

Recovery Model

Recovery Algorithm

用于保证数据库的一致性，原子性和持久性

包含两个部分：

- 在事务运行时做的操作，确保崩溃能够恢复
- 故障发生后数据库进行恢复的操作，保证原子性一致性和持久性

In-Memory Database Recovery

比disk-oriented DBMS更要简单，因为：

崩溃后，内存中所有记录都删掉，从磁盘上的checkpoint处加载

- 不用跟踪硬盘上的脏页
- 不需要存储undo log，只需要redo log
- 不需要记录索引修改的log，因为recovery时候是直接在内存中重建索引，不需要从磁盘上加载索引

但是瓶颈仍然是非易失性存储的慢sync速度。

Logging Schemes

LOGGING SCHEMES

Approach #1: Physical Logging

- Record the changes made to a specific record in the database.
- Example: Store the original value and after value for an attribute that is changed by a query.

Approach #2: Logical Logging

- Record the high-level operations executed by txns.
- Example: The **UPDATE**, **DELETE**, and **INSERT** queries invoked by a txn.

举个例子，假如对1billion个tuple进行更新操作,第一种方法需要存储1billion个记录，第二种方法只需要存储1项记录即你更新做的操作。

但是第二种方法的缺点是recovery的过程可能会很慢，因为你是在replay操作，不像第一种方法直接更改就行。

所以大部分系统都是用的第一种方法。

LOG FLUSHING

LOG FLUSHING

Approach #1: All-at-Once Flushing

- Wait until a txn has fully committed before writing out log records to disk.
- Do not need to store abort records because uncommitted changes are never written to disk.

Approach #2: Incremental Flushing

- Allow the DBMS to write a txn's log records to disk before it has committed.

- 方法一：事务完全提交之后，一次性把所有log record落盘
- 方法二：允许在事务提交前就写事务的log

假如要更新1billion条数据，第一种方法需要buffer能够支持存储1billion条数据，内存可能不够用。因此通常采用第二种方案

组提交

把多组事务的日志刷盘聚集在一个同一个 `fsync` 进行

- 日志要么在超时之后或者在buffer满了之后才会刷盘

分摊了多个事务的IO平均开销

EARLY LOCK RELEASE

EARLY LOCK RELEASE

A txn's locks can be released before its commit record is written to disk if it does not return results to the client before becoming durable.

Other txns that speculatively read data updated by a **pre-committed** txn become dependent on it and must wait for their predecessor's log records to reach disk.

For more info: <https://dzone.com/articles/early-lock-release>

Standard transaction	Early Lock Release Transaction
<ul style="list-style-type: none">• Take locks• Perform work• Commit transaction in memory• Flush transaction to log file• Release locks• Notify user that the transaction successfully completed	<ul style="list-style-type: none">• Take locks• Perform work• Commit transaction in memory• Flush transaction to log file async• Release locks• Notify user that the transaction successfully completed when the log flush competes

在开始写transaction log的时候就释放锁，而不是写完才释放锁。这样可以提高并发速度。如果落盘失败了，那么该事务及其之后的txn都会被abort

USE MVCC TIME-TRAVEL TABLE AS RECOVERY LOG

MSSQL CONSTANT TIME RECOVERY

Physical logging protocol that uses the DBMS's MVCC **time-travel** table as the recovery log.

- The version store is a persistent append-only storage area that is flushed to disk.
- Leverage versions meta-data to "undo" updates without having to process undo records in WAL.

Recovery time is measured based on the number of version store records that must be read from disk.

大意是利用MVCC的历史版本数据来替代WAL中的undo log，尝试把恢复的时间复杂度降低为常数级，即只和需要读取的版本数量有关。

version的存储方式又可以分为两种:

Approach #1: In-row Versioning

- Store small updates to a tuple as a delta record embedded with the latest version in the main table.
- Same as Cicada "best-effort in-lining" technique.

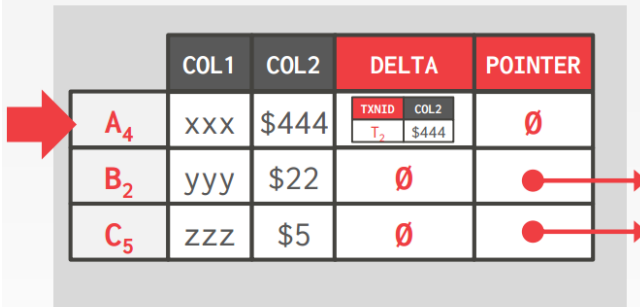
Approach #2: Off-row Versioning

- Specialized data table to store the old versions that is optimized for concurrent inserts.
- Versions from all tables are stored in a single table.
- Store redo records for inserts on this table in WAL.

其中In-row versioning: 把事务号和旧的版本数据直接存在原Table的DELTA列中

MSSQL CTR: IN-ROW VERSIONING

Main Table



	COL1	COL2	DELTA	POINTER				
A₄	xxx	\$444	<table><tr><th>TXNID</th><th>COL2</th></tr><tr><td>T₂</td><td>\$444</td></tr></table>	TXNID	COL2	T ₂	\$444	∅
TXNID	COL2							
T ₂	\$444							
B₂	yyy	\$22	∅	● →				
C₅	zzz	\$5	∅	● →				

Store small updates to a tuple as a delta record embedded with the latest version in the main table.

The delta record space is **not** pre-allocated per tuple in a disk-oriented DBMS.

Recovery Protocols

- Phase1:

从checkpoint加载看是哪个transacion开始。

- Redo:

重做，恢复main table与version store的状态为崩溃发生时的状态。

- Undo:

把未提交的事务标记为aborted，然后通过**logical revert**来增量删除老的版本。

Logical Revert

MSSQL CTR: LOGICAL REVERT

Approach #1: Background Cleanup

- GC thread scans all blocks and removes reclaimable versions.
- If latest version in main table is from an aborted txn, then it will move the committed version back to main table.

Approach #2: Aborted Version Overwrite

- Txns can overwrite the latest version in the main table if that version is from an aborted txn.

CHECKPOINT PROTOCOL

IN-MEMORY CHECKPOINTS

The different approaches for how the DBMS can create a new checkpoint for an in-memory database are tightly coupled with its concurrency control scheme.

The checkpoint thread(s) scans each table and writes out data asynchronously to disk.

checkpoint的理想情况

- 不应该影响正常事务的处理速度
- 不能引入不能接受的延迟
- 不能引入太多的内存开销

CONSISTENT VS. FUZZY CHECKPOINTS

Approach #1: Consistent Checkpoints

- Represents a consistent snapshot of the database at some point in time. No uncommitted changes.
- No additional processing during recovery.

Approach #2: Fuzzy Checkpoints

- The snapshot could contain records updated from transactions that committed after the checkpoint started.
- Must do additional processing to figure out whether the checkpoint contains all updates from those txns.

CHECKPOINT MECHANISM

CHECKPOINT MECHANISM

Approach #1: Do It Yourself

- The DBMS is responsible for creating a snapshot of the database in memory.
- Can leverage multi-versioned storage to find snapshot.

Approach #2: OS Fork Snapshots

- Fork the process and have the child process write out the contents of the database to disk.
- This copies everything in memory.
- Requires extra work to remove uncommitted changes.

父进程fork出一个子进程，子进程的内存会拥有和父进程一模一样的拷贝。而且两者的内存视角是隔离的，父进程可以继续进程事务操作修改数据，而子进程是看不到父进程的修改。这就是Redis的RDB做法。

HYPER – OS FORK SNAPSHOTS

Create a snapshot of the database by forking the DBMS process.

- Child process contains a consistent checkpoint if there are not active txns.
- Otherwise, use the in-memory undo log to roll back txns in the child process.

Continue processing txns in the parent process.

那我们的checkpoint具体要存储什么呢？

CHECKPOINT CONTENTS

Approach #1: Complete Checkpoint

- Write out every tuple in every table regardless of whether were modified since the last checkpoint.

Approach #2: Delta Checkpoint

- Write out only the tuples that were modified since the last checkpoint.
- Can merge checkpoints together in the background.

checkpoint的频率呢？

Approach #1: Time-based

- Wait for a fixed period of time after the last checkpoint has completed before starting a new one.

Approach #2: Log File Size Threshold

- Begin checkpoint after a certain amount of data has been written to the log file.

Approach #3: On Shutdown (Mandatory)

- Perform a checkpoint when the DBA instructs the system to shut itself down. Every DBMS (hopefully) does this.

Restart Protocols

Not all DBMS restarts are due to crashes.

- Updating OS libraries
- Hardware upgrades/fixes
- Updating DBMS software

Need a way to be able to quickly restart the DBMS without having to re-read the entire database from disk again.

- SHARED MEMORY RESTART

Approach #1: Shared Memory Heaps

- All data is allocated in SM during normal operations.
- Have to use a custom allocator to subdivide memory segments for thread safety and scalability.
- Cannot use lazy allocation of backing pages with SM.

Approach #2: Copy on Shutdown

- All data is allocated in local memory during normal operations.
- On shutdown, copy data from heap to SM.

In summary

- Physical logging is a general-purpose approach (物理日志存储的是页中改变的字节, 逻辑日志存储的是改变的操作)
- Copy-on-update checkpoints are the way to go especially if you are using MVCC