# COMP90050 Advanced Database Systems

1. Database architectures (both hardware and software issues). For example, you should be able to compute memory or disk access times with and without caches to understand performance issues. You should know spatial and temporal locality how these issues are handled in database systems.

## ① 什么影响了DB Performance

(1) Hardware:
- The speed of processor
- Number of processors
- Number of disk drives and I/O bandwidth
- Storage Systems (e.g. different RAID levels)
- Size of main memory
- Type of architecture (e.g. Centralised DB, Client-server DB, Distributed DB)

(2) Software:
- Database Technologies (e.g. Simple file system like UNIX, RDBMS, OODB, NoSQL)
- Application itself (e.g. code quality, using threads)
- Message passing (Session based or datagrams)
- Balances between utilisation and response time

## ② 计算access time

(1) Effective memory access time:
- Hit ratio = Number of cache hits / (Number of cache hits + Number of cache misses)
- $EA = H * C + (1 - H) * S$
- H = hit ratio, C = cache access time, S = memory access time

(2) Disk access time:
- Disk access time = seek time + rotational time + (transfer length / bandwidth)
- EA一样，C = buffer access time, S = disk access time

## ③ Temporal locality and Spatial locality

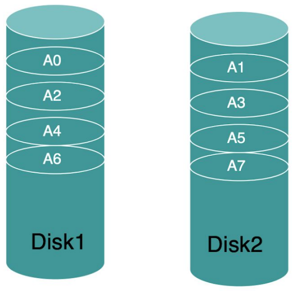
(1) Temporal locality: 时间局部性 If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.
(2) Spatial locality: 空间局部性 If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.
(3) DB如何handle:
- 时间局部性store a copy of the referenced data in faster memory storage
- 空间局部性attempt to guess the size and shape of the area around the current reference, using larger cache blocks

## ③ RAID

| RAID0 | RAID1 | RAID2 | RAID3 | RAID4 | RAID5 | RAID6 |
|---|---|---|---|---|---|---|
| Block | Mirroring | Bit | Byte | Block | Block | Block |
| 横着顺着写A | 两个完全一样A | 横着顺着写b | 前俩横顺写B第三个P | 前俩横顺写A第三个P | 右上到左下斜着填P | 5块右上到左下斜2P |
| MTTF/2 | MTTF^2 | Half RAID0 | MTTF^2 /3 | MTTF^2 /3 | MTTF^2 /3 | MTTF^3 /10 |
| Throughput *2 | High read Low write | Higher transfer rate | Higher transfer rate | Higher Throughput Very low write | Higher Throughput Low write but better than 4 | Any two disk failure can safe recover |

### RAID 0 (Block level Striping)

- A means Block (4K or 8K bytes of storage)
- B means Byte
- b means bit
- P is parity

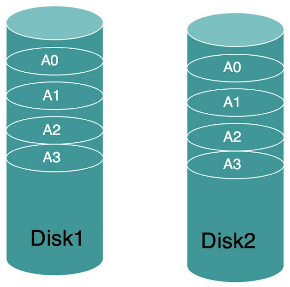


A0, A1, A2, A3, .. are contiguous blocks of data of a file

- Provides balanced i/o of disk drives – throughput approximately doubles
- Any one of the of disks failure can be catastrophic and MTTF (mean time to failure) reduces by a factor of 2

### RAID 1 (mirroring)

- A means Block (4K or 8K bytes of storage)

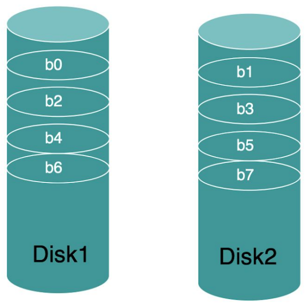


A0, A1, A2, A3, .. are logically contiguous blocks of data of a file

- Provides higher read throughput but lower write throughput (half of the speed – i.e. single disk speed) and half storage utilization.
- MTTF increases substantially (quadratic improvement –i.e. MTTF$^2$!)

## RAID 2 (bit level striping)

- b means bit

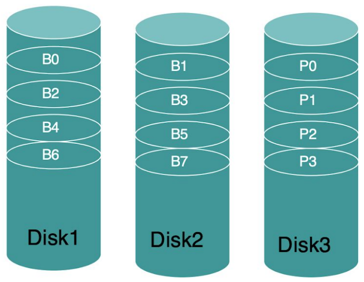


b0, b1, b2, b3, .. are contiguous bits of data of file

- Striping takes place at bit level
- Provides higher transfer rate (double the single disk)
- MTTF decreases by half as in RAID 0

## RAID 3 (Byte level striping)

- B means Byte
- P is parity



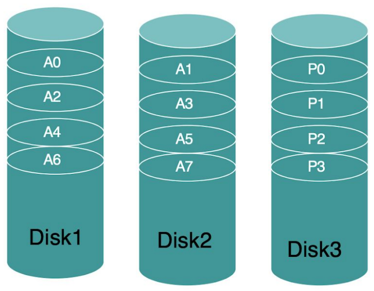
B0, B1, B2, B3, .. are bytes of data of file

- Striping takes place at byte level
- Provides higher transfer rate as in RAID 0
- MTTF increases substantially (1/3 of RAID1 = $MTTF^2/3$)
- P0 is parity for bytes B0 and B1
- $P_i = B_{2i} \oplus B_{2i+1}$, here $\oplus$ is exclusive-or operator

- MTTF = Mean Time To Failure

## RAID 4 (Block level level striping)

- A means Block (4K or 8K bytes of storage)
- P is parity



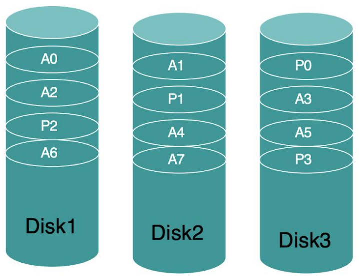
- A0, A1, A2, A3, .. are blocks of data of file
- Striping takes place at block level as in RAID 0
- Dedicated disk for parity blocks
- Provides higher throughput but very slow writes. Note: Disk3 has more writes than either Disk1 or Disk2 as Parity has to be updated on every write operation.
- MTTF increases substantially (same as RAID3)
- P0 is parity block for blocks A0 and A1
- $Pi = A_{2i} \oplus A_{2i+1}$, here $\oplus$ is an exclusive-or operator

- MTTF = Mean Time To Failure

## RAID 5 with 3 disks (Block level level striping)

•A means Block (4K or 8K bytes of storage)

•P is parity



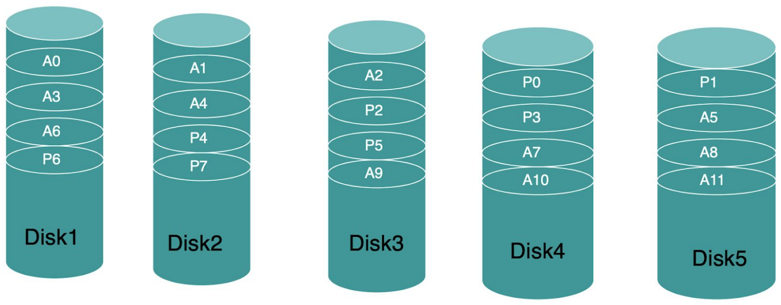
- A0,A1, A2, A3, .. are contiguous blocks of data of a file
- Striping takes place at block level as in RAID 0
- No dedicated disk for parity blocks, Parity blocks are also striped
- Provides higher throughput but slower writes but better than RAID 4 as Parity bits are distributed among all disks and the number of write operations on average equal among all 3 disks.
- MTTF increases substantially (same as RAID3)
- P0 is parity block for blocks A0 and A1
- Pi = $A_{2i} \oplus A_{2i+1}$, here $\oplus$ is an exclusive-or operator

• MTTF = Mean Time To Failure

## RAID 6 (Block level level striping)

• A means Block (4K or 8K bytes of storage)

• P is parity



• Similar to RAID 5 except two parity bocks used.

• Reliability is of the order of $MTTF^3/10$

• P1 and P2 are parity blocks for blocks A0, A1 and A2. These are computed in such way that any two disc failures can be safe to recover the data.

## 2. ACID Properties: You should know all these properties and do not describe them in the exam unless explicitly asked. You should have a clear understanding how these properties can be enforced by what mechanism. We have discussed these issues many times in the lectures and quite extensively.

### ① **Atomicity**

(1) Transaction may abort.
(2) Two phase commit 保证在distributed DB中的atomicity，包括recovery.
(3) Atomic disk writes such as duplex write guarantees at least one block has consistent data.
(4) Write-Ahead Logging (WAL) must force the log record for an update before the data page gets to disk, guarantees atomicity 在crash后用于undo.

### ② Consistency

Atomicity, isolation and durability are conditions of consistency.

### ③ Isolation

(1) All transactions are two phase locking
(2) All transactions are well-formed transactions
(3) History is legal, isolated, which means no wormholes.
(4) Multiple degrees of Isolation
(5) Predicte locks, Granularity locks, optimistic locking and snapshot isolation

### ④ Durability

(1) Ensure logging and recovery by algorithms like Aries.
(2) Must write all log records to the disk for a transaction before commit.
(3) Atomic disk writes such as duplex write guarantees at least one block has consistent data.

## 3. Types of transaction models – why they are interesting

### ① Flat Transaction

E.g. airline booking

```
        BEGIN WORK
              S1: book flight from Melbourne to Singapore
              S2: book flight from Singapore to London
              S3: book flight from London to Dublin
        END WORK
```

Problem: from Dublin if we cannot reach our final destination instead we wish to fly to Paris from Singapore and then reach our final destination.

If we roll back we need to re do the booking from Melbourne to Singapore which is a waste.

Limitation: Any failure of long running transaction requires lot of unnecessary computation.
改进: Flat Transaction with save points

### ② Nested Transaction

(1) Commit rule
• A subtransaction can either commit or abort, however, commit cannot take place unless the parent itself commits.
• Subtransactions have A, C, and I properties but not have D property unless all its ancestors commit.
• Commit of a sub transaction makes its results available only to its parents.

(2) Roll back Rules
If a subtransaction rolls back all its children are forced to roll back.
(3) Visibility Rules
Changes made by a sub transaction are visible to the parent only when the sub transaction commits. Whereas all objects of parent are visible to its children. Implication of this is that the parent should not modify objects while children are accessing them. This is not a problem as parent is not run in parallel with its children.

## 4. Isolation concepts, degrees of isolation and their usefulness

### ① Violation of Isolation

### ② Well-formed Transaction

### ③ Two Phase Transaction

### ④ History

一组Transactions的所有操作序列，在保持各自顺序的情况下归并成的一个单一序列 P19

### ⑤ Theorems

### ⑥ Degrees

| | Degree 0 | Degree 1 | Degree 2 | Degree 3 |
|---|---|---|---|---|
| Two phase | No | write | write | Yes |
| Well formed | write | write | Yes | Yes |
| Repeatable reads | No | No | No | Yes |
| Conflicts | None | write -> write | write -> write<br>write -> read | write -> write<br>write -> read<br>read -> write |
| Examples | Google | Library | Ticket Sale System | Bank, Stock |

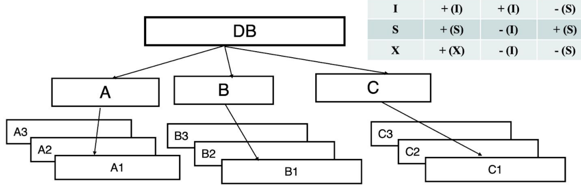
5. Lock compatibility issues, including granular locks. Do not need to study implementation details locks – no code need to be memorised but you should be able to have a clear idea of who to do develop such codes when required – e.g., google job interviews). You are asked the use Fault tolerance and reliability (expected to know the basic mathematics to solve simple problems – no need of a calculator to answer the questions in the exam). You will be given disk storage configuration and you are expected to compute say MTTF.

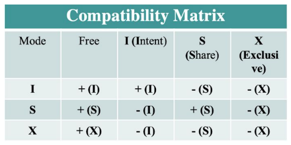
## ① Lock Compatibility

| | None | IS | IX | S | SIX | U | X |
|---|---|---|---|---|---|---|---|
| IS | + | + | + | + | + | | |
| IX | + | + | + | | | | |
| S | + | + | | + | | | |
| SIX | + | + | | | | | |
| U | + | + | | + | | | |
| X | + | | | | | | |

SIX: a coarse granularity shared lock with an Intent to set finer granularity exclusive locks

Granular Locks





Compatibility Matrix

| Mode | Free | I (Intent) | S (Share) | X (Exclusive) |
|---|---|---|---|---|
| I | + (I) | + (I) | - (S) | - (X) |
| S | + (S) | - (I) | + (S) | - (X) |
| X | + (X) | - (I) | - (S) | - (X) |

| T1 | T2 | T3 | T4 | T5 |
|---|---|---|---|---|
| Lock(I, DB) | Lock(S, DB) | Lock(I, DB) | Lock(I, DB) | Lock(X, DB) |
| Lock(I, A) | Read (A3) | Lock(X, B) | Lock(S, B) | Read (A3) |
| Lock(S, A3) | Read (B2) | Read (B2) | Read (B2) | Write (A1) |
| Read (A3) | Read (C1) | write (B3) | Read (B3) | Write (B2) |
| Unlock(all) | Unlock(all) | Unlock(all) | Unlock(all) | Read (C1) |
| | | | | Unlock(all) |

T1 and T2 could access the resources but compatibility matrix does not allow this. T1 and T3 have no conflicts and can run in parallel. T3 and T4 has conflict and will be run sequentially either T3;T4 or T4;T3. T5 has conflict with other Transactions.

## ② Fault Tolerance

(1) Failvote - use two or more modules and compare their outputs. Stops if there are no majority outputs agreeing. It fails twice as often with duplication but gives clean failure semantics 有n个module会在n/2 fail后stop，需要超过半数的投票通过，10个中有6个存活且全部通过

(2) Failfast (voting)- this scheme is similar to fail voting except the system senses which modules are available and then uses the majority of the available modules. 有n个module会在只剩1个时stop，只需要存活的投票，10个中挂了6个，余下4个只要3个通过就行
E.g. Consider a system with modules each with MTTF of 10 years including soft faults. 每个module的MTTF是10年，采用failvote
2个module时: MTTF = 10/2 = 5 years
3个module时: 10/3 for the first failure + 10/2 for the 2nd failure = 8.3 years

### ③ N-plex repair

In this configuration the faulty equipment is repaired with an average time of MTTR (mean time to repair) as soon as a fault is detected.

MTTF of N-plex system = $\left(\frac{MTTF}{n}\right)\left(\frac{MTTF}{MTTR}\right)^{(n-1)}$

## 6. You will be asked question on granular locks. You need to know when such a concept is applicable.

Pick a fixed set of predicates (read and write sets)
They form a tree
Lock the nodes in this tree

## 7. Deadlocks (make sure you know how to compute probability of a deadlock occurring). You will be given a situation of an application say with the number of transactions running, average number of records a transaction accesses and the size of the database and you are asked to estimate probability of deadlock.

略

## 8. Optimistic Locking and its relationship to Snapshot Isolation. You need to know why they are useful and when. You also need to know when they fail and should be able to demonstrate say by an example.

### ① Optimistic locking: 乐观锁

• Read the current quantity into the application by Slocks briefly, then release the Slocks
• Process the values read and determine new values to be written if any.
• At commit time take appropriate locks (Slocks on items to be tested and Xlocks on tuples to be modified) check the values read are not modified.
• If modified abort the transaction (release all locks) and restart the transaction
• If not make changes.
对使用对象马上上锁称为悲观，仅在提交时上锁称为乐观
**Why useful and when**:

Generally used in environments with **low data contention**. Transactions can complete without managing locks and without having transactions wait for other transactions. **Higher throughput** than other concurrency control methods.
**When they fail**:
In reality works **worse** than locking scheme if there are **many hotspots**. 如果有contention会导致大量的重复计算
**Example**:

```
Read A into A1
Read B into B1
Read C into C1
Loop: Compute new values based on A1 and B1
    Slock A, Read A into A2
    Slock B, Read B into B2
    Xlock C, Read C into C2
    if (A1 == A2 & B1 == B2 & C1 == C2)
        Write new value into C
        Commit
        Unlock A, B and C
    else
        A1 = A2
        B1 = B2
        C1 = C2
        Unlock A, B and C
        goto Loop
    end
```

## ② Snapshot Isolation: 一种隔离级别

is a guarantee that all reads made in a transaction will see a consistent snapshot of the database, and the transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot.
**Relationship with optimistic locking**:
Snapshot Isolation mode provides another mechanism for implementing optimistic locking with less recomputations. 在commit时不再检查A和B有没有被别的T给update，只保证此次要modify的C没有被update过就行了（预防lost update）
**Why useful and when**:
Its transaction throughput is very high compared to two phase locking scheme.
**When they fail**:
It will not guarantee **Serializability**.
**Example**:

```
Read C into C1
Read D into D1
Loop:
    Read A into A1
    Read B into B1
    Compute new values based on A1 and B1
    Let new value for C is C3 and for D is D3
    Xlock C
    Xlock D
```

```
Read C into C2
Read D into D2
if (C1 = = C2 & D1 = = D2)
    write C3 to C
    write D3 to D
    Commit
    Unlock(C and D)
else
    C1 = C2
    D1 = D2
    Unlock(C and D)
    goto Loop
end
```

**Fail Example**: Integrity constraint A+B >= 0; A = 100; B = 100;

T1:

```
Loop:
    Read A to A1;
    Read B to B1;
    A3 = A1 - 200;
    Lock(X, A)
    Read A to A2
    if (A1 != A2)
        Unlock(A)
        goto Loop
    elseif (A3+ B1 >= 0)
        Write A3 to A
        Commit
    else abort
Unlock (all locks)
```

T2:

```
Loop:
    Read A to A1;
    Read B to B1;
    B3 = B1 - 200;
    Lock(X, B)
    Read B to B2
    if (B1 != B2)
        Unlock(B)
        goto Loop
    elseif (A1+ B3 >= 0)
        Write B3 to B
        Commit
    else abort
Unlock (all locks)
```

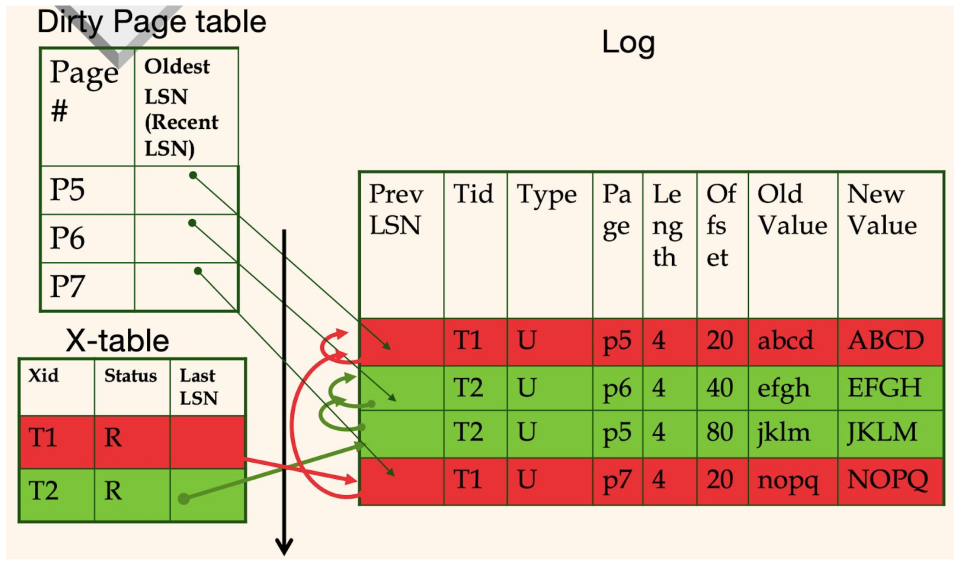
One or both transactions can commit but when both are committed, it is **not serializable** as only one should be able to commit. 两个都commit不满足A+B >= 0

## 9. Database Recovery -- Aries (you need to fully understand this). You need 100% of understanding of how it works. I will give you a crash scenario and you need to show how Aries restores the database system to its normal state. You need to know how to construct Dirty page table and Transaction table as recovery progresses. You should practice this a lot before the examination.

Recovery manager guarantees **atomicity** and **durability**.
**Force**: Write to disk at commit (Poor response time, provides durability)
**No Force**: Leaves pages as long as possible in buffer caches (Faster response, higher efficiency)
**Steal**: No force, allowing writing pages to disk even if they are uncommitted.
**The Write-Ahead Logging Protocol**:
1. 被steal前force写log (both old and new) (保证了Atomicity，如果abort依据log可undo)
2. commit前force写log (保证了Durablity，如果crash了已经commit依据log可redo)

**WAL & the Log**:
1. 每一条Log record都有一个独特的Log Sequence Number (LSN) 始终增加
2. 每一页data page有一个pageLSN（表示最近的一条有关此page的log）
3. 系统track flushedLSN（表示那些logs已经flushed to disk）

4. 在每一个page is written to disk之前要确保pageLSN <= flushedLSN（即page相关logs存过）
**Transaction Table:** 每个entry对应一个active transaction，存T的<u>status</u> (running/committed/aborted) 以及<u>lastLSN</u>（该T的对应的最后一条log）
**Dirty Page Table**: 每个entry对应一个dirty page in the buffer pool，存recLSN（即这个page被从disk中load出来以后第一个改动它的log）


Dirty Page table

| Page # | Oldest LSN (Recent LSN) |
|---|---|
| P5 | |
| P6 | |
| P7 | |

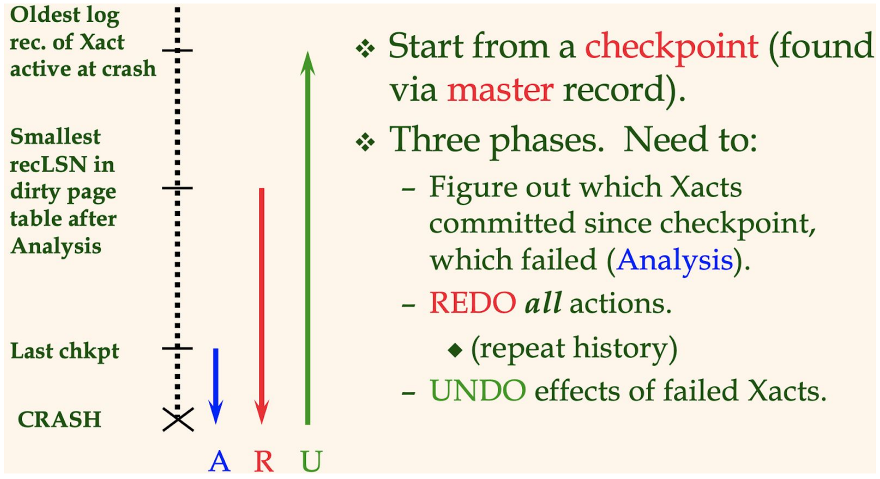
X-table

| Xid | Status | Last LSN |
|---|---|---|
| T1 | R | |
| T2 | R | |

Log

| Prev LSN | Tid | Type | Page | Length | Offset | Old Value | New Value |
|---|---|---|---|---|---|---|---|
| | T1 | U | p5 | 4 | 20 | abcd | ABCD |
| | T2 | U | p6 | 4 | 40 | efgh | EFGH |
| | T2 | U | p5 | 4 | 80 | jklm | JKLM |
| | T1 | U | p7 | 4 | 20 | nopq | NOPQ |

LSN的实际值在图中省略了，PrevLSN是属于同一T的上一条log，形成链表，指向自己即结束
Type: Update, Commit, Abort, End (signifies end of commit or abort), Compensation Log Records (CLRs) (for UNDO actions)
**Checkpointing**: Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover when crash. Write to log:
1. Begin checkpoint record: Indicates when chkpt began.
2. End checkpoint record: Contains current <u>Xact table</u> and <u>dirty page table</u>.
如我在备份的时候有别的加入table忽略不管，fuzzy checkpoint
Store LSN of chkpt record in a safe place (master record).
**Transaction Abort**:
1. 在restore old value前，write a CLR which has one extra field “undonextLSN” points to the next LSN to undo. CLRs never Undone (but they might be Redone when repeating history)
2. At end of UNDO, write an “end” log record.
**Transaction Commit**:
1. Write commit record to log.
2. All log records up to Xact’s lastLSN are flushed. (Guarantees that flushedLSN >= lastLSN)
3. Commit() returns.
4. Write end record to log.
**Crash Recovery**:


Oldest log rec. of Xact active at crash
Smallest recLSN in dirty page table after Analysis
Last chkpt
CRASH
A
R
U

- Start from a checkpoint (found via master record).
- Three phases. Need to:
  - Figure out which Xacts committed since checkpoint, which failed (Analysis).
  - REDO *all* actions.
    - (repeat history)
  - UNDO effects of failed Xacts.

**The Analysis Phase**:

1. Reconstruct state at checkpoint. (via end_checkpoint record)
2. Scan log forward from checkpoint.

    – End record: Remove Xact from Xact table. 已经perfectly done的T不管

    – Other records:

    Add Xact to Xact table, set lastLSN=LSN, change Xact status on commit.

    – Update record: If P not in Dirty Page Table,

    Add P to Dirty Page Table, set its recLSN=LSN.

**The REDO Phase**:

1. We repeat History to reconstruct state at crash:

    – Reapply all updates (even of aborted Xacts!), redo CLRs.

2. Scan forward from log rec containing **<u>smallest recLSN in D.P.T.</u>** For each CLR or update log rec LSN, REDO the action unless:

    – Affected page is not in the Dirty Page Table, or（不在DPT里说明already written）

    – Affected page is in D.P.T., but has recLSN > LSN, or（这个change也已经recorded）

    – pageLSN (in DB) >= LSN.（DB里已经比现在的新了）

    以上操作都是在RAM里做的会非常快

3. To REDO an action:

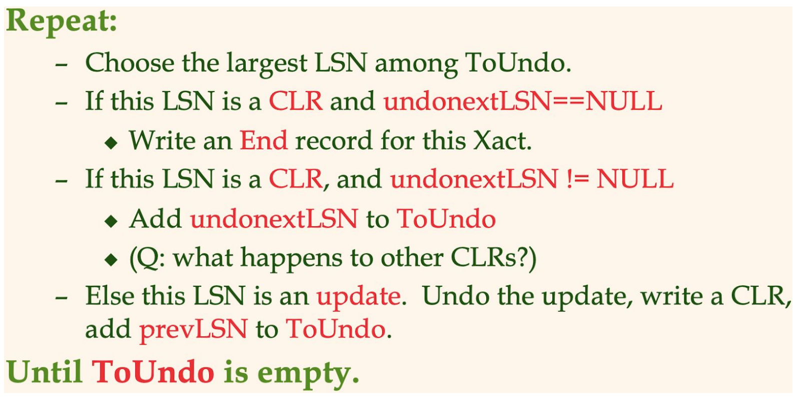
    – Reapply logged action. (Note: This happens in the buffer pool.)

    – Set pageLSN to LSN. No additional logging!

**The UNDO Phase**:

ToUndo = { l | l is a lastLSN of a “loser” Xact}

We can form this list from Xtable.



**Repeat:**

- Choose the largest LSN among ToUndo.
- If this LSN is a CLR and undonextLSN==NULL
  - Write an End record for this Xact.
- If this LSN is a CLR, and undonextLSN != NULL
  - Add undonextLSN to ToUndo
  - (Q: what happens to other CLRs?)
- Else this LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo.

**Until ToUndo is empty.**

从最大的LSN开始把没有end的loser T都给从下往上Undo掉

**Additional Crash Issues**:

如果在analysis期间crash，没有办法，只能做checkpoint more frequently

如果在REDO期间crash，只能重新做，解决方法是periodically flash buffer pages

REDO和analysis期间都不写log

如何减少UNDO? Avoid long-running Xacts.

## 10. Distributed committed protocols for Database recovery. This essential part of understanding CAP theorem.

**Two phase commit**: 比如一个T包含3个sub-transactions在不同城市run，其中一个是master

Phase 1: The coordinator sends **a prepare message** to each subordinate, 收到信息后subordinate决定是commit还是abort. It force-writes an abort or prepare log record and sends yes or no message to the coordinator.

Phase 2: 如果coordinator从所有的subordinate都收到了yes, it force-writes a commit log record and then sends commit messages to all subordinates. 如果收到了一个no或是一定时间内无响应it force-writes an abort log and sends abort message to all subordinates.

When a subordinate receives an abort message it force-writes an abort log record and sends an **ack** to the coordinator. It aborts the subtransaction.