

Implementing Recovery

- Recovery
- Logging
- Undo Logging
- Checkpointing
- Redo Logging
- Undo/Redo Logging
- Recovery in PostgreSQL

❖ Recovery

For a DBMS to recover from a system failure, it needs

- a mechanism to record what updates were "in train" at failure time
- methods for restoring the database(s) to a valid state afterwards

Assume multiple transactions are running when failure occurs

- uncommitted transactions need to be rolled back (**ABORT**)
- committed, but not yet finalised, tx's need to be completed

A critical mechanism in achieving this is the **transaction (event) log**

❖ Logging

Three "styles" of logging

- **undo** ... removes changes by any uncommitted tx's
- **redo** ... repeats changes by any committed tx's
- **undo/redo** ... combines aspects of both

All approaches require:

- a sequential file of log records
- each log record describes a change to a data item
- log records are written *before* changes to data
- actual changes to data are written later

Known as **write-ahead logging** (PostgreSQL uses WAL)

❖ Undo Logging

Simple form of logging which ensures atomicity.

Log file consists of a **sequence** of small records:

- **<START T>** ... transaction **T** begins
- **<COMMIT T>** ... transaction **T** completes successfully
- **<ABORT T>** ... transaction **T** fails (no changes)
- **<T, X, v>** ... transaction **T** changed value of **X** from **v**

Notes:

- we refer to **<T, X, v>** generically as **<UPDATE>** log records
- update log entry created for each **WRITE** (not **OUTPUT**)
- update log entry contains *old* value (new value is not recorded)

❖ Undo Logging (cont)

Data must be written to disk in the following order:

1. **<START>** transaction log record
2. **<UPDATE>** log records indicating changes
3. the changed data elements themselves
4. **<COMMIT>** log record

Note: sufficient to have **<T, X, v>** output before **X**, for each **X**

❖ Undo Logging (cont)

For the example transaction, we would get:

t	Action	v	B(A)	B(B)	D(A)	D(B)	Log
(0)	BEGIN	.	.	.	8	5	<START T>
(1)	READ(A,v)	8	8	.	8	5	
(2)	v = v*2	16	8	.	8	5	
(3)	WRITE(A,v)	16	16	.	8	5	<T,A,8>
(4)	READ(B,v)	5	16	5	8	5	
(5)	v = v+1	6	16	5	8	5	
(6)	WRITE(B,v)	6	16	6	8	5	<T,B,5>
(7)	FlushLog						
(8)	StartCommit						
(9)	OUTPUT(A)	6	16	6	16	5	
(10)	OUTPUT(B)	6	16	6	16	6	
(11)	EndCommit						<COMMIT T>
(12)	FlushLog						

Note that T is not regarded as committed until (12) completes.

❖ Undo Logging (cont)

Simplified view of recovery using UNDO logging:

- scan **backwards** through log
 - if **<COMMIT T>**, mark **T** as committed
 - if **<T, X, v>** and **T** not committed, set **X** to **v** on disk
 - if **<START T>** and **T** not committed, put **<ABORT T>** in log

Assumes we scan entire log; use checkpoints to limit scan.

❖ Undo Logging (cont)

Algorithmic view of recovery using UNDO logging:

```
committedTrans = abortedTrans = startedTrans = {}
for each log record from most recent to oldest {
    switch (log record) {
        <COMMIT T> : add T to committedTrans
        <ABORT T>  : add T to abortedTrans
        <START T>  : add T to startedTrans
        <T,X,v>    : if (T in committedTrans)
                    // don't undo committed changes
                    else // roll-back changes
                        { WRITE(X,v); OUTPUT(X) }
    }
}
for each T in startedTrans {
    if (T in committedTrans) ignore
    else if (T in abortedTrans) ignore
    else write <ABORT T> to log
}
flush log
```


COMP9315 21T1 ◇ Implementing Recovery ◇ [7/20]

❖ Checkpointing

Simple view of recovery implies reading entire log file.

Since log file grows without bound, this is infeasible.

Eventually we can delete "old" section of log.

- i.e. where **all** prior transactions have committed

This point is called a **checkpoint**.

- all of log prior to checkpoint can be ignored for recovery

❖ Checkpointing (cont)

Problem: many concurrent/overlapping transactions.

How to know that all have finished?

1. periodically, write log record **<CHKPT (T1, ..., Tk)>**
(contains references to all active transactions \Rightarrow active tx table)
2. continue normal processing (e.g. new tx's can start)
3. when all of **T1, ..., Tk** have completed,
write log record **<ENDCHKPT>** and flush log

Note: tx manager maintains chkpt and active tx information

❖ Checkpointing (cont)

Recovery: scan backwards through log file processing as before.

Determining where to stop depends on ...

- whether we meet **<ENDCHKPT>** or **<CHKPT . . . >** first

If we encounter **<ENDCHKPT>** first:

- we know that all incomplete tx's come after prev **<CHKPT . . . >**
- thus, can stop backward scan when we reach **<CHKPT . . . >**

If we encounter **<CHKPT (T1, . . . , Tk) >** first:

- crash occurred *during* the checkpoint period
- any of **T1, . . . , Tk** that committed before crash are ok
- for uncommitted tx's, need to continue backward scan

❖ Redo Logging

Problem with UNDO logging:

- all changed data must be output to disk before committing
- conflicts with optimal use of the buffer pool

Alternative approach is **redo** logging:

- allow changes to remain only in buffers after commit
- write records to indicate what changes are "pending"
- after a crash, can apply changes during recovery

❖ Redo Logging (cont)

Requirement for redo logging: **write-ahead rule**.

Data must be written to disk as follows:

1. start transaction log record
2. update log records indicating changes
3. then commit log record (**OUTPUT**)
4. then **OUTPUT** changed data elements themselves

Note that update log records now contain $\langle T, x, v' \rangle$, where v' is the *new* value for x .

❖ Redo Logging (cont)

For the example transaction, we would get:

t	Action	v	B(A)	B(B)	D(A)	D(B)	Log
(0)	BEGIN	.	.	.	8	5	<START T>
(1)	READ(A,v)	8	8	.	8	5	
(2)	v = v*2	16	8	.	8	5	
(3)	WRITE(A,v)	16	16	.	8	5	<T,A,16>
(4)	READ(B,v)	5	16	5	8	5	
(5)	v = v+1	6	16	5	8	5	
(6)	WRITE(B,v)	6	16	6	8	5	<T,B,6>
(7)	COMMIT						<COMMIT T>
(8)	FlushLog						
(9)	OUTPUT(A)	6	16	6	16	5	
(10)	OUTPUT(B)	6	16	6	16	6	

Note that T is regarded as committed as soon as (8) completes.

❖ Redo Logging (cont)

Simplified view of recovery using REDO logging:

- identify all committed tx's (backwards scan)
- scan **forwards** through log
 - if **<T, X, v>** and **T** is committed, set **X** to **v** on disk
 - if **<START T>** and **T** not committed, put **<ABORT T>** in log

Assumes we scan entire log; use checkpoints to limit scan.

❖ Undo/Redo Logging

UNDO logging and REDO logging are incompatible in

- order of outputting **<COMMIT T>** and changed data
- how data in buffers is handled during checkpoints

Undo/Redo logging combines aspects of both

- requires new kind of update log record
<T, X, v, v' > gives both old and new values for **X**
- removes incompatibilities between output orders

As for previous cases, requires write-ahead of log records.

Undo/redo logging is common in practice; Aries algorithm.

❖ Undo/Redo Logging (cont)

For the example transaction, we might get:

t	Action	v	B(A)	B(B)	D(A)	D(B)	Log
(0)	BEGIN	.	.	.	8	5	<START T>
(1)	READ(A,v)	8	8	.	8	5	
(2)	v = v*2	16	8	.	8	5	
(3)	WRITE(A,v)	16	16	.	8	5	<T,A,8,16>
(4)	READ(B,v)	5	16	5	8	5	
(5)	v = v+1	6	16	5	8	5	
(6)	WRITE(B,v)	6	16	6	8	5	<T,B,5,6>
(7)	FlushLog						
(8)	StartCommit						
(9)	OUTPUT(A)	6	16	6	16	5	
(10)							<COMMIT T>
(11)	OUTPUT(B)	6	16	6	16	6	

Note that T is regarded as committed as soon as (10) completes.

❖ Undo/Redo Logging (cont)

Simplified view of recovery using UNDO/REDO logging:

- scan log to determine committed/uncommitted txs
- for each uncommitted tx **T** add **<ABORT T>** to log
- scan **backwards** through log
 - if **<T, X, v, w>** and **T** is not committed, set **X** to **v** on disk
- scan **forwards** through log
 - if **<T, X, v, w>** and **T** is committed, set **X** to **w** on disk

❖ Undo/Redo Logging (cont)

The above description simplifies details of undo/redo logging.

Aries is a complete algorithm for undo/redo logging.

Differences to what we have described:

- log records contain a sequence number (LSN)
- LSNs used in tx and buffer managers, and stored in data pages
- additional log record to mark **<END>** (of commit or abort)
- **<CHKPT>** contains only a timestamp
- **<ENDCHKPT . . >** contains tx and dirty page info

❖ Recovery in PostgreSQL

PostgreSQL uses write-ahead undo/redo style logging.

It also uses multi-version concurrency control, which

- tags each record with a tx and update timestamp

MVCC simplifies some aspects of undo/redo, e.g.

- some info required by logging is already held in each tuple
- no need to undo effects of aborted tx's; use old version

❖ Recovery in PostgreSQL (cont)

Transaction/logging code is distributed throughout backend.

Core transaction code is in **src/backend/access/transam**.

Transaction/logging data is written to files in **PGDATA/pg_wal**

- a number of very large files containing log records
- old files are removed once all txs noted there are completed
- new files added when existing files reach their capacity (16MB)
- number of tx log files varies depending on tx activity

Produced: 20 Apr 2021