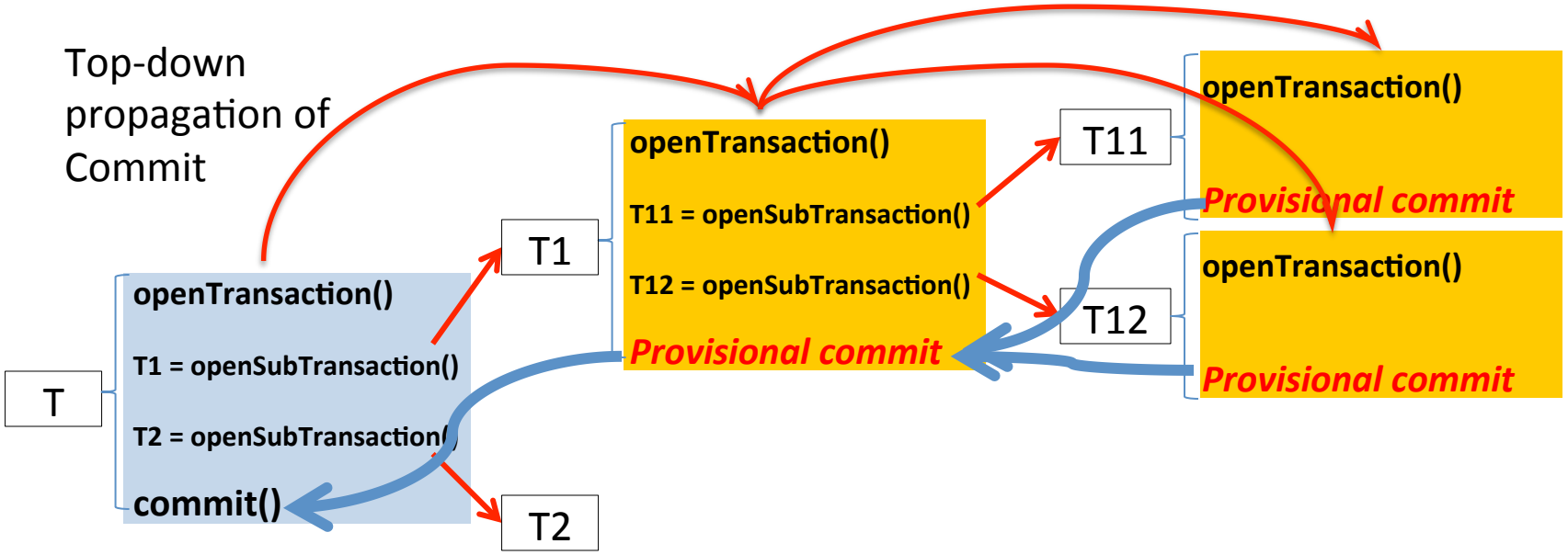


- Only the top-level transaction performs the final commit of the whole nested transaction
- A subtransaction can only report a “provisional” commit:
  - This expresses that the subtransaction is ready to commit its write actions, however, it is up to the parent transaction to finalise this commit
- Provisional commits are reported “bottom up” to the top-level transaction

Provisional Commit of Subtransactions

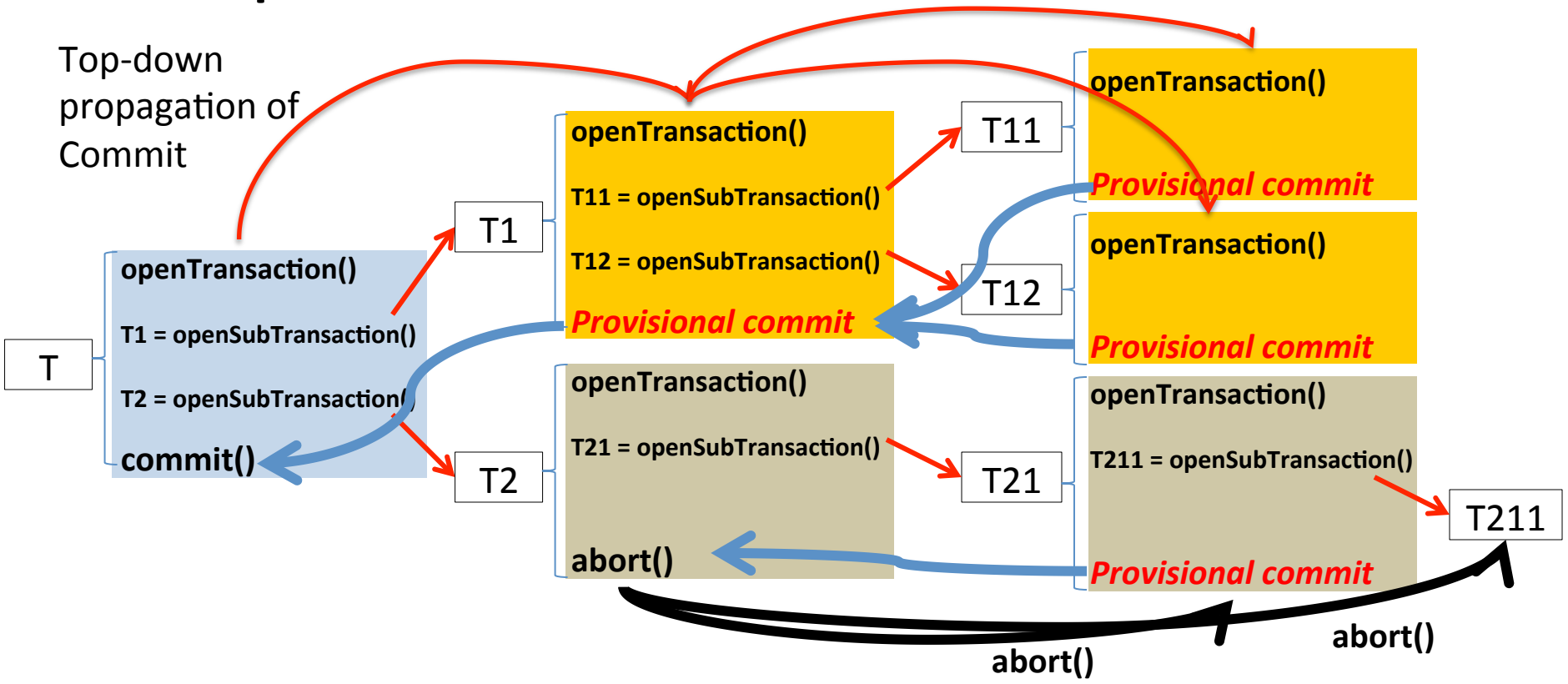
- Only the top-level transaction performs the final commit of the whole nested transaction
- A subtransaction can only report a “provisional” commit:
  - This expresses that the subtransaction is ready to commit its write actions, however, it is up to the parent transaction to finalise this commit
- Provisional commits are reported “bottom up” to the top-level transaction
- Final commits are propagated “top down” from the top-level transaction to all subtransactions
- Aborts happen immediately

Final Commit in Nested Transactions



- Final commits are propagated “top down” from the top-level transaction to all subtransactions

Independent Abort of Subtransactions



- Aborts happen immediately and independent of parent / top-level transaction
- A parent transaction can commit, even if some sub-transactions aborted

Committing Nested Transactions

- Subtransactions:
  - When a subtransaction completes, it makes an independent decision either to **provisionally** commit or to abort (a decision to abort is final)
    - The effect of sub-transactions are not permanent until the top-level transaction commits
  - When a subtransaction aborts, the parent can decide whether to abort as well or continue execution and commit
    - E.g.: this allows a parent to repeat a failed sub-transaction
- Parent transactions:
  - A parent transaction may commit or abort only after all its subtransactions have completed
  - If a parent transaction commits, all **provisionally** committed subtransactions may finally commit
    - If the top-level transaction commits, all nested provisionally committed transactions will commit finally
  - If a parent transaction aborts, all its subtransactions are aborted as well
    - If the top-level transaction aborts, the complete nested transaction aborts

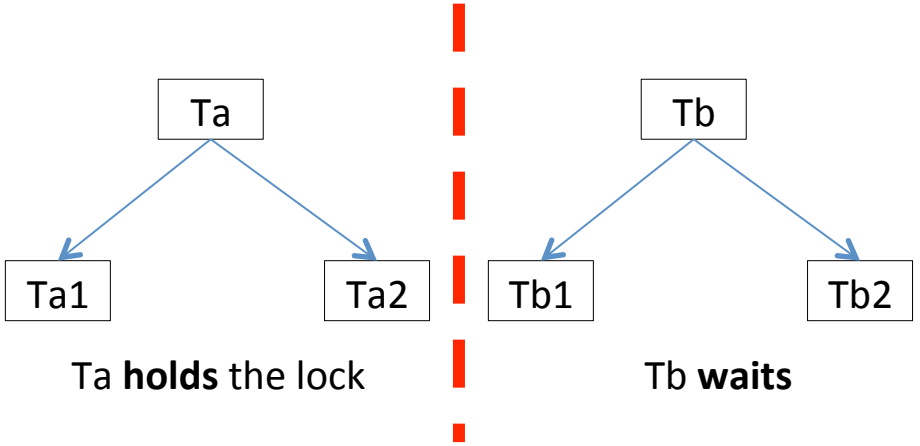
Benefits of Nested Transactions

- Improves concurrency
  - Subtransactions of the same hierarchy level (and their descendent sub-transactions) may run **concurrently**
  - Subtransactions may be distributed to different physical servers – true parallelisation of parts of a transaction
- Improves robustness
  - Independent commit / abort of Subtransactions
  - Parent transaction can make decisions based on the behaviour of subtransactions:
    - The abort of a sub-transaction results only in a partial loss / undo of actions of the overall nested transaction

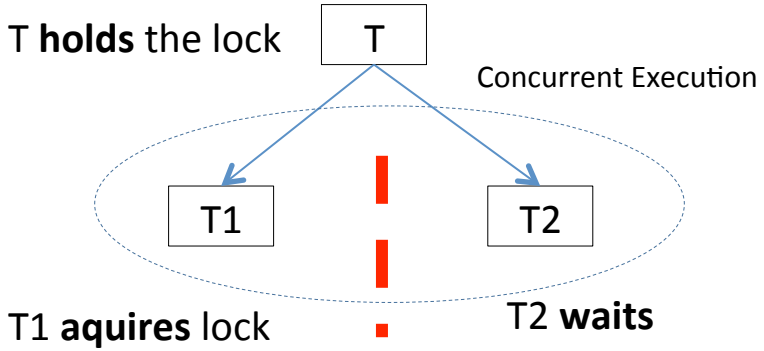
Locking in Nested Transactions

Using Locking in Nested Transactions

- Rule 1: Isolation *between* sets of concurrently running sub-transactions:
  - Each set of nested transactions is a single entity that must be prevented from observing the partial effects (write operations) of any other set of nested transactions



- Rule 2: Isolation *within* a set of concurrently running sub-transactions:
  - Each transaction within a set of nested transactions must be prevented from observing the partial effects (write operations) of the other transactions in the set



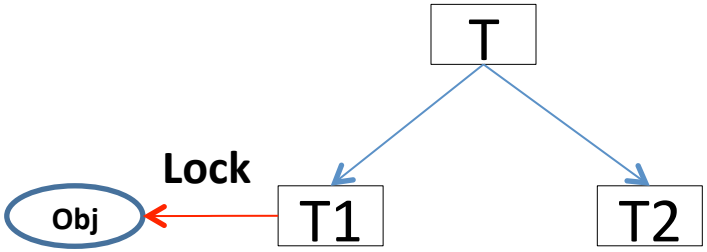
Simple Exclusive Locks

- Remember:
  - If a lock is granted, then the locking transaction has exclusive access to the locked object, until the transaction commits or aborts
  - No other transaction may lock the object during this time
- What does this mean for nested transactions?
  - As we know: subtransactions can only **provisionally** commit
  - As we know: **two-phase locking** must be observed
    - Locks can only be released at commit / abort of a transaction
    - Therefore, only the top-level transaction can release the lock in case of commit

Holding and Retaining exclusive Locks

- We distinguish two ways of possessing a lock
  - **Holding** a lock: the holding transaction has exclusive access to the locked object
    - There can at most be only one lock **holder** for exclusive locks
  - **Retaining** locks: an ancestor transaction **retains** all exclusive locks from provisionally committing subtransactions
    - A retained lock is a placeholder to hinder other transactions concurrent to the ancestor transaction to access the exclusively locked object, until provisional commit becomes a final commit

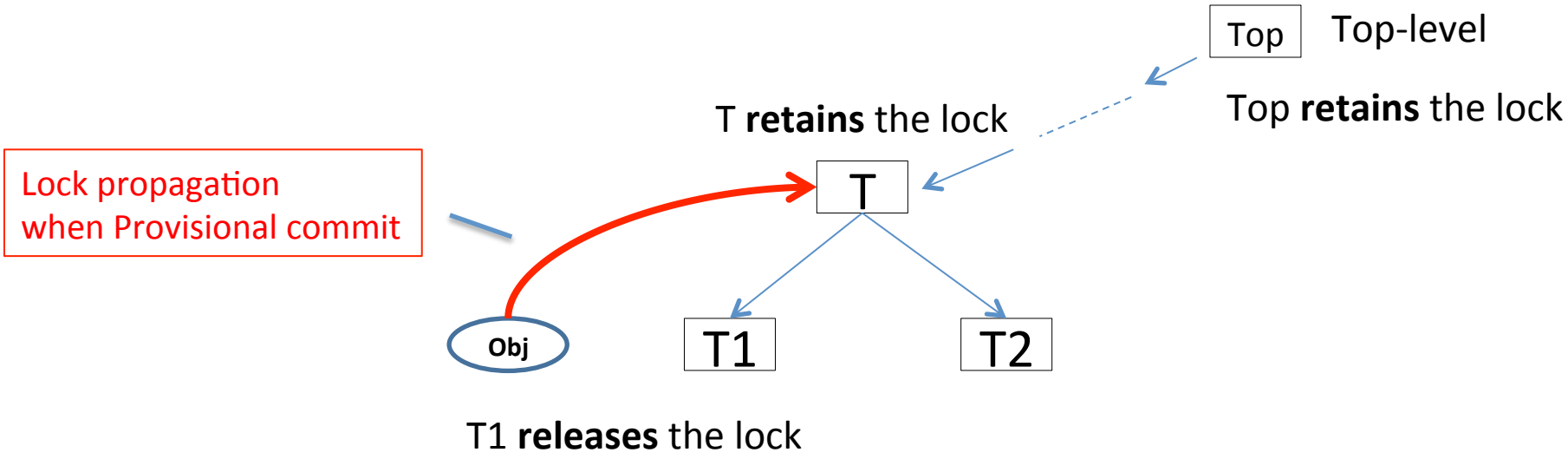
Simple Exclusive Lock for Nested Transactions



T1 **holds** the lock

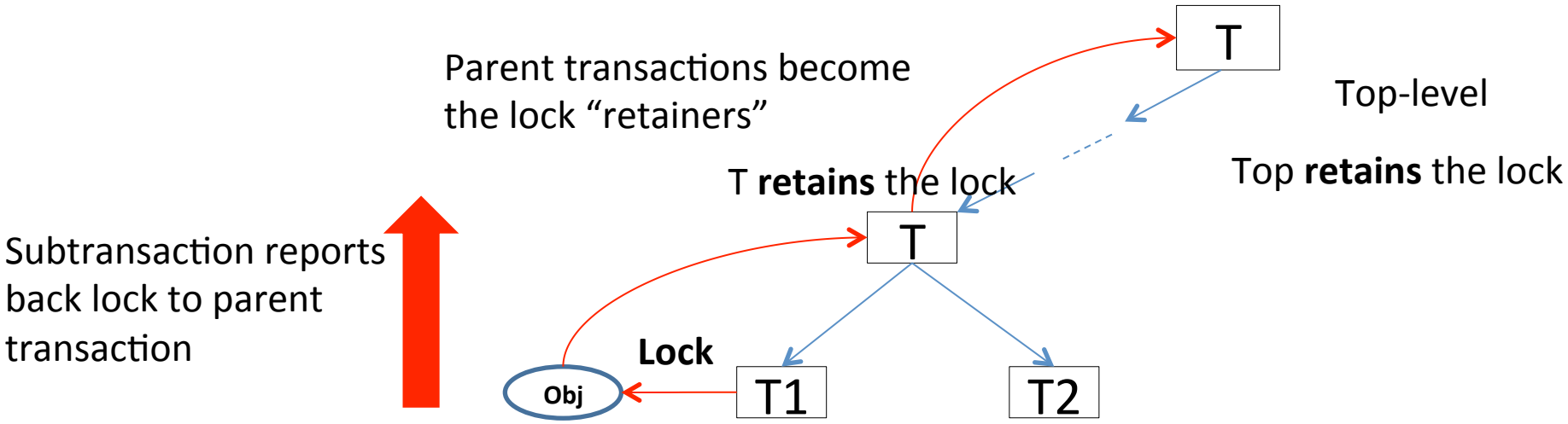
- Subtransaction T1 locks data object: T1 **holds** the lock
- When T1 provisionally commits, this lock cannot entirely be released
  - Parent transaction T may still abort the whole nested transaction
  - Parent retains the lock until the whole transaction is committed

Simple Exclusive Lock for Nested Transactions



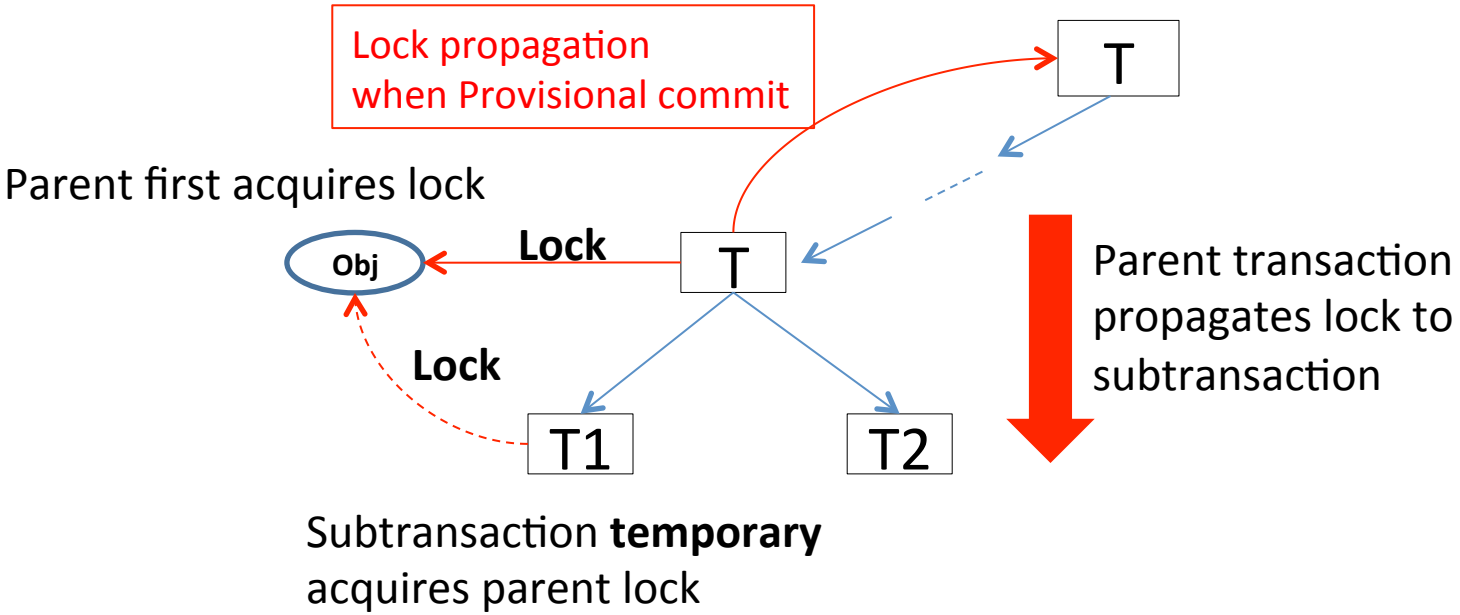
- When T1 provisionally commits, this lock cannot entirely be released
  - Parent transaction T may still abort the whole nested transaction
- Therefore: lock has to be propagated up to the parent transaction T
  - The Parent transaction T then **retains** the lock
- A retained lock is propagated up the hierarchy if the parent transaction also provisionally commits

Lock held by Sibling Transaction



- Every lock that is acquired by a sub-transaction and released at provisional commit, is inherited by its parent
- All ancestor transactions becomes "retainers" for this lock
- The top-level transaction will finally commit

Lock held by Parent Transaction

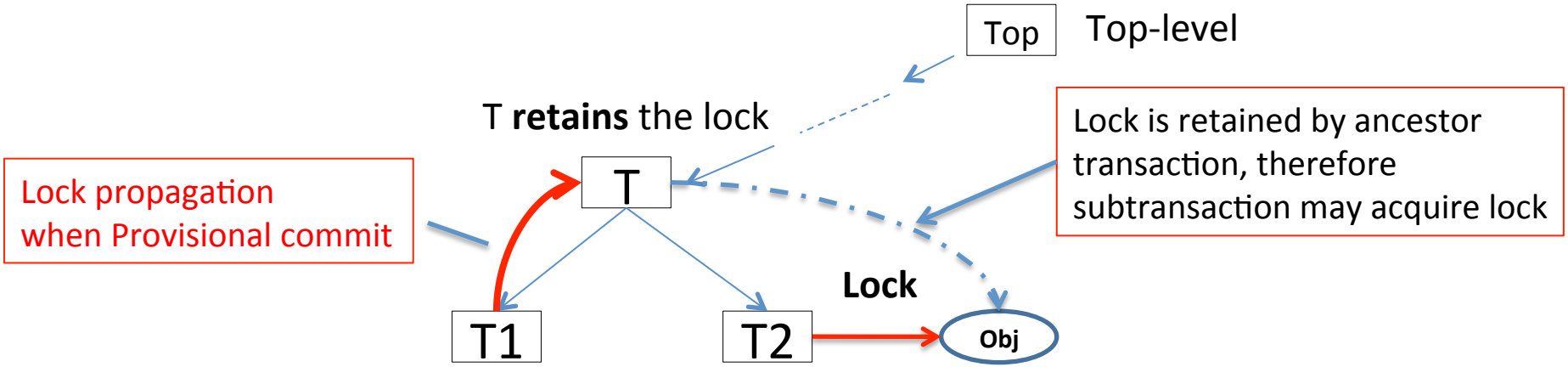


- Parents are not allowed to run concurrently with subtransactions – they are suspended, when subtransactions are running
- If a parent transaction acquires and hold a lock and starts a subtransaction (e.g. T1), then the subtransaction may acquire and hold this lock **temporarily**

Locking Rules for Exclusive Locks

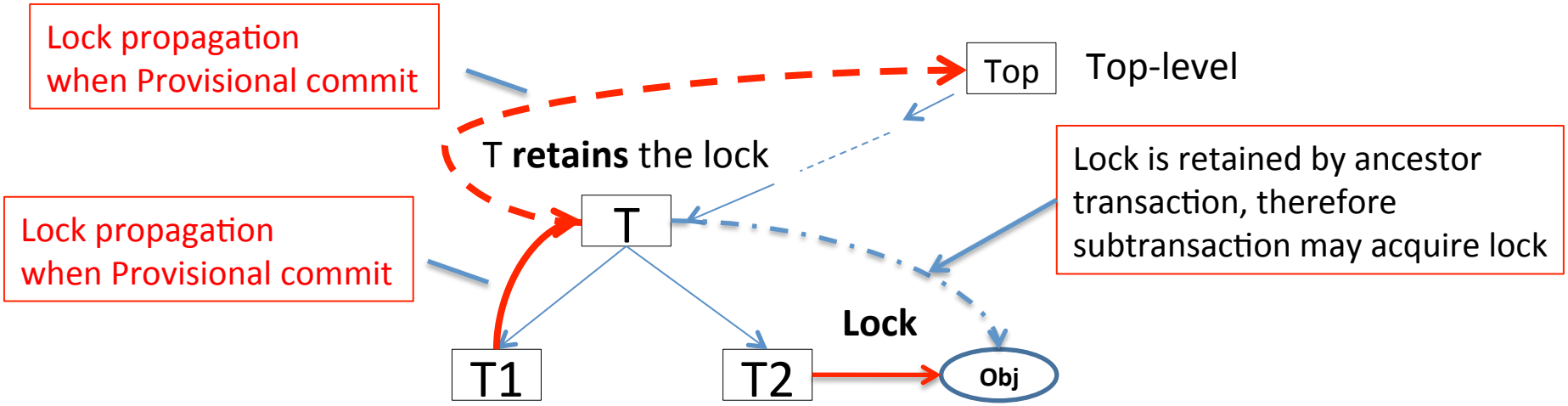
- Acquisition of locks
  - A transaction may acquire and **hold** a lock if no other transaction holds the lock, and
  - all **retainers** of the lock (if there are any) are ancestor transactions to the acquiring transaction
    - This propagates retained locks down to subtransactions
- Propagation of locks at transaction commit
  - When a transaction commits, its parent (if any) **retains** all locks this transaction held or retained
    - This propagates all locks up the transaction hierarchy
- Transaction abort
  - When a transaction aborts, all locks it **holds** are released.
  - If any ancestor transactions **retain** these locks they continue to do so
    - This allows to restart a subtransaction, which then may reacquire and **hold** these locks

Locking Rules for Exclusive Locks
Acquisition of Locks



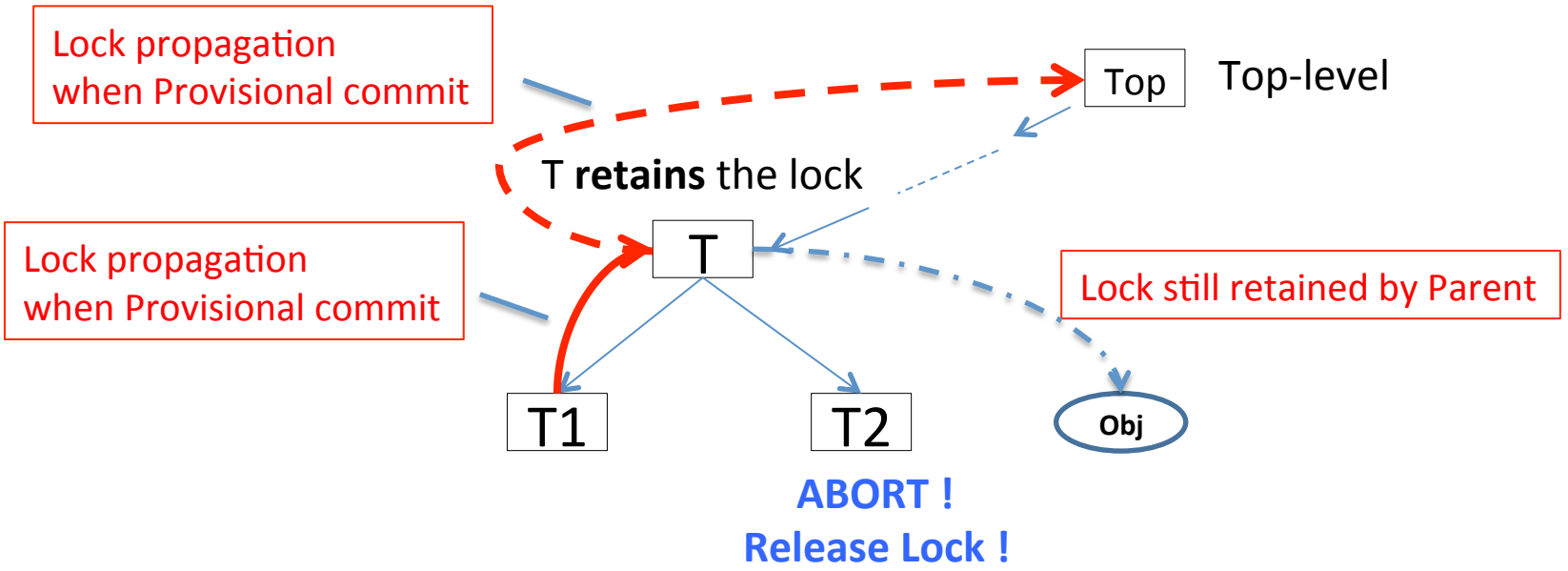
- A transaction may acquire and **hold** a lock if no other transaction holds the lock, and
- If an ancestor **retains** a lock, then any subtransaction in the hierarchy may acquire this lock
  - This propagates retained locks down to subtransactions

Locking Rules for Exclusive Locks
Propagation of Locks



- Propagation of locks at transaction commit
  - When a transaction commits, its parent (if any) **retains** all locks this transaction held or retained
    - This propagates all locks up the transaction hierarchy
- A retained lock is propagated up the hierarchy if the parent transaction also provisionally commits

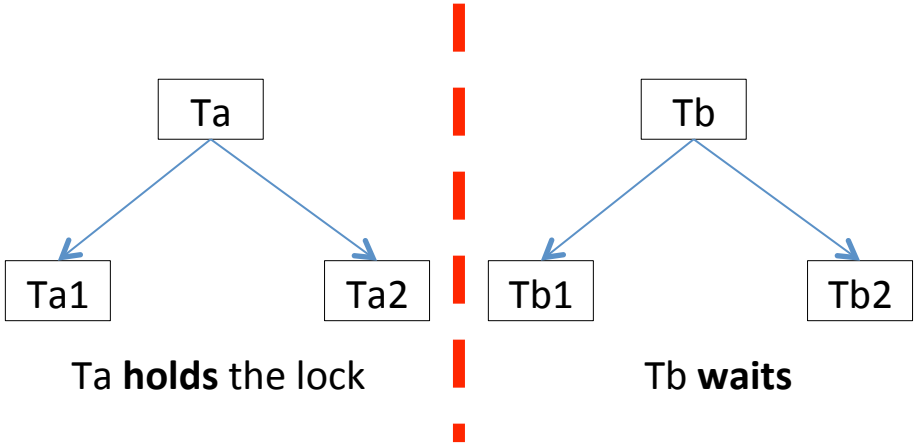
Locking Rules for Exclusive Locks Transaction Abort



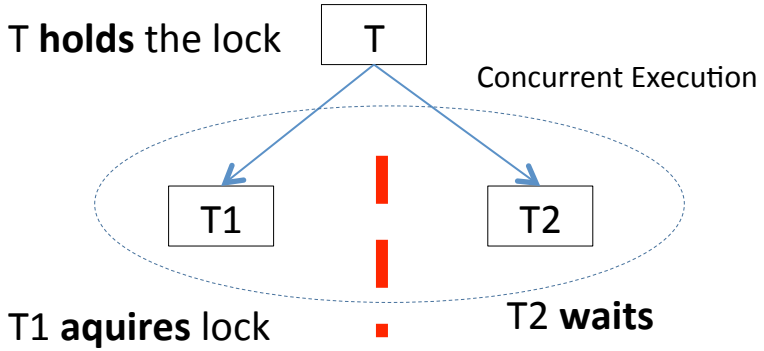
- When a transaction aborts, all locks it **holds** are released.
- If any ancestor transactions **retain** these locks they continue to do so
  - This allows to restart a subtransaction, which then may reacquire and **hold** these locks

Remember …

- Rule 1: Isolation *between* sets of concurrently running sub-transactions:
  - Each set of nested transactions is a single entity that must be prevented from observing the partial effects (write operations) of any other set of nested transactions



- Rule 2: Isolation *within* a set of concurrently running sub-transactions:
  - Each transaction within a set of nested transactions must be prevented from observing the partial effects (write operations) of the other transactions in the set



Isolation between Sets of Sub-Transactions Lock Propagation up

- Enforcing Rule 1 by propagation of locks up the transaction hierarchy:
  - Every lock that is acquired by a successful sub-transaction (provisional commit), is inherited by its parent when the sub-transaction completes
  - All ancestor transactions will inherit this lock and become “retainers” of this lock, until it is inherited by the top-level transaction
- As only the top-level transaction can finalize all commits and, with that, release locks, all locks have to propagate up the hierarchy

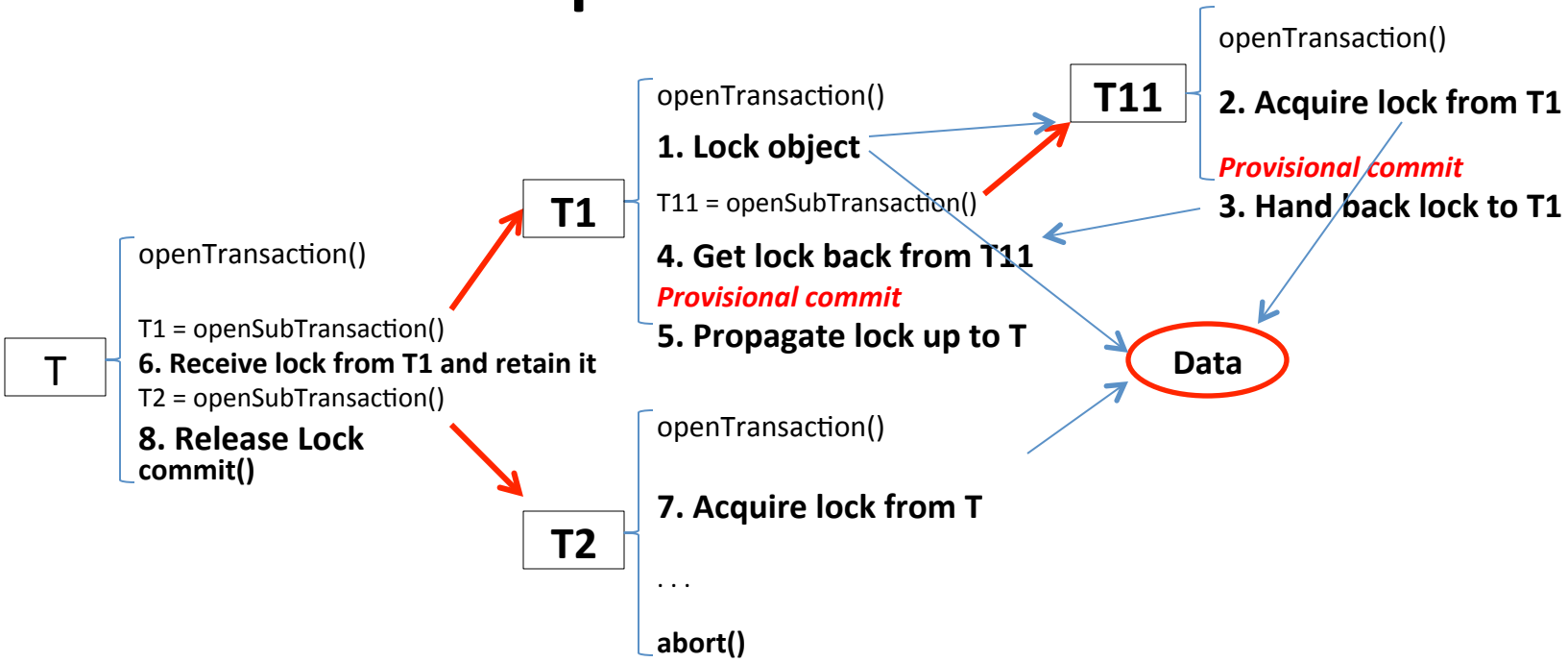
Benefit of Lock Retainment

- If a transaction ***retains*** a **write lock** on a particular object, it is not allowed to manipulate the object, but it still hinders other transactions to acquire a write lock on this object
- The lock is held or "retained" **until the top-level transaction commits/aborts** (all its sub-transactions committed/aborted)
- This prevents that members of different sub-trees of nested transactions can observe one anothers' partial effects

Isolation within a Set of Sub-transactions Lock Propagation down

- Enforcing Rule 2: Sub-transactions acquire Parent locks
  - Parent transactions are not allowed to run concurrently with their sub-transactions
    - they are suspended while sub-transactions are executing
- If a parent transaction has a lock on an object, it retains this lock and may hand it over to the next sub-transaction that is executed
  - The sub-transaction temporarily acquires the lock from its parent
- That means: if a parent acquires a lock on an object, this lock can be “re-used” in any of its sub-transactions
- Sub-transactions at the same level may run concurrently, however, we have to serialize their execution with a locking scheme if they want to access the same data object

Example Exclusive Locks

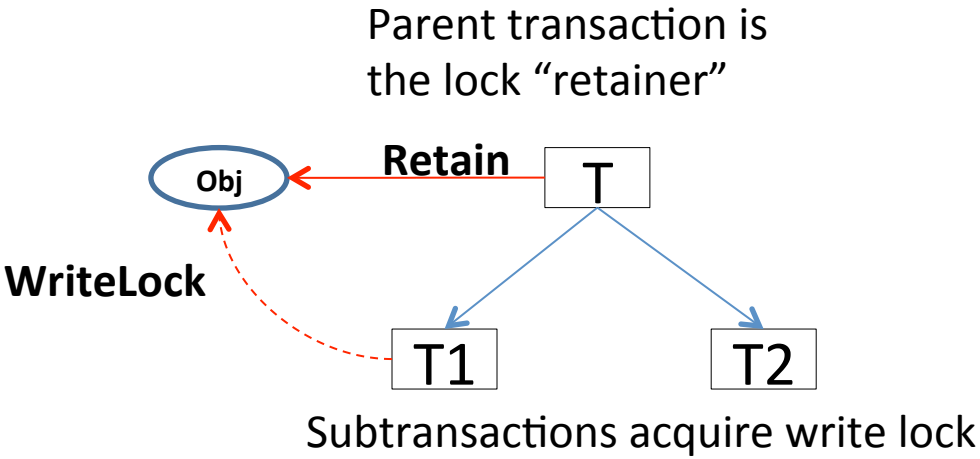


- Sub-transactions T1, T2 and T11 want to access a common object
- Subtransaction T1 first successfully acquires a lock, which it passes on to T11
- T11 holds this inherited lock for the duration of its execution and gives up this lock at its own completion; T1 regains the lock
- On completion, T1 will propagate this lock up the hierarchy to the top-level transaction T
- T retains this lock until its own completion
- As T retains this lock, sub-transaction T2 can acquire this lock and perform further manipulation actions on the data object
- Execution of subtransactions occurs serialized in this example, initiated by parent transactions

Lock Acquisition and Release Read and Write Locks

- Sub-transaction wants to acquire a read lock on a data object:
  - No other active transaction can have a **write lock** on that object
  - Only the ancestors of the sub-transaction may retain **read and write locks** on that data object
- Sub-transaction wants to acquire a write lock on a data object:
  - No other active transaction can have a **read or write lock** on that object
  - Only the ancestors of the sub-transaction may retain **read and write locks** on that data object

Shared Read / Exclusive Write Locks



- If the subtransactions at a particular level in the transaction hierarchy want to access a shared object, this access has to be serialized in the usual fashion:
  - Only one subtransaction may exclusively write: only one subtransaction may be the holder of a write lock at a point in time
  - Shared read is allowed, if there is no write lock

Lock Acquisition and Release Read and Write Locks

- When the sub-transaction commits
  - Its locks are inherited by its parent, allowing the parent to retain the locks in the same mode (read or write lock) as the sub-transaction, until its own commit or abort
- When the sub-transaction aborts
  - Its locks are discarded
  - When the parent already retains the locks it can continue to do so (restart of sub-transaction possible)

JDBC

Lecture 10

Java Database Connectivity

Java Database Connectivity (JDBC)

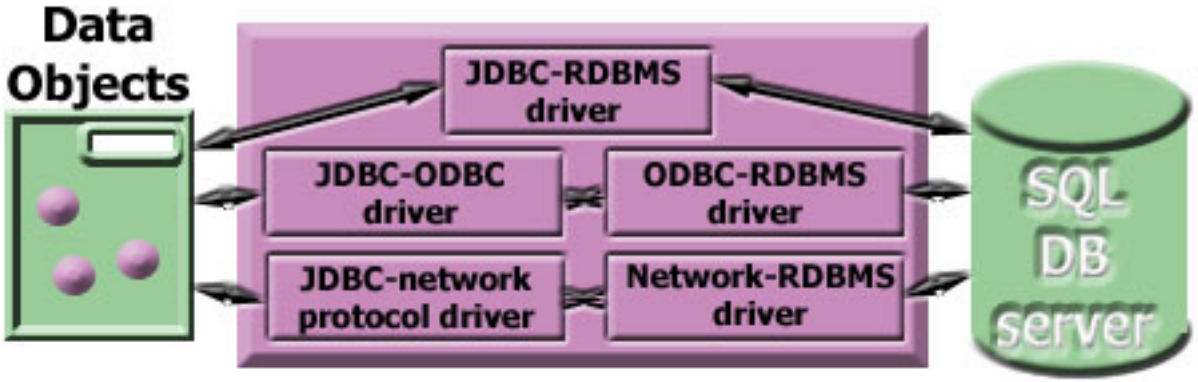
- JDBC is an API that provides a database-independent interface for communicating with relational databases
- JDBC uses SQL
  - Query on existing databases
  - Creating and updating database tables
- JDBC uses its own SQL syntax, a database driver for a particular database system translates this into DB-System-specific operations

JDBC API

- The JDBC API is a Java API that enables access to any kind of tabular data
  - Simple files
  - Relational Databases
- The JDBC API supports the following activities
  - Create tables
  - Manipulate data: INSERT, UPDATE, DELETE of data objects
  - Formulate queries
- JDBC concepts
  - Prepared statements
  - Support for stored procedures
  - Transactions
  - Handling exceptions and warnings

JDBC Database Drivers

- The JDBC connection between middle layers and DB is actually a substantial layer of software in its own right:



- Driver software accepts the JDBC *database transaction protocol* and converts it to the appropriate *native commands*.

Fundamentals of JDBC

- JDBC helps to write Java applications that manage these three programming activities:
  1. Connect to a data source, e.g. a database:
     - Database driver
     - Database name (URL)
     - Login/password for establishing a connection
  2. Send queries and update statements to the data source:
     - Create an SQL Statement for an established connection
     - Execute the SQL Statement
  3. Retrieve and process the results of a query received from the data source

SQL Statements

- Database drivers enable you to execute any valid SQL-92 statement; for example:

```
create table
    Students( SID int,
              FirstName char(10),
              LastName char(10) );

delete * from Students;

insert into
    Students( SID, FirstName, LastName )
    values( 1234, 'John', 'Smith' );
```

- These could be used to initialise our Students table

Using JDBC: Establish a Connection

- First, we need to establish a connection to the database
- To do this, we first need the database driver classes loaded into the VM:

```
try
{
    Class.forName( "org.gjt.mm.mysql.Driver" );
}
catch (ClassNotFoundException e)
{
    System.err.println( "Can't load driver." );
}
```

- This call to Class.forName() loads a driver class for a particular data source – in our example, this is a MySQL database

Connecting to the Database

- Databases are referred to by URLs of the form

  jdbc:driver:Database

- JDBC provides the class **DriverManager** to open a connection to a particular database

```
try
{
    String dbURL = "jdbc:mysql://student-mysql/cs3524";
    Connection dbCon =
          DriverManager.getConnection( dbURL,
                                       dbUname,
                                       dbPasswd );
}
catch (SQLException e)
{
    System.err.println( "Can't connect." );
}
```

Executing SQL Statements

- JDBC provides the class **`Statement`** for executing SQL statements
- It has to be instantiated from the database connection

```
try
{
    Statement stmt = dbCon.createStatement();
    stmt.execute( "create table " +
                  "Students( SID int, FirstName char(10)," +
                  "LastName char(10) )" );
}
catch (SQLException e)
{
    System.out.println( "...table exists; " +
                        "deleting entries..." );
    stmt.execute( "delete * from Students" );
}
```

A simple JDBC Query

- A simple query and result processing example:

```
try
{
     Statement stmt = dbCon.createStatement();
     ResultSet rs =
               stmt.executeQuery( "select * from Students" );
     ResultSetMetaData rsmd = rs.getMetaData() ;
     int cols = rsmd.getColumnCount() ;
     while( rs.next() )
     {
          for (int i = 1; i <= cols; i++) {
             System.out.print( rs.getString( i ) + "\t" );
          }
          System.out.println();
     }
} // Then catch and handle SQLExceptions.
```

ResultSet Processing
The Cursor concept

- The Statement object is used to send the SQL query to the DBMS
- The **`executeQuery()`** method returns a **`ResultSet`**
- The **`ResultSet`** implements a ***cursor:***
  - A cursor is a control structure that allows to traverse a result set of a query
  - It provides a **`next()`** method that returns the next dataset
  - In a Java application, we can iterate over a result set of the query
  - Compare this to the implementation of **`java.util.Iterator`**

ResultSet Processing
Database Schema

- As well as containing the results of the query, a ResultSet contains ResultSetMetaData
- This meta data provides information on the database schema
  - The database schema defines the types of the entries returned and other information including:
    - The number of columns in the relation returned
    - The names of each column; e.g. SID

Efficiency Issues

- Database access is costly
  - The SQL string must be parsed and validated every time the method **`executeQuery()`** is invoked
  - Each query execution involves database access overheads
- How can we minimise these?
  - Use a JDBC database driver that has been optimised for your RDBMS
  - Use ***prepared statements*** for common queries so that the SQL parsing is done only once
  - Try to batch queries if possible

Prepared Statements

- Consider the operation to insert an entry in the **`Students`** table; this must be done many times, so why not parse and verify the SQL only once?

```
PreparedStatement pstmt =
    _dbCon.prepareStatement (
                    "insert into Students " +
                    "(SID, FirstName, LastName)" +
                    " values( ?, ?, ? )" );
  pstmt.clearParameters();
  pstmt.setInt( 1, 1234 );
  pstmt.setString( 2, "John" );
  pstmt.setString( 3, "Smith" );
  ResultSet rs = pstmt.executeUpdate();
```

Note the use of question marks

Query Batching

- JDBC provides us with a means to batch sets of queries and execute them all at once
- Suppose we have either a **`Statement`** or a **`PreparedStatement`**
- We can use methods **`addBatch()`** and **`executeBatch()`**
- This minimises the overheads of contacting the database through the driver when we execute statements
- Let's now look at an example that uses both prepared statements and query batching

Prepared Statements Plus Batching

```
PreparedStatement pstmt =
    _dbCon.prepareStatement (
                    "insert into Students " +
                    "(SID, FirstName, LastName)" +
                    " values( ?, ?, ? )" );
  addStudent( pstmt, 1234, "Tony", "Blair" );
  addStudent( pstmt, 2341, "Michael", "Howard" );
  addStudent( pstmt, 3412, "Charles", "Kennedy" );
  addStudent( pstmt, 4123, "Gordon", "Brown" );
  addStudent( pstmt, 4321, "Oliver", "Letwing" );
  addStudent( pstmt, 3214, "Vincent", "Cable" );
  pstmt.executeBatch();
```

Add a Student to the Batch

```
void addStudent( PreparedStatement pstmt,
                 int sid,
                 String firstName,
                 String lastName )
    throws SQLException
  {
    pstmt.clearParameters();
    pstmt.setInt( 1, sid );
    pstmt.setString( 2, firstName );
    pstmt.setString( 3, lastName );
    pstmt.addBatch();
  }
```

Transactions in JDBC

- JDBC allows to manage transactions
- Default behaviour:
  - When a connection is created, it is in auto-commit mode – each individual SQL statement is treated as a transaction and is auto-committed right after execution
- Explicit transaction management is possible
  - We want to group more than one SQL statement together as a transaction
- To do
  - Set auto-commit to false

```
con.setAutoCommit(false);
```

  - Call commit() explicitly for a database connection

```
con.commit();
```

Transactions in JDBC

```
con.setAutoCommit(false);
PreparedStatement updateSales =
     con.prepareStatement(
        "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");
updateSales.setInt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal =
     con.prepareStatement(
       "UPDATE COFFEES SET TOTAL = TOTAL + ? " +
       "WHERE COF_NAME LIKE ?");
updateTotal.setInt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();

con.commit();

con.setAutoCommit(true);
```

Transactions in JDBC

- Setting a Savepoint and perform Rollback
  - The transaction is rolled back to the Savepoint – first INSERT is successful, second INSERT is undone

```
Statement stmt = conn.createStatement();
int rows =
      stmt.executeUpdate("INSERT INTO TAB1 (COL1) VALUES " +
                         "(?FIRST?)");
// set savepoint
Savepoint svpt1 = conn.setSavepoint("SAVEPOINT_1");

rows =
      stmt.executeUpdate("INSERT INTO TAB1 (COL1) " +
                         "VALUES (?SECOND?)");
...
conn.rollback(svpt1);
...
conn.commit();
```