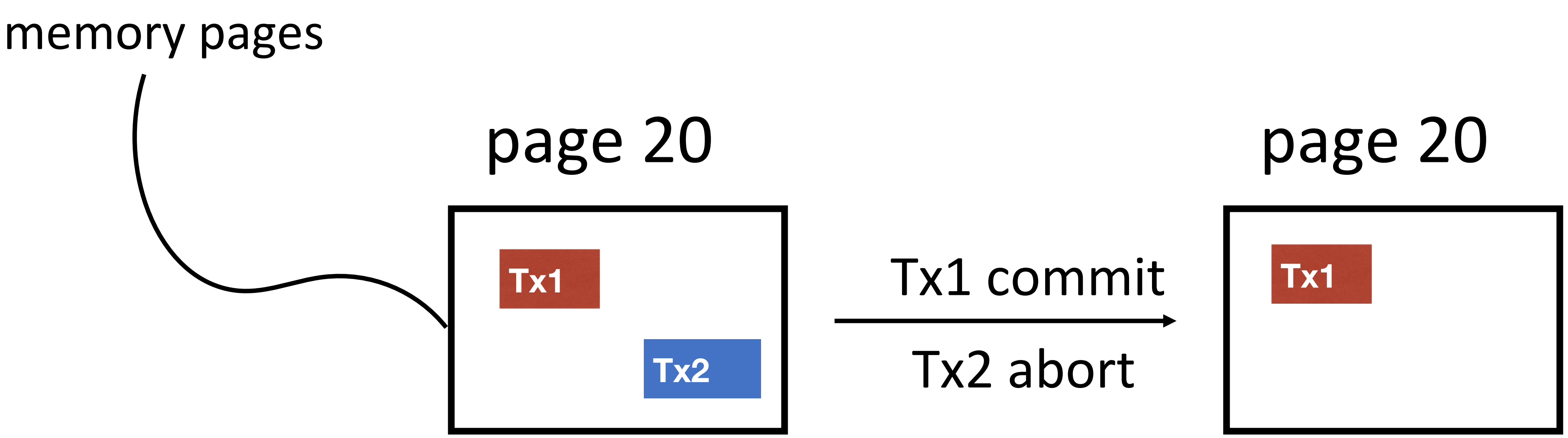
ARIES

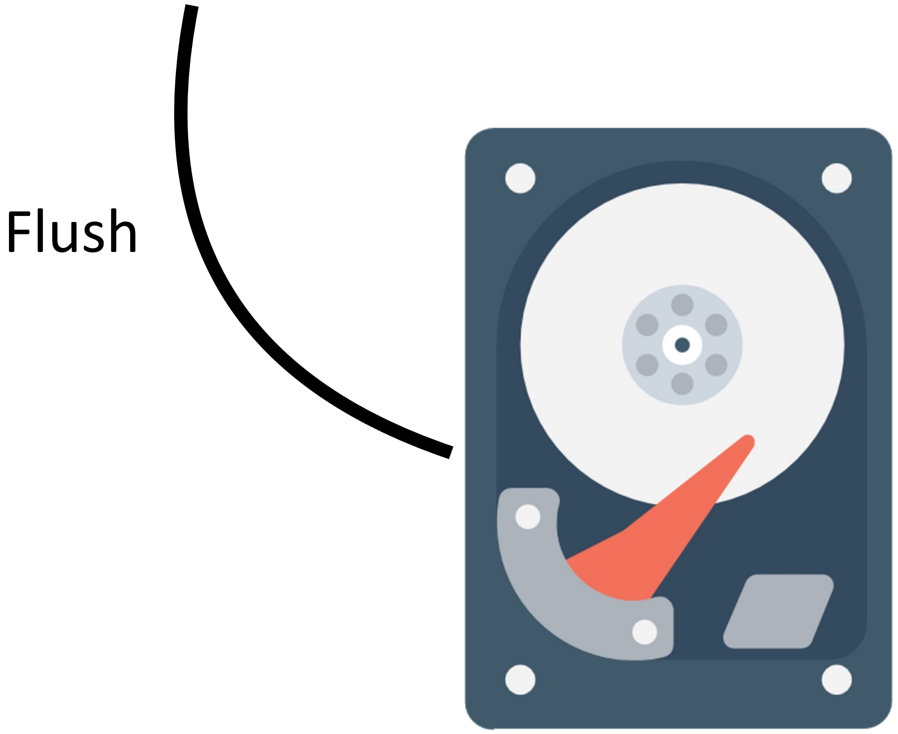
DataLab

Introduction to Database Systems

2022 Spring

What we expected





- In page 20 :
  - Tx1 is a winner tx
  - Tx2 is a loser tx

However…

- Steal
  - Due to buffer management, dirty pages may be flushed to disk before txs commit
  - The changes made by **loser** txs must UNDO
- No Force
  - Due to performance reason, dirty pages won't get flushed immediately after txs commit
  - The changes made by **winner** txs must REDO

Logs in ARIES

Physical Log Record

- Record format :
  - Set Value Record

    <**Op Code**, **txNum**, fileName, blockNum, offset, sqlType, **oldVal**, **newVal** >
  - Index Page Insert/Delete Record :

    <Op Code, txNum, fileName, blockNum, insertSlot, insertKey, insertRidBlkNum, insertRidId>
- REDO :
  - Apply newVal to the page
- UNDO :
  - Apply oldVal to the page
  - Append its ***Compensation Log Record***

Compensation Log Record

- A **CLR** describes the actions taken to undo the actions of a previous update record.
- CLRs are added to log like any other record.
- Only need to **Redo** CLRs.
- It has all the fields of an update log record plus the **undoNext** pointer (the next-to-be-undone LSN).

Why CLR is Redo Only ?

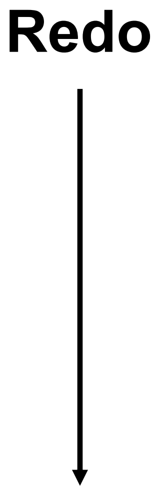
[0] <Start 1>

[1] <SetVal , 1 , Page 20 , 0 , 1>

[2] <SetVal , 1 , Page 20 , 1 , 2>

[3] <SetVal , 1 , Page 20 , 2 , 3>

Crash Here !



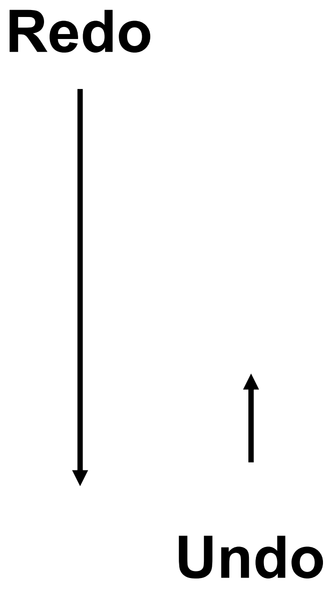
Why CLR is Redo Only ?

[0] <Start 1>

[1] <SetVal , 1 , Page 20 , 0 , 1>

[2] <SetVal , 1 , Page 20 , 1 , 2>

[3] <SetVal , 1 , Page 20 , 2 , 3>



Redo

Undo

Crash Here !

[4]<SetValClr , 1, Page 20 , 3 , 2 > // Append Undo [3] Redo log

Crash Again !

Why CLR is Redo Only ?

Redo

[0] <Start 1>

[1] <SetVal , 1 , Page 20 , 0 , 1>

[2] <SetVal , 1 , Page 20 , 1 , 2>

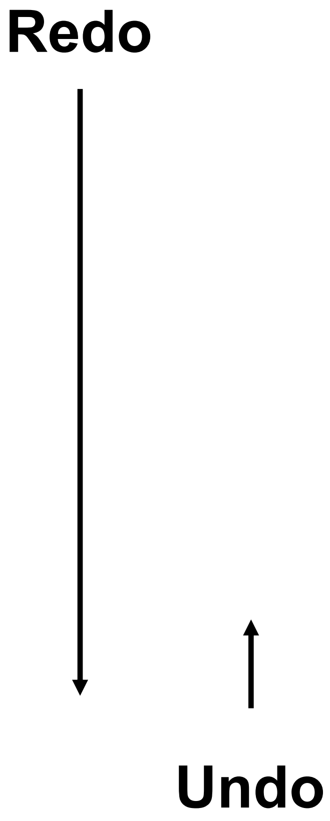
[3] <SetVal , 1 , Page 20 , 2 , 3>

Crash Here !

[4]<SetValClr , 1, Page 20 , 3 , 2 >

Crash Again !

Why CLR is Redo Only ?



Redo

Undo

[0] <Start 1>

[1] <SetVal , 1 , Page 20 , 0 , 1>

[2] <SetVal , 1 , Page 20 , 1 , 2>

[3] <SetVal , 1 , Page 20 , 2 , 3>

Crash Here !

[4]<SetValClr , 1, Page 20 , 3 , 2 >

Crash Again !

[5]<SetValClr , 1, Page 20 , 2 , 3 >   // Append Undo { Undo [3] Redo log } Redo log

Crash Again !

Why CLR is Redo Only ?

Redo

[0] <Start 1>

[1] <SetVal , 1 , Page 20 , 0 , 1>

[2] <SetVal , 1 , Page 20 , 1 , 2>

[3] <SetVal , 1 , Page 20 , 2 , 3>

Crash Here !

[4]<SetValClr , 1, Page 20 , 3 , 2 >

Crash Again !

[5]<SetValClr , 1, Page 20 , 2 , 3 >

Crash Again !

Why CLR is Redo Only ?

[0] <Start 1>

[1] <SetVal , 1 , Page 20 , 0 , 1>

[2] <SetVal , 1 , Page 20 , 1 , 2>

[3] <SetVal , 1 , Page 20 , 2 , 3>

Crash Here !

[4]<SetValClr , 1, Page 20 , 3 , 2 >

Crash Again !

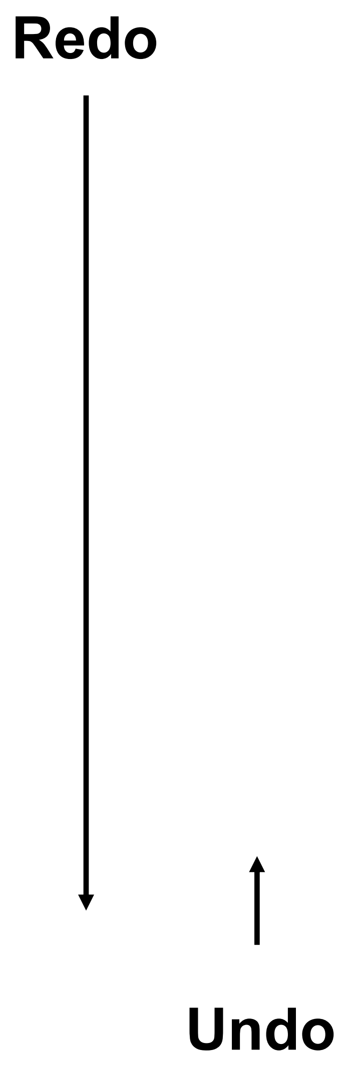
[5]<SetValClr , 1, Page 20 , 2 , 3 >

Crash Again !

[6]<SetValClr , 1, Page 20 , 3 , 2 >

// Append Undo { Undo { Undo [3] Redo log } Redo log} Redo log

...



Why CLR is Redo Only ?

[0] <Start 1>

[1] <SetVal , 1 , Page 20 , 0 , 1>

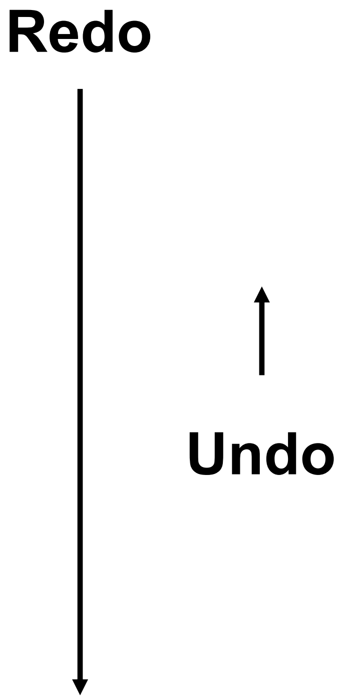
[2] <SetVal , 1 , Page 20 , 1 , 2>

[3] <SetVal , 1 , Page 20 , 2 , 3>

Crash Here !

[4]<SetValClr , 1, Page 20 , 3 , 2 >

Crash Again !



Redo

Undo

Why CLR is Redo Only ?

[0] <Start 1>

[1] <SetVal , 1 , Page 20 , 0 , 1>

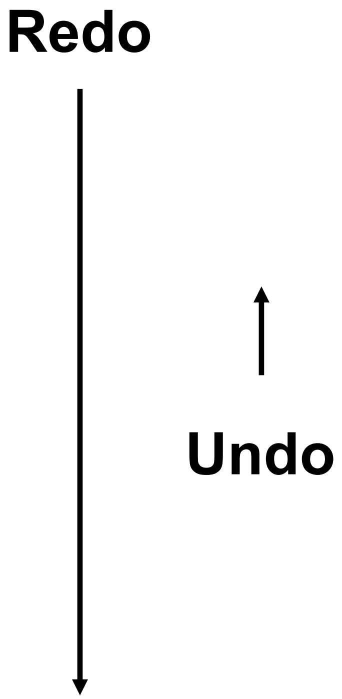
[2] <SetVal , 1 , Page 20 , 1 , 2>

[3] <SetVal , 1 , Page 20 , 2 , 3>

Crash Here !

[4]<SetValClr , 1, Page 20 , 3 , 2 >

Crash Again !



Redo

Undo

How do we know where Undo should start?

Why CLR Needs UndoNext?

- UndoNext helps skip logs which have been Undone by Redo

```
[0] <Start 1>
[1] <SetVal , 1 , Page 20 ,  0 , 1>
[2] <SetVal , 1 , Page 20 ,  1 , 2>
[3] <SetVal , 1 , Page 20 ,  2 , 3>
        Crash Here !
[4]<SetValClr 1, Page 20 , 3 , 2 , [3] >  // Append Undo [3] Redo log
        Crash Again !
[5]<SetValClr 1, Page 20 , 2 , 1 , [2] > // Append Undo [2] Redo log
[6]<SetValClr 1, Page 20 , 1 , 0 , [1] >
```

Logical Log Record

- Record format :

  - Logical - Start Record

    <OP Code, txNum>

  - Record File Insert/Delete End Record :

    <Op Code, txNum,fileName, blockNum, slotId , logicalStartLSN>

  - Index Insert/Delete End Record :

    <Op Code, txNum, tblName, fldName, searchKey, recordBlockNum, recordSlotId , logicalStartLSN>

- REDO :

  - Do nothing

- UNDO :

  - Undo **completed** logical log **logically**

  - Undo **partial** logical log **physically**

  - Append **Logical Abort** log record

Rollback a completed Logical Log Record

[0] <Start 1>

[1] <LogicalStart, 1 >

[2] <Index Page Insert , 1 , … >

[3] <SetVal , 1 , Page 2 , 1 , 2>

[4] <SetVal , 1 , Page 20 , 2 , 3>

[5] <Record File Insert End , 1, … , [1] >

Crash Here !

Rollback a completed Logical Log Record

[0] <Start 1>

[1] <LogicalStart, 1 >

[2] <Index Page Insert , 1 , … >

[3] <SetVal , 1 , Page 2 , 1 , 2>

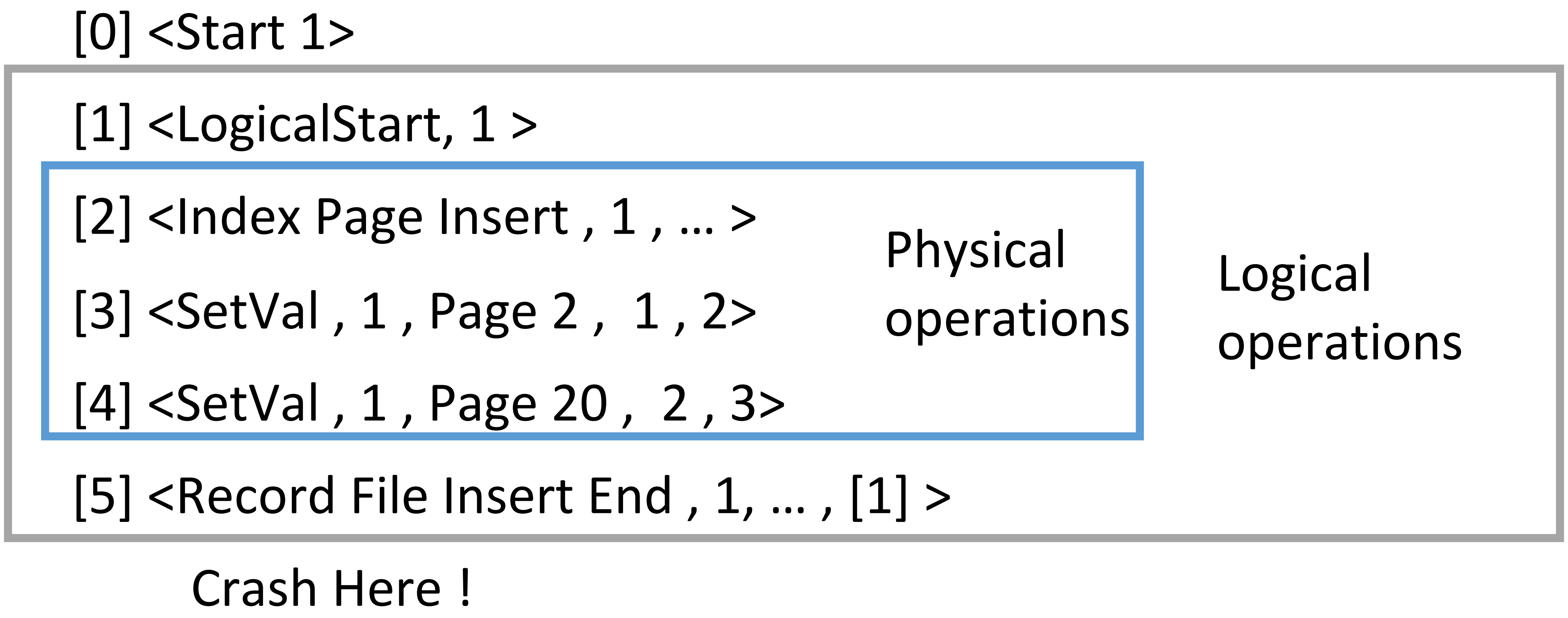
[4] <SetVal , 1 , Page 20 , 2 , 3>

[5] <Record File Insert End , 1, … , [1] >

Logical operations

Crash Here !

Rollback a completed Logical Log Record



[0] <Start 1>

[1] <LogicalStart, 1 >

[2] <Index Page Insert , 1 , … >

[3] <SetVal , 1 , Page 2 , 1 , 2>

[4] <SetVal , 1 , Page 20 , 2 , 3>

[5] <Record File Insert End , 1, … , [1] >

Crash Here !

Physical operations

Logical operations

Rollback a completed Logical Log Record

[0] <Start 1>

[1] <LogicalStart, 1 >

[2] <Index Page **Insert** , 1 , … >

[3] <SetVal , 1 , Page 2 , 1 , 2>

[4] <SetVal , 1 , Page 20 , 2 , 3>

[5] <Record File Insert End , 1, … , [1] >

Crash Here !

[6] <Start 2 >

[7] <LogicalStart, 2 >

[8] <Index Page **Delete** , 2 , … >

[9] <SetVal , 2 , Page 2 , 2 , 1>

[10] <SetVal , 2 , Page 20 , 3 , 2>

Physical operations

[11] <Record File Delete End , 2, … , [7]>

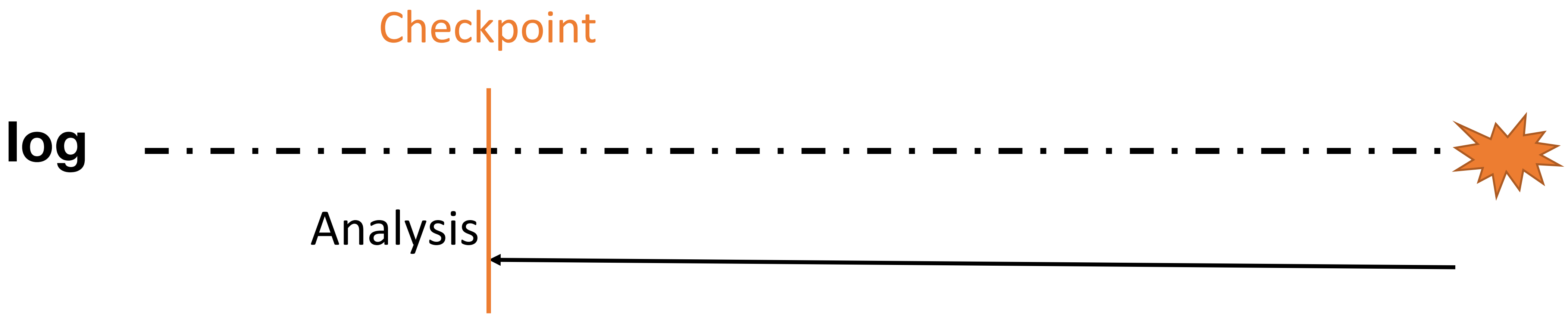
Logical operations

[12] <Logical Abort 1 , [1]>

Recovery Phases of ARIES

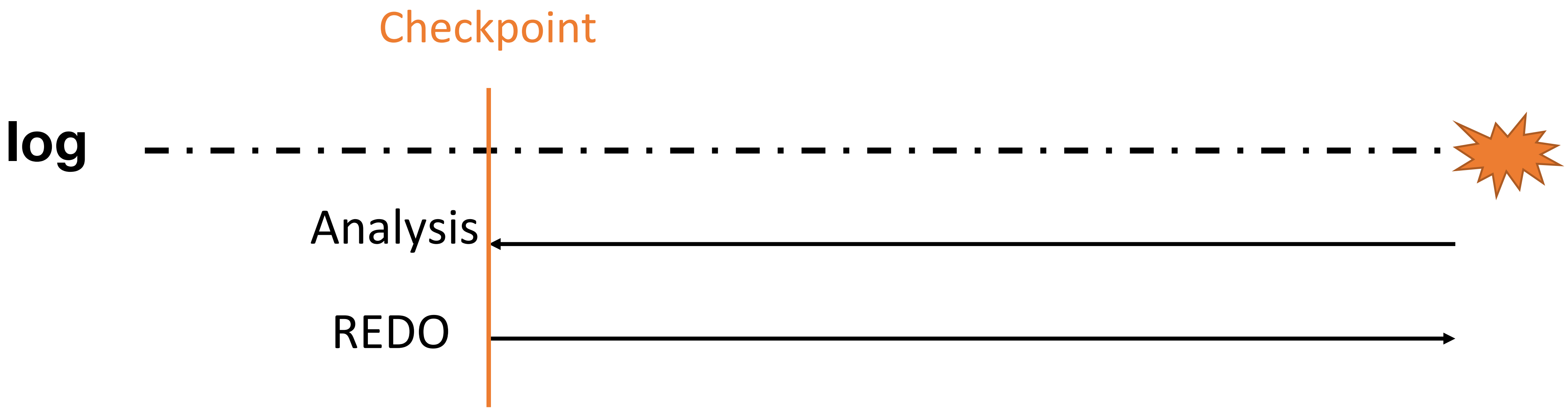
Recovery Phases of ARIES

- Analysis Phase
  - Find the earliest possible start point of dirty page
  - Find loser txs



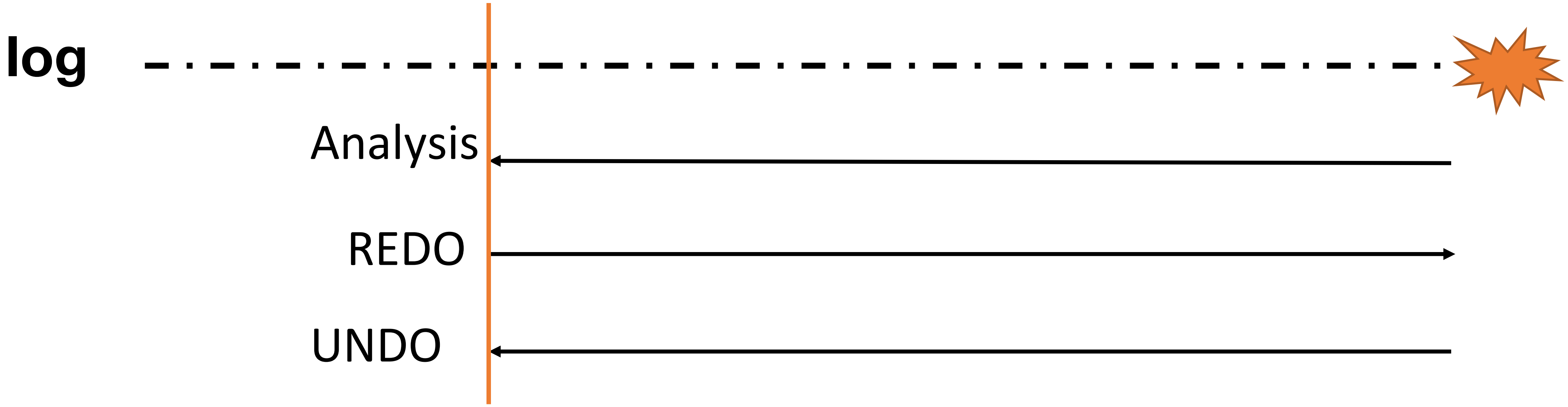
Recovery Phases of ARIES

- Analysis Phase
  - Find the earliest possibly start point of dirty page
  - Find loser txs
- REDO Phase
  - Repeat history (*both* winner and loser changes)
  - Recovery exact page status when the failure occurred

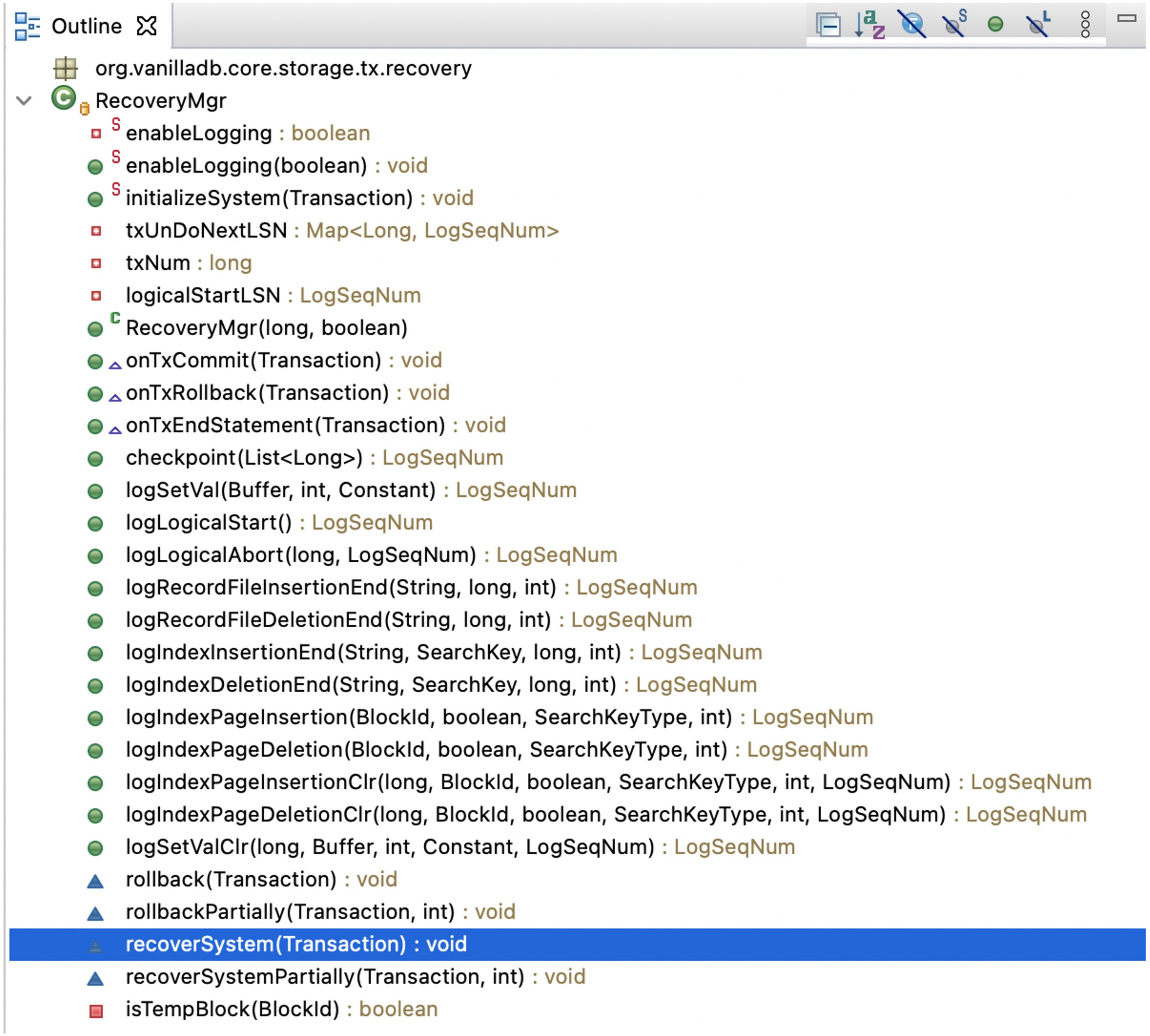


Recovery Phases of ARIES

- Analysis Phase
  - Find the earliest possibly start point of dirty page
  - Find loser txs
- REDO Phase
  - Repeat history (*both* winner and loser changes)
  - Recovery exact page status when the failure occurred
- UNDO Phase
  - Rollback *loser* txs changes



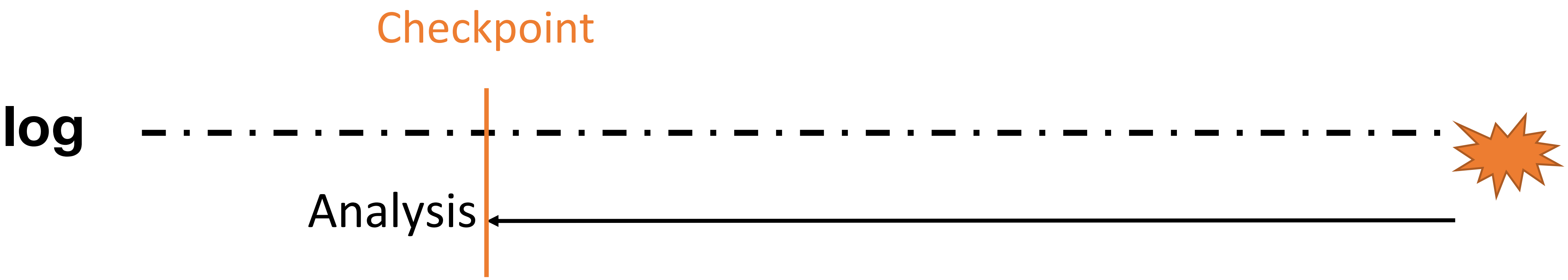
Recovery in VanillaDB

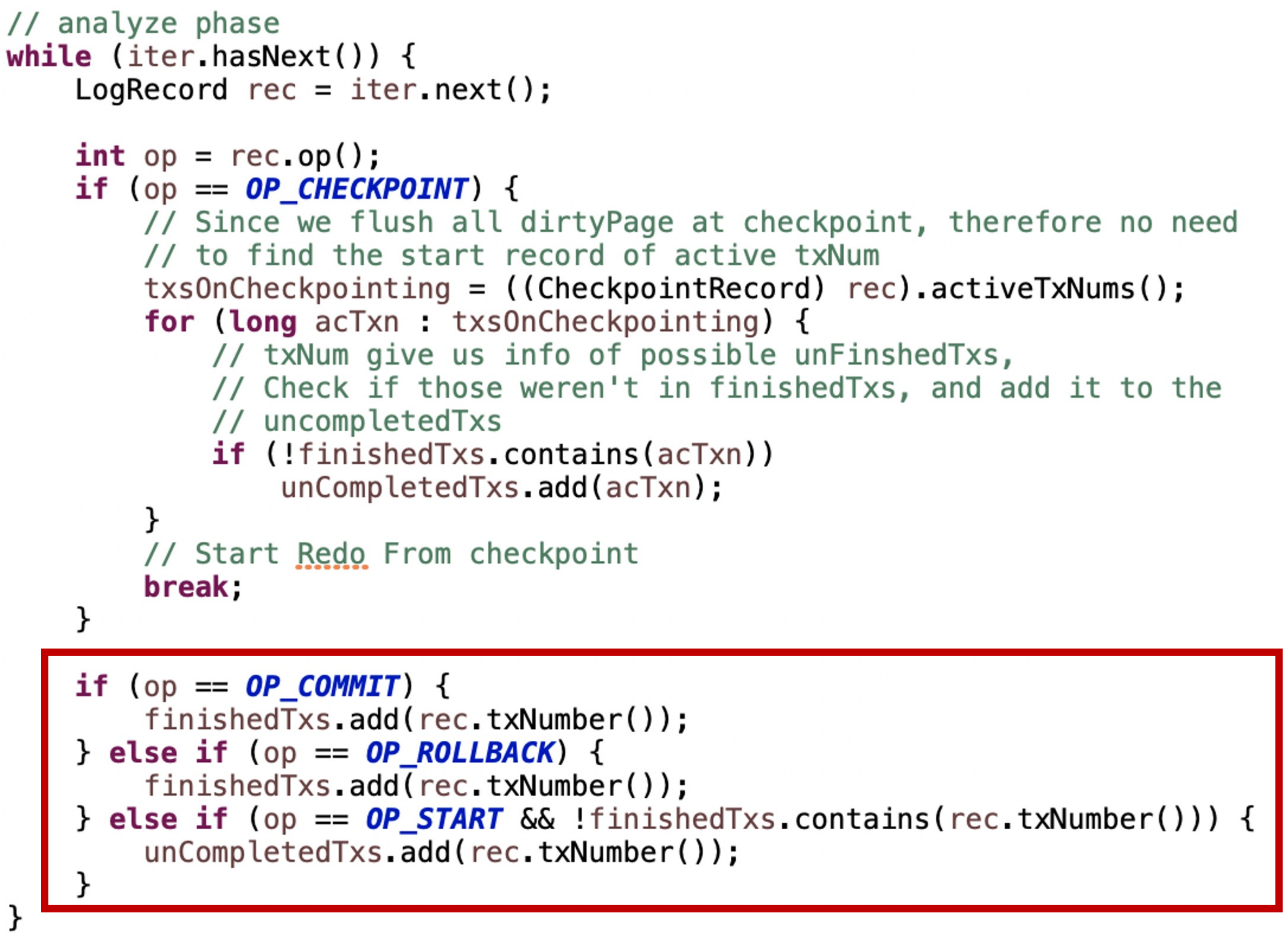


Outline

```
org.vanilladb.core.storage.tx.recovery
RecoveryMgr
    S enableLogging : boolean
    S enableLogging(boolean) : void
    S initializeSystem(Transaction) : void
    txUnDoNextLSN : Map<Long, LogSeqNum>
    txNum : long
    logicalStartLSN : LogSeqNum
    C RecoveryMgr(long, boolean)
    onTxCommit(Transaction) : void
    onTxRollback(Transaction) : void
    onTxEndStatement(Transaction) : void
    checkpoint(List<Long>) : LogSeqNum
    logSetVal(Buffer, int, Constant) : LogSeqNum
    logLogicalStart() : LogSeqNum
    logLogicalAbort(long, LogSeqNum) : LogSeqNum
    logRecordFileInsertionEnd(String, long, int) : LogSeqNum
    logRecordFileDeletionEnd(String, long, int) : LogSeqNum
    logIndexInsertionEnd(String, SearchKey, long, int) : LogSeqNum
    logIndexDeletionEnd(String, SearchKey, long, int) : LogSeqNum
    logIndexPageInsertion(BlockId, boolean, SearchKeyType, int) : LogSeqNum
    logIndexPageDeletion(BlockId, boolean, SearchKeyType, int) : LogSeqNum
    logIndexPageInsertionClr(long, BlockId, boolean, SearchKeyType, int, LogSeqNum) : LogSeqNum
    logIndexPageDeletionClr(long, BlockId, boolean, SearchKeyType, int, LogSeqNum) : LogSeqNum
    logSetValClr(long, Buffer, int, Constant, LogSeqNum) : LogSeqNum
    rollback(Transaction) : void
    rollbackPartially(Transaction, int) : void
    recoverSystem(Transaction) : void
    recoverSystemPartially(Transaction, int) : void
    isTempBlock(BlockId) : boolean
```

Analysis Phase



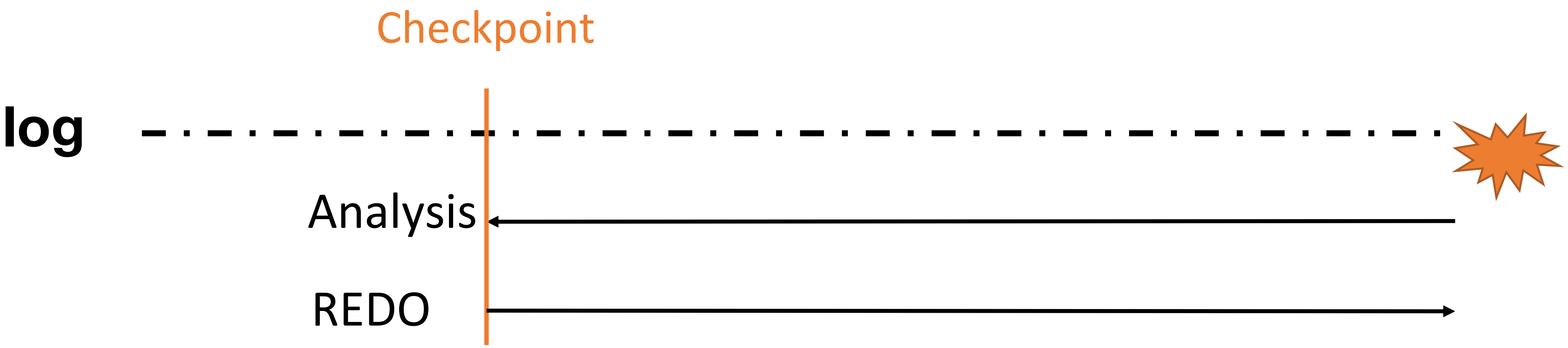


```
// analyze phase
while (iter.hasNext()) {
    LogRecord rec = iter.next();

    int op = rec.op();
    if (op == OP_CHECKPOINT) {
        // Since we flush all dirtyPage at checkpoint, therefore no need
        // to find the start record of active txNum
        txsOnCheckpointing = ((CheckpointRecord) rec).activeTxNums();
        for (long acTxn : txsOnCheckpointing) {
            // txNum give us info of possible unFinshedTxs,
            // Check if those weren't in finishedTxs, and add it to the
            // uncompletedTxs
            if (!finishedTxs.contains(acTxn))
                unCompletedTxs.add(acTxn);
        }
        // Start Redo From checkpoint
        break;
    }

    if (op == OP_COMMIT) {
        finishedTxs.add(rec.txNumber());
    } else if (op == OP_ROLLBACK) {
        finishedTxs.add(rec.txNumber());
    } else if (op == OP_START && !finishedTxs.contains(rec.txNumber())) {
        unCompletedTxs.add(rec.txNumber());
    }
}
```

Redo Phase

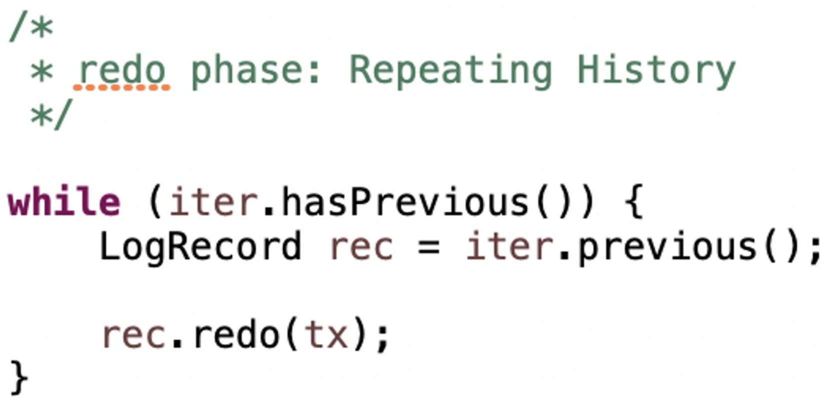


Checkpoint

log

Analysis

REDO

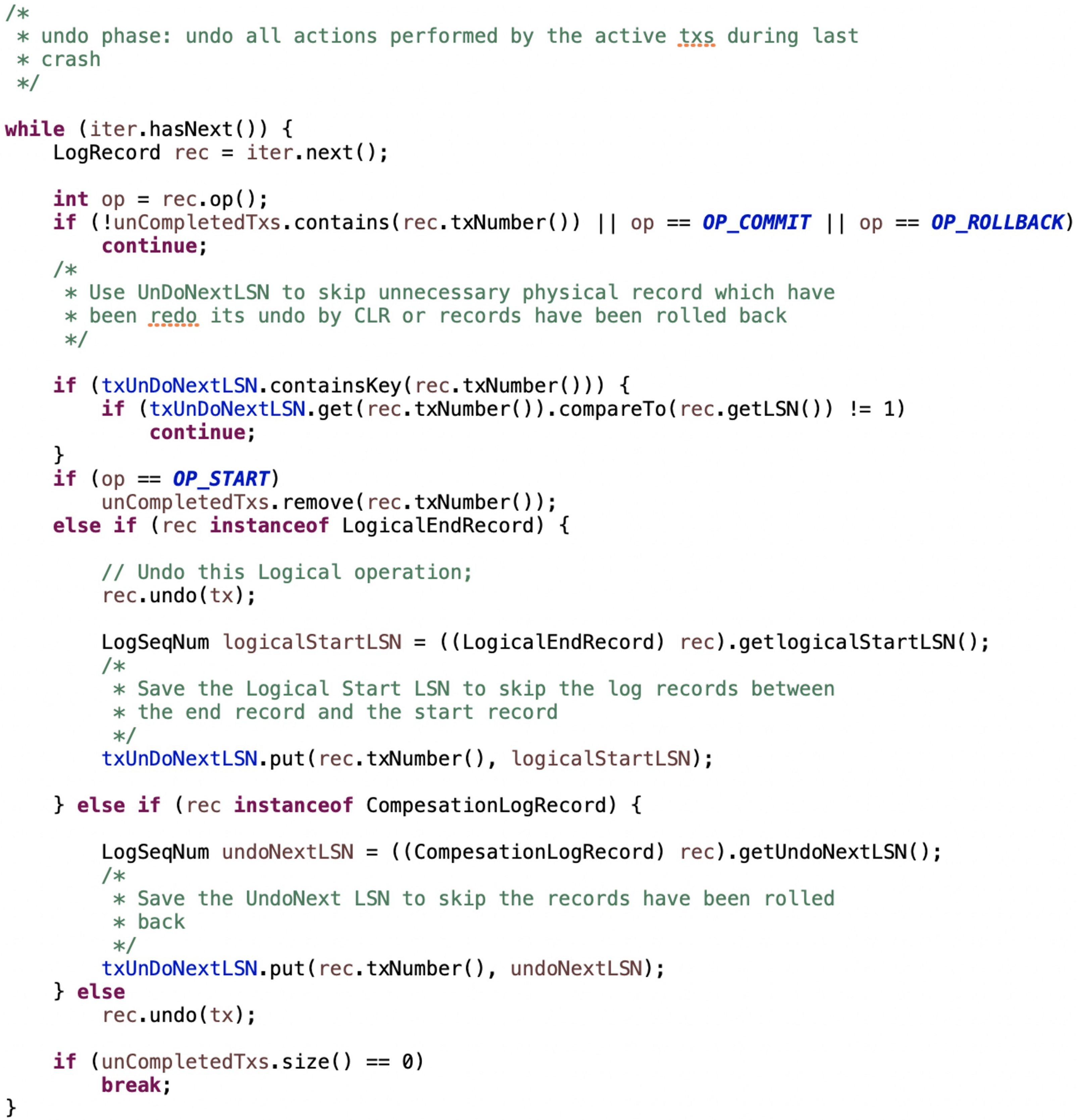


```
/*
 * redo phase: Repeating History
 */

while (iter.hasPrevious()) {
    LogRecord rec = iter.previous();

    rec.redo(tx);
}
```

Undo Phase



```
/*
 * undo phase: undo all actions performed by the active txs during last
 * crash
 */

while (iter.hasNext()) {
    LogRecord rec = iter.next();

    int op = rec.op();
    if (!unCompletedTxs.contains(rec.txNumber()) || op == OP_COMMIT || op == OP_ROLLBACK)
        continue;
    /*
     * Use UnDoNextLSN to skip unnecessary physical record which have
     * been redo its undo by CLR or records have been rolled back
     */

    if (txUnDoNextLSN.containsKey(rec.txNumber())) {
        if (txUnDoNextLSN.get(rec.txNumber()).compareTo(rec.getLSN()) != 1)
            continue;
    }
    if (op == OP_START)
        unCompletedTxs.remove(rec.txNumber());
    else if (rec instanceof LogicalEndRecord) {

        // Undo this Logical operation;
        rec.undo(tx);

        LogSeqNum logicalStartLSN = ((LogicalEndRecord) rec).getlogicalStartLSN();
        /*
         * Save the Logical Start LSN to skip the log records between
         * the end record and the start record
         */
        txUnDoNextLSN.put(rec.txNumber(), logicalStartLSN);

    } else if (rec instanceof CompesationLogRecord) {

        LogSeqNum undoNextLSN = ((CompesationLogRecord) rec).getUndoNextLSN();
        /*
         * Save the UndoNext LSN to skip the records have been rolled
         * back
         */
        txUnDoNextLSN.put(rec.txNumber(), undoNextLSN);
    } else
        rec.undo(tx);

    if (unCompletedTxs.size() == 0)
        break;
}
```

Reference

- ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging