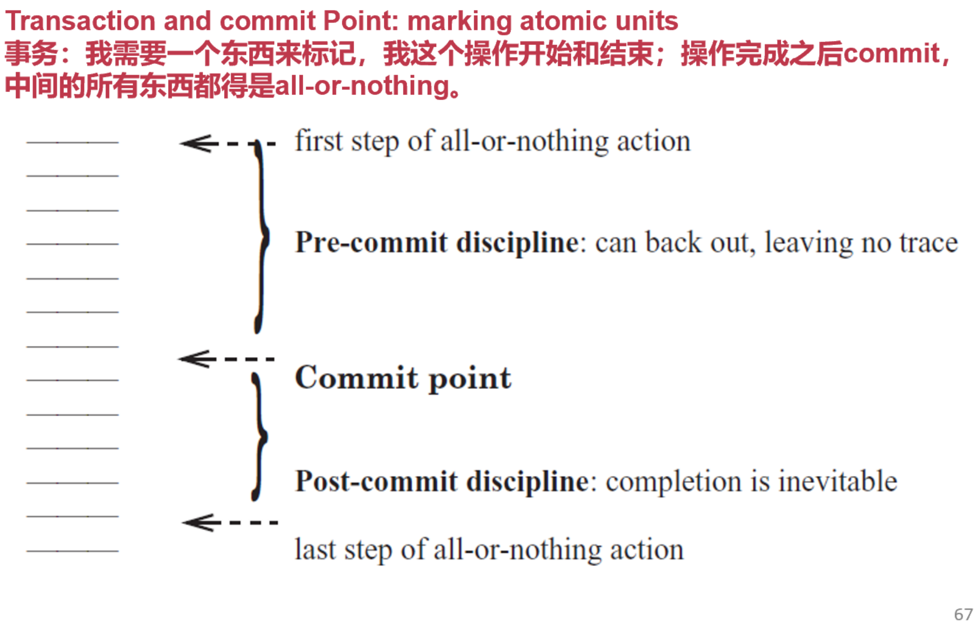
# 【计算机系统工程】事务1: Atomicity+Durability

@credits Xingda Wei, IPADS

## 1. What all-or-nothing atomicity



Transaction and commit Point: marking atomic units
事务：我需要一个东西来标记，我这个操作开始和结束；操作完成之后commit，中间的所有东西都得是all-or-nothing。

first step of all-or-nothing action

**Pre-commit discipline**: can back out, leaving no trace

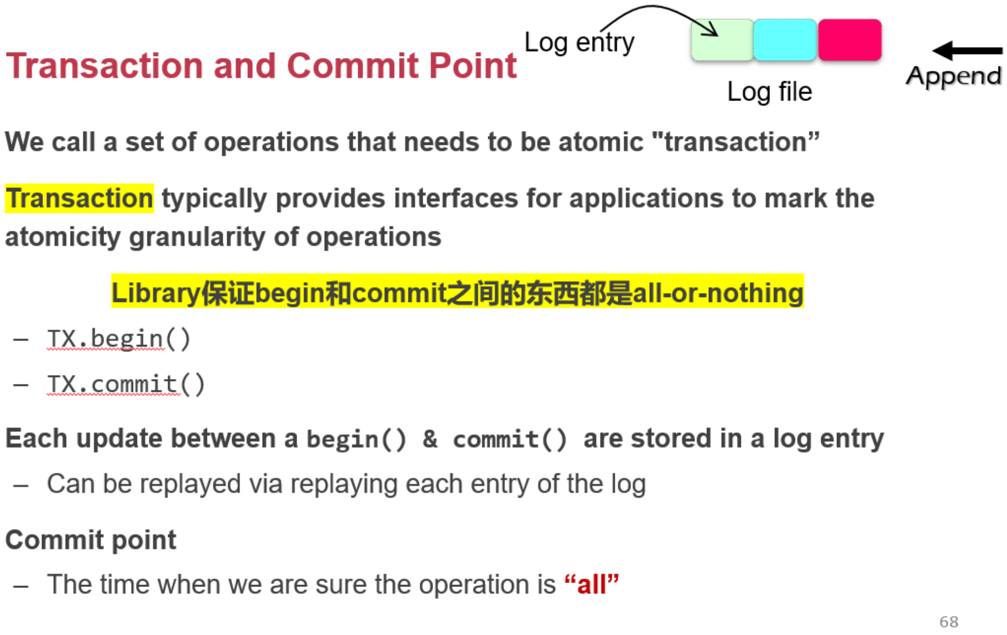
**Commit point**

**Post-commit discipline**: completion is inevitable

last step of all-or-nothing action

67

说穿了就是一个系统库，提供了begin和commit的调用，库保证了中间所有操作都是all or nothing的状态。



Log entry
Log file
Append

**Transaction and Commit Point**

**We call a set of operations that needs to be atomic "transaction"**

**Transaction typically provides interfaces for applications to mark the atomicity granularity of operations**

**Library保证begin和commit之间的东西都是all-or-nothing**

- `TX.begin()`
- `TX.commit()`

**Each update between a `begin()` & `commit()` are stored in a log entry**

- Can be replayed via replaying each entry of the log

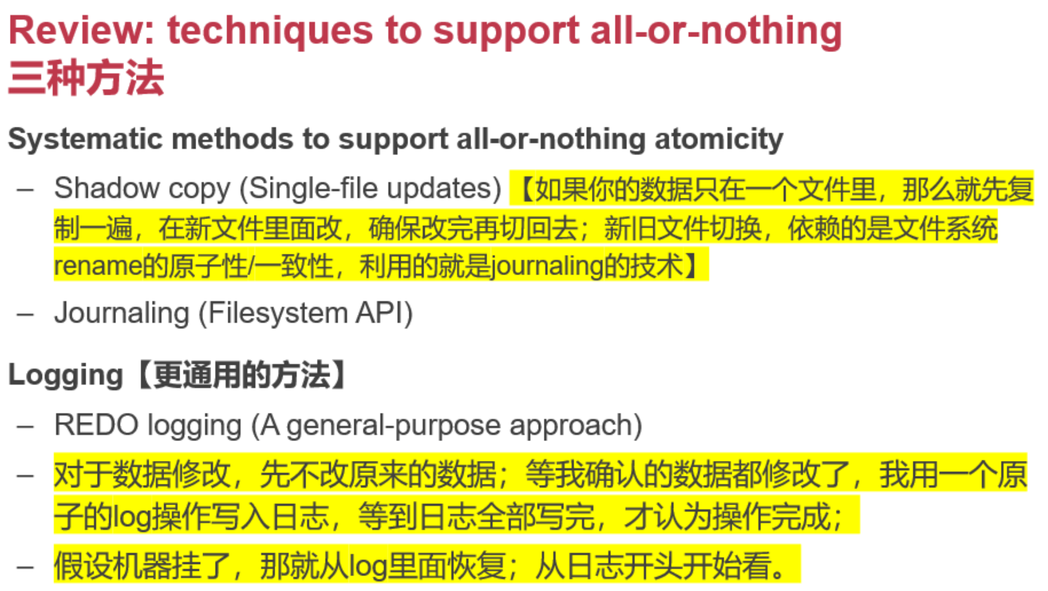
**Commit point**

- The time when we are sure the operation is **"all"**

68

# 2. How atomicity: Redo logging

## 1. What redo logging



### Review: techniques to support all-or-nothing 三种方法

**Systematic methods to support all-or-nothing atomicity**

– Shadow copy (Single-file updates) 【如果你的数据只在一个文件里，那么就先复制一遍，在新文件里面改，确保改完再切回去；新旧文件切换，依赖的是文件系统rename的原子性/一致性，利用的就是journaling的技术】

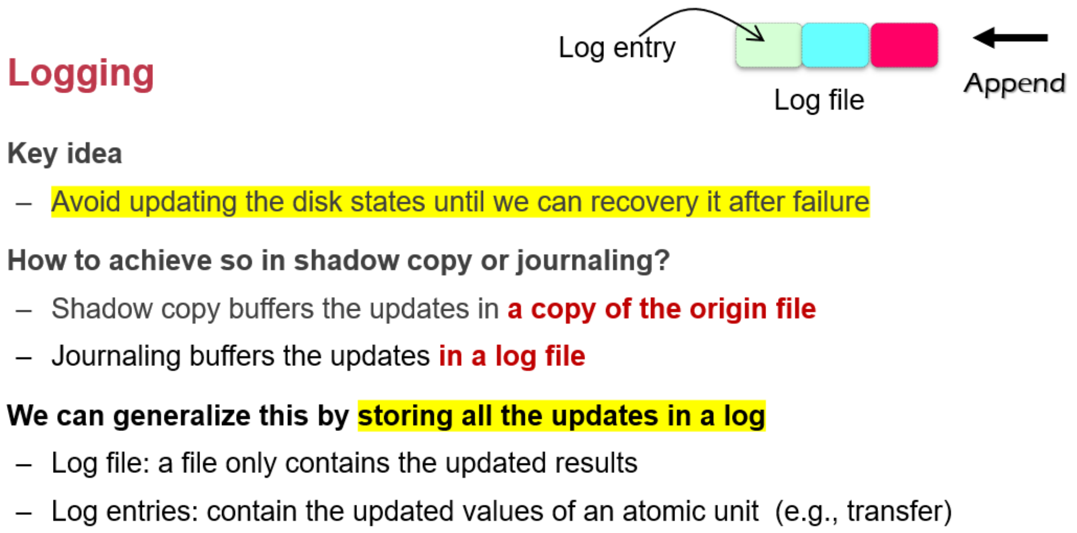
– Journaling (Filesystem API)

**Logging【更通用的方法】**

– REDO logging (A general-purpose approach)

– 对于数据修改，先不改原来的数据；等我确认的数据都修改了，我用一个原子的log操作写入日志，等到日志全部写完，才认为操作完成；

– 假设机器挂了，那就从log里面恢复；从日志开头开始看。



### Logging

**Key idea**

– Avoid updating the disk states until we can recovery it after failure

**How to achieve so in shadow copy or journaling?**

– Shadow copy buffers the updates in **a copy of the origin file**

– Journaling buffers the updates **in a log file**

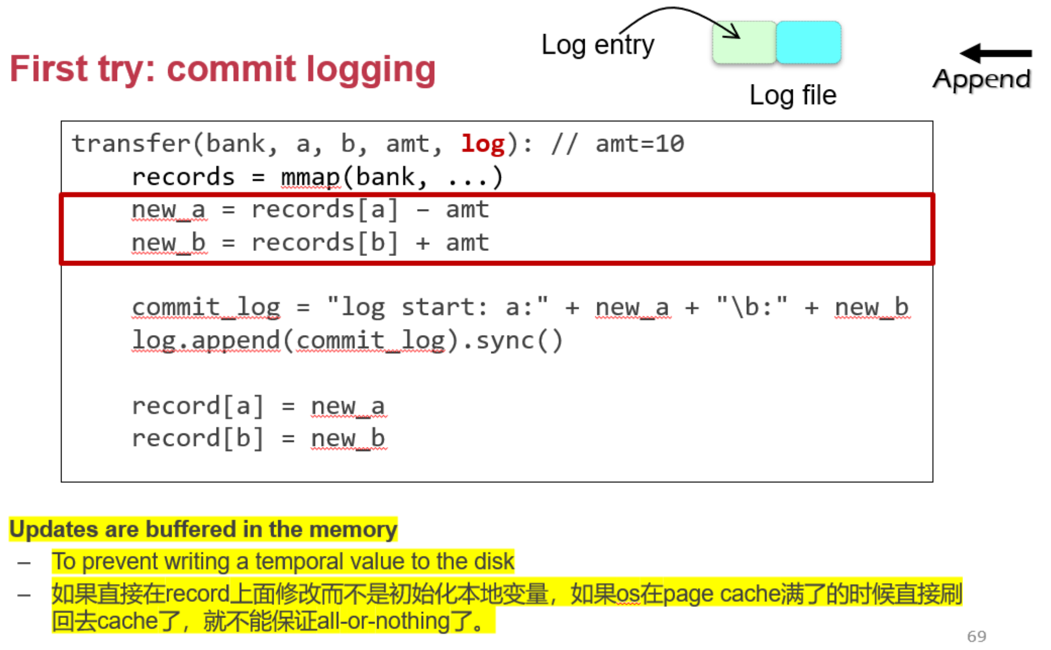
**We can generalize this by storing all the updates in a log**

– Log file: a file only contains the updated results

– Log entries: contain the updated values of an atomic unit  (e.g., transfer)

## 2. How redo logging

### 1. Buffer the update in the memory



**First try: commit logging**

Log entry

Log file

Append

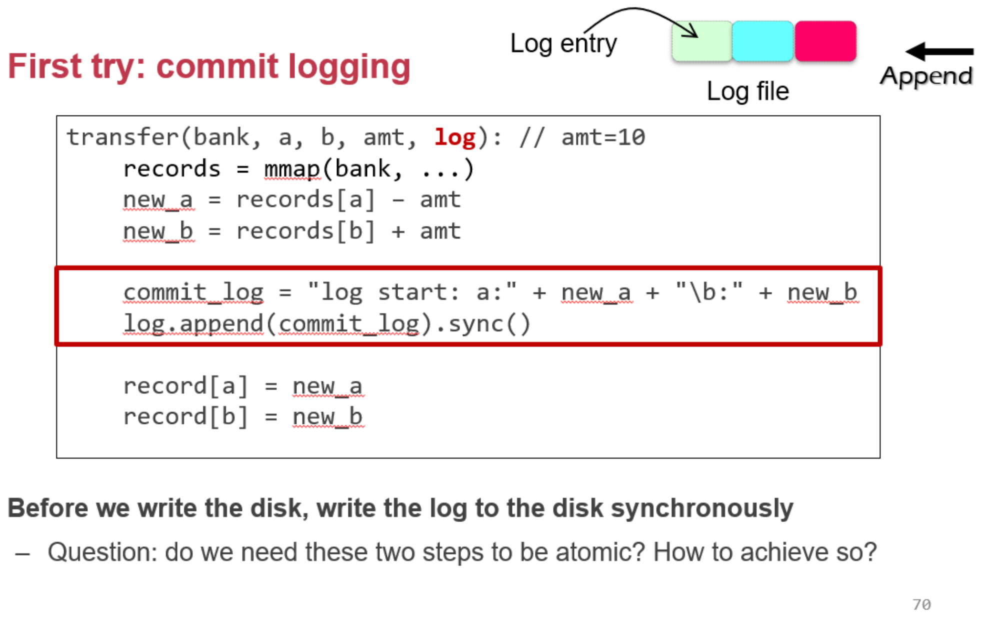
```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    new_a = records[a] - amt
    new_b = records[b] + amt

    commit_log = "log start: a:" + new_a + "\b:" + new_b
    log.append(commit_log).sync()

    record[a] = new_a
    record[b] = new_b
```

**Updates are buffered in the memory**

- To prevent writing a temporal value to the disk
- 如果直接在record上面修改而不是初始化本地变量，如果os在page cache满了的时候直接刷回去cache了，就不能保证all-or-nothing了。

69

## 2. Write log



**First try: commit logging**

Log entry

Log file

Append

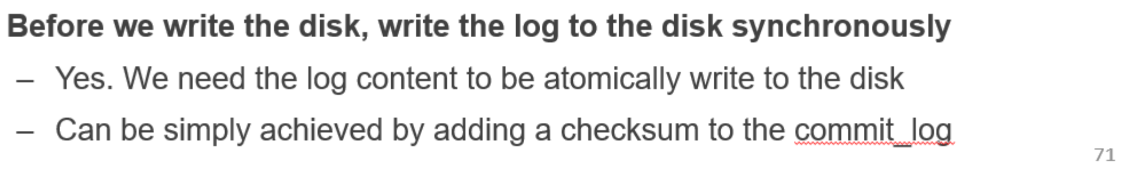
```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    new_a = records[a] - amt
    new_b = records[b] + amt

    commit_log = "log start: a:" + new_a + "\b:" + new_b
    log.append(commit_log).sync()

    record[a] = new_a
    record[b] = new_b
```

**Before we write the disk, write the log to the disk synchronously**

- Question: do we need these two steps to be atomic? How to achieve so?

70

我们要保证红色框里面写log的两句话是原子的。

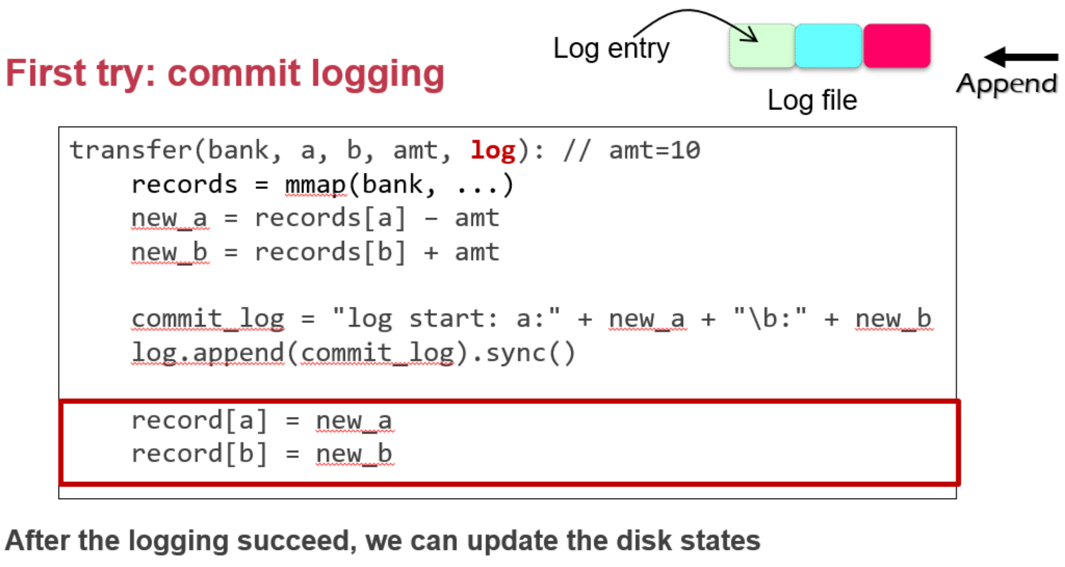


**Before we write the disk, write the log to the disk synchronously**

- Yes. We need the log content to be atomically write to the disk
- Can be simply achieved by adding a checksum to the commit_log

71

## 3. Update the disk

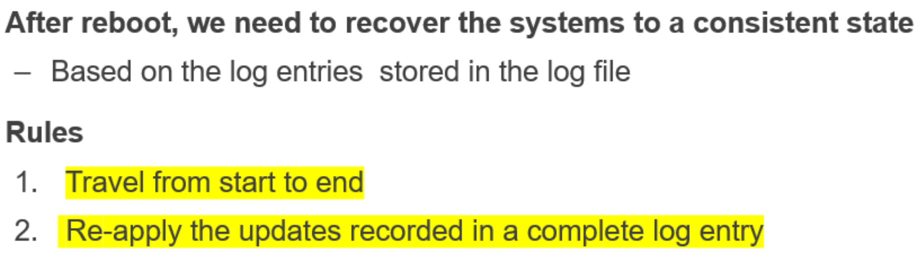


**First try: commit logging**

```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    new_a = records[a] - amt
    new_b = records[b] + amt

    commit_log = "log start: a:" + new_a + "\b:" + new_b
    log.append(commit_log).sync()

    record[a] = new_a
    record[b] = new_b
```

After the logging succeed, we can update the disk states

## 3. Recover from redo logging



**After reboot, we need to recover the systems to a consistent state**

– Based on the log entries stored in the log file

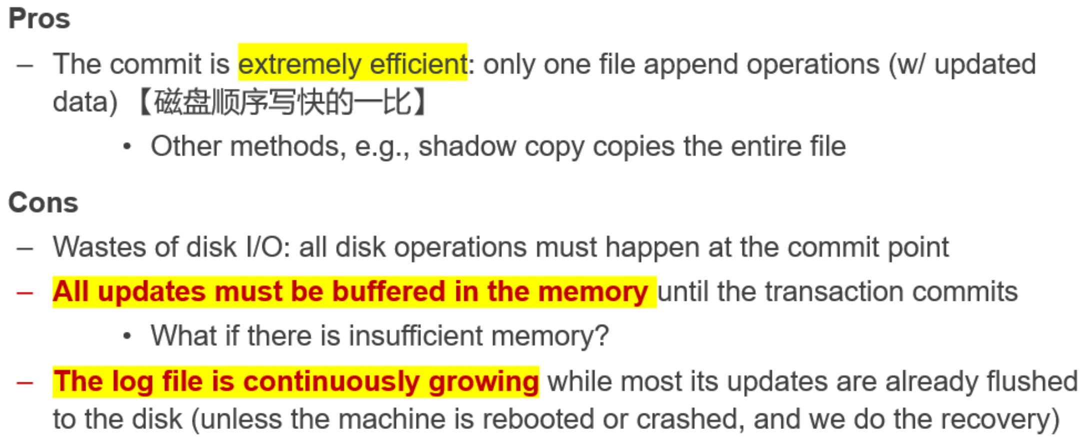
**Rules**

1. Travel from start to end
2. Re-apply the updates recorded in a complete log entry

## 3. Pros and cons of redoing logging

当计算机不发生crash的时候，logging就会一直涨上去，所以我们需要一个机制来规避这一点：

- 比如一个tx，它所有的东西都已经从内存刷回去了，那它的redo logging就没有必要存在了。



**Pros**

– The commit is extremely efficient: only one file append operations (w/ updated data) 【磁盘顺序写快的一比】

  • Other methods, e.g., shadow copy copies the entire file

**Cons**

– Wastes of disk I/O: all disk operations must happen at the commit point

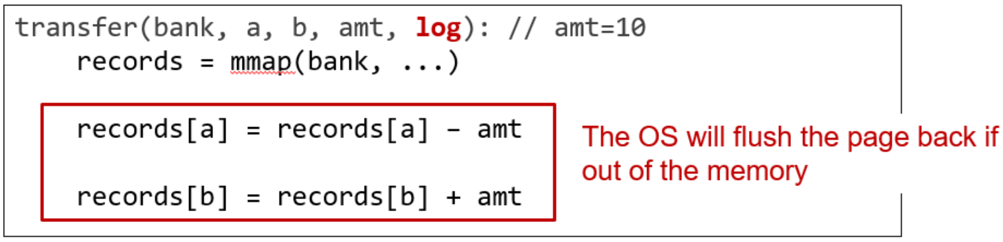
– **All updates must be buffered in the memory** until the transaction commits

  • What if there is insufficient memory?

– **The log file is continuously growing** while most its updates are already flushed to the disk (unless the machine is rebooted or crashed, and we do the recovery)

## 1. Fix1: Buffer all updates in memory?

如何避免把所有东西都塞到内存里面？我们**允许事务把uncommitted values直接写到disk上面**！

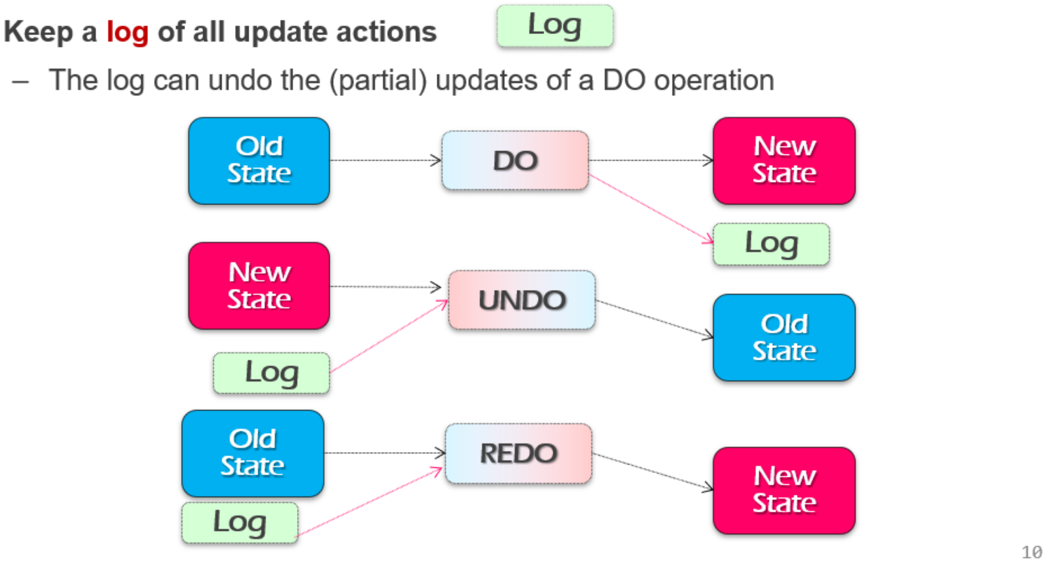


```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)

    records[a] = records[a] - amt

    records[b] = records[b] + amt
```

The OS will flush the page back if out of the memory

## 1. Undo logging: roll back uncommitted tx

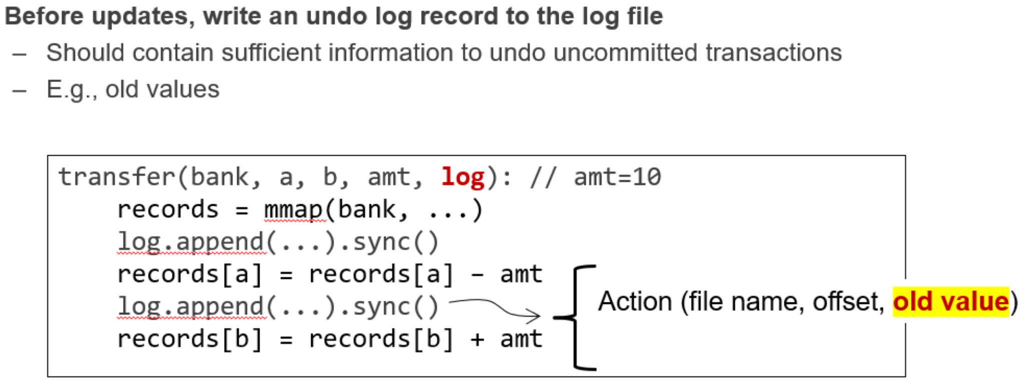
但是我们知道，OS刷回去page是随机挑选的，可能recordA所在的page刷回去了，但是recordB的还没回去呢，然后这个时候系统崩了！所以我们需要一种方法，来帮我们做之前做的操作的回滚——undo！



**Keep a log of all update actions**

– The log can undo the (partial) updates of a DO operation

10



**Before updates, write an undo log record to the log file**

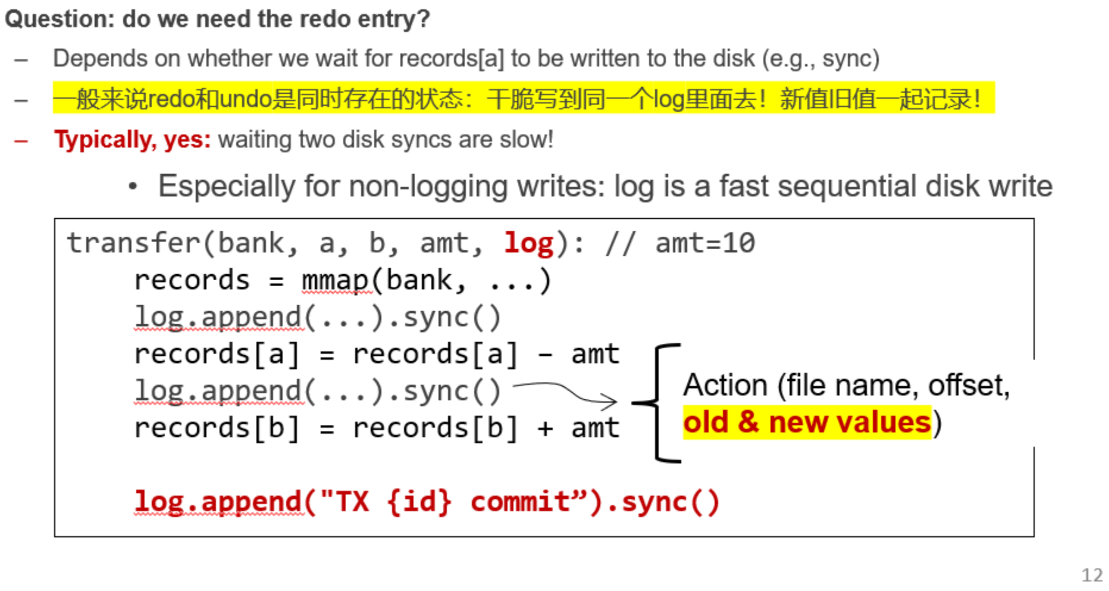
– Should contain sufficient information to undo uncommitted transactions

– E.g., old values

```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    log.append(...).sync()
    records[a] = records[a] - amt
    log.append(...).sync()
    records[b] = records[b] + amt
```

Action (file name, offset, **old value**)

## 2. Undo-redo logging



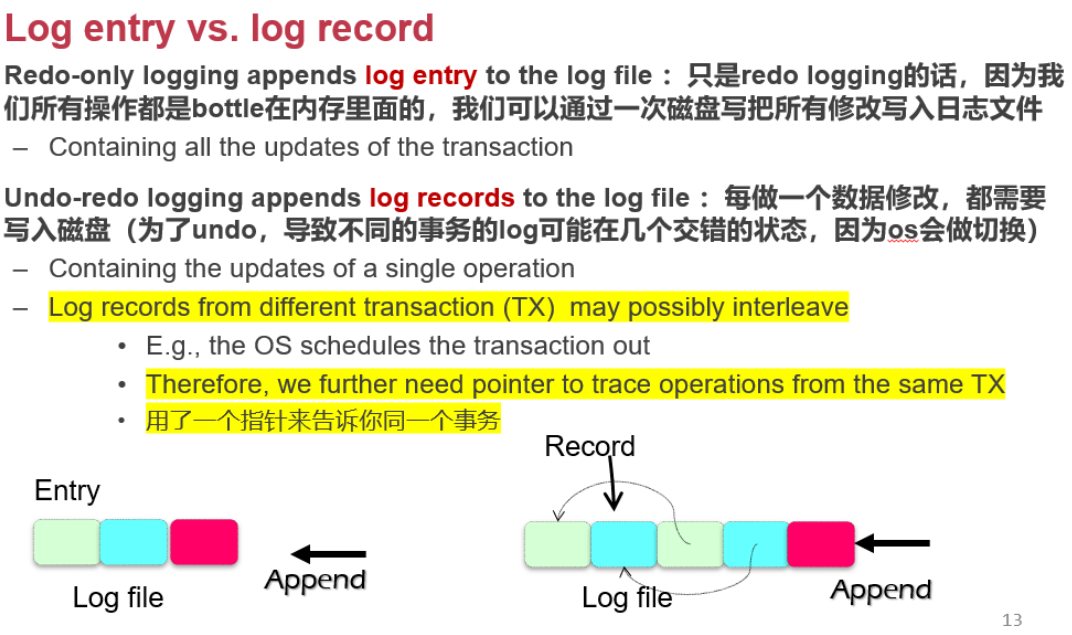
**Question: do we need the redo entry?**

- Depends on whether we wait for records[a] to be written to the disk (e.g., sync)
- 一般来说redo和undo是同时存在的状态：干脆写到同一个log里面去！新值旧值一起记录！
- **Typically, yes:** waiting two disk syncs are slow!
  - Especially for non-logging writes: log is a fast sequential disk write

```
transfer(bank, a, b, amt, log): // amt=10
    records = mmap(bank, ...)
    log.append(...).sync()
    records[a] = records[a] - amt
    log.append(...).sync()
    records[b] = records[b] + amt

    log.append("TX {id} commit").sync()
```

Action (file name, offset, **old & new values**)

12



## Log entry vs. log record

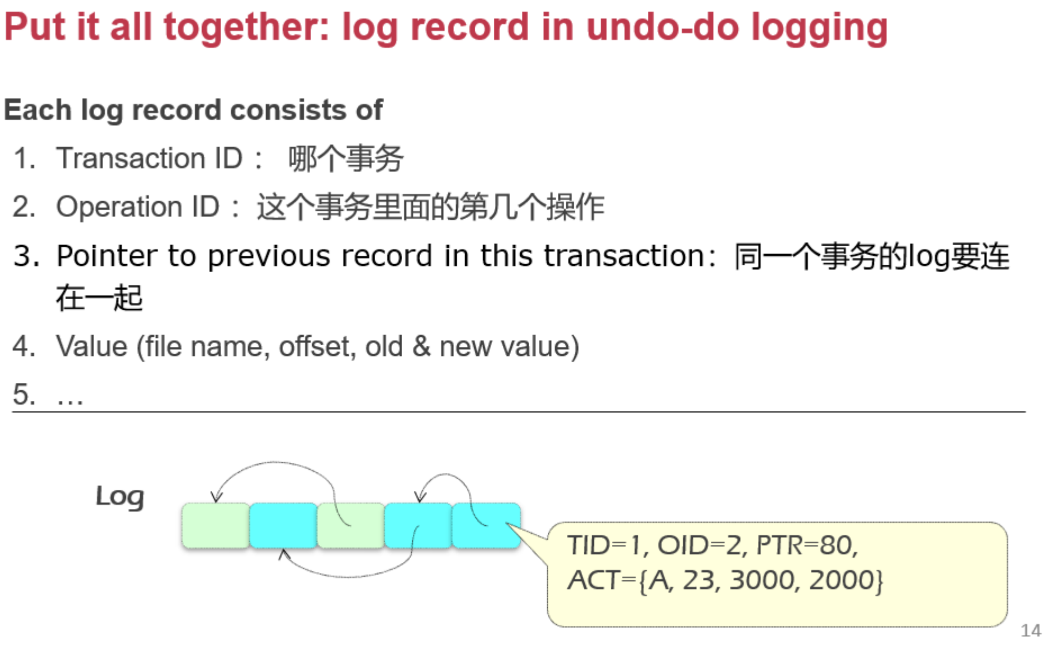
**Redo-only logging appends log entry to the log file：只是redo logging的话，因为我们所有操作都是bottle在内存里面的，我们可以通过一次磁盘写把所有修改写入日志文件**

- Containing all the updates of the transaction

**Undo-redo logging appends log records to the log file：每做一个数据修改，都需要写入磁盘（为了undo，导致不同的事务的log可能在几个交错的状态，因为os会做切换）**

- Containing the updates of a single operation
- Log records from different transaction (TX) may possibly interleave
  - E.g., the OS schedules the transaction out
  - Therefore, we further need pointer to trace operations from the same TX
  - 用了一个指针来告诉你同一个事务

13



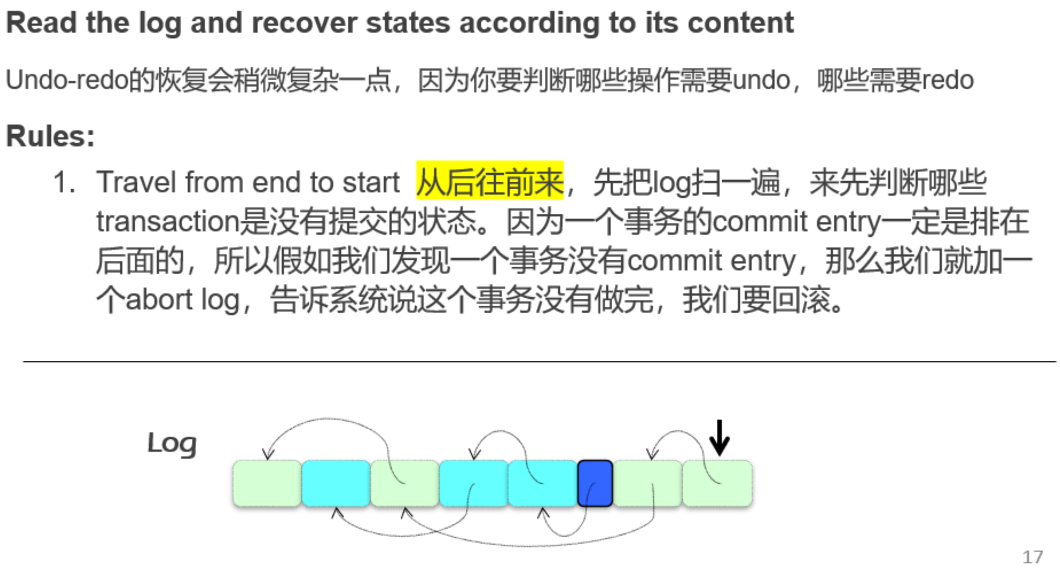
## Put it all together: log record in undo-do logging

**Each log record consists of**

1. Transaction ID： 哪个事务
2. Operation ID：这个事务里面的第几个操作
3. Pointer to previous record in this transaction：同一个事务的log要连在一起
4. Value (file name, offset, old & new value)
5. …

14

## 3. Recovery: undo-redo logging



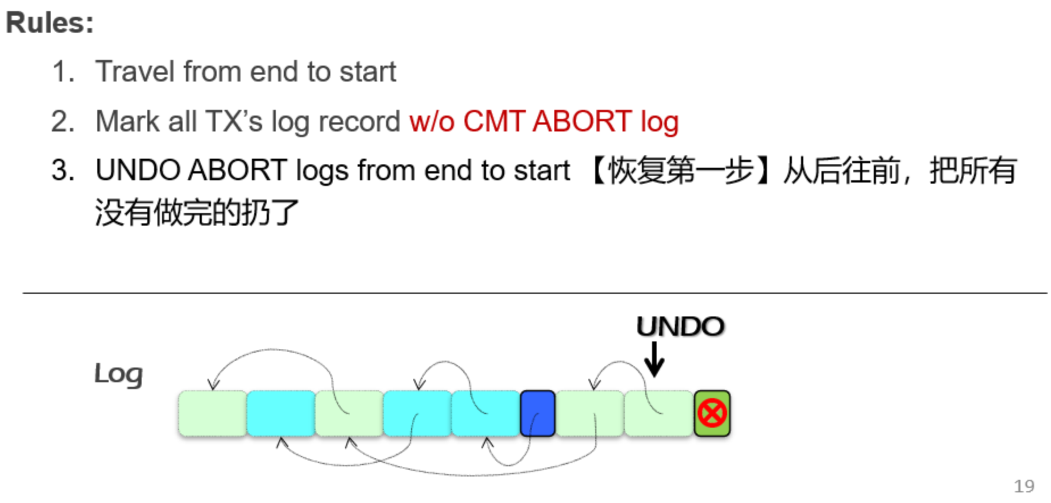
**Read the log and recover states according to its content**

Undo-redo的恢复会稍微复杂一点，因为你要判断哪些操作需要undo，哪些需要redo

**Rules:**

1. Travel from end to start 从后往前来，先把log扫一遍，来先判断哪些transaction是没有提交的状态。因为一个事务的commit entry一定是排在后面的，所以假如我们发现一个事务没有commit entry，那么我们就加一个abort log，告诉系统说这个事务没有做完，我们要回滚。

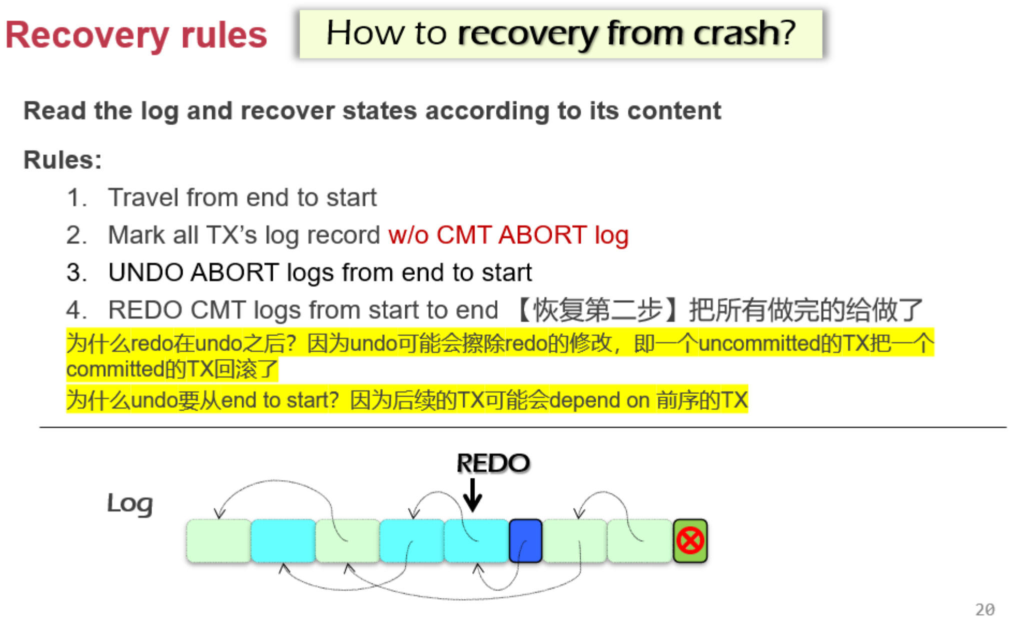
17



**Rules:**

1. Travel from end to start
2. Mark all TX’s log record w/o CMT ABORT log
3. UNDO ABORT logs from end to start 【恢复第一步】从后往前，把所有没有做完的扔了

19



**Recovery rules** How to **recovery from crash**?

**Read the log and recover states according to its content**

**Rules:**

1. Travel from end to start
2. Mark all TX’s log record w/o CMT ABORT log
3. UNDO ABORT logs from end to start
4. REDO CMT logs from start to end 【恢复第二步】把所有做完的给做了

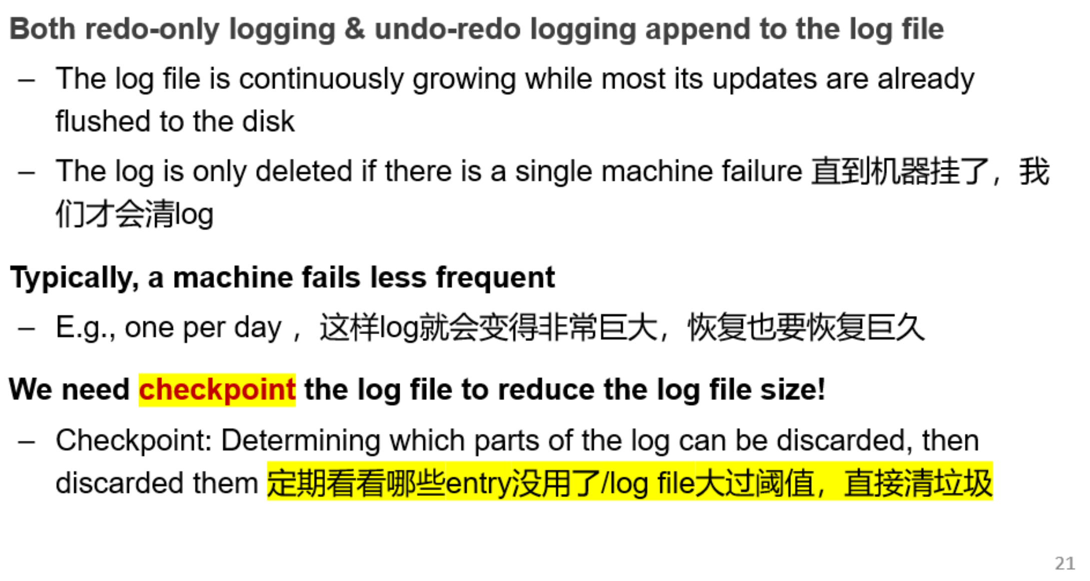
为什么redo在undo之后？因为undo可能会擦除redo的修改，即一个uncommitted的TX把一个committed的TX回滚了

为什么undo要从end to start？因为后续的TX可能会depend on 前序的TX

20

## 2. Fix2: Continuously growing logging

logging一直在上涨，除非机器崩坏了，怎么办？



**Both redo-only logging & undo-redo logging append to the log file**

- The log file is continuously growing while most its updates are already flushed to the disk
- The log is only deleted if there is a single machine failure 直到机器挂了，我们才会清log

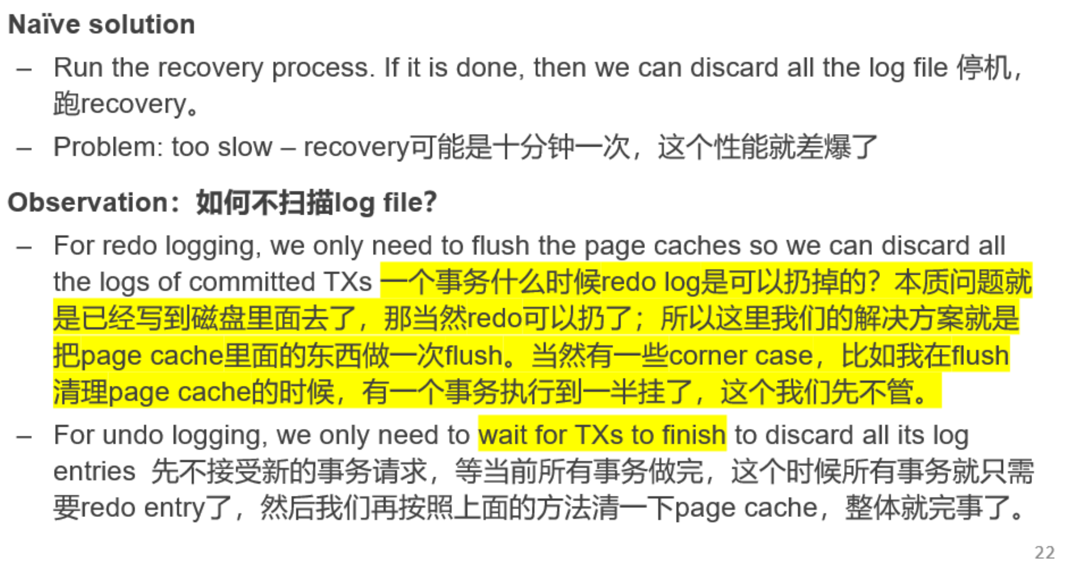
**Typically, a machine fails less frequent**

- E.g., one per day， 这样log就会变得非常巨大，恢复也要恢复巨久

**We need checkpoint the log file to reduce the log file size!**

- Checkpoint: Determining which parts of the log can be discarded, then discarded them 定期看看哪些entry没用了/log file大过阈值，直接清垃圾

## 1. Method: CKPT



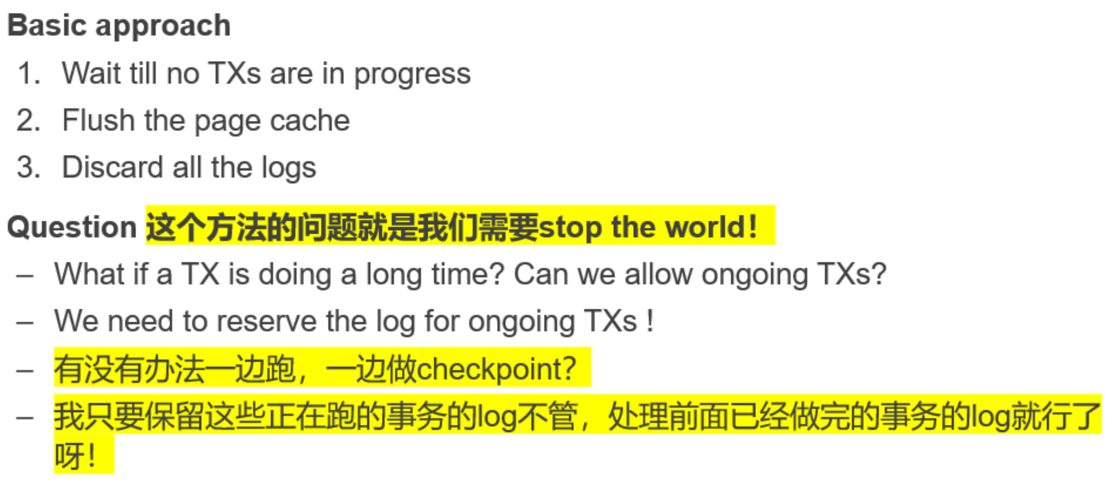
**Naïve solution**

- Run the recovery process. If it is done, then we can discard all the log file 停机，跑recovery。
- Problem: too slow – recovery可能是十分钟一次，这个性能就差爆了

**Observation：如何不扫描log file?**

- For redo logging, we only need to flush the page caches so we can discard all the logs of committed TXs 一个事务什么时候redo log是可以扔掉的？本质问题就是已经写到磁盘里面去了，那当然redo可以扔了；所以这里我们的解决方案就是把page cache里面的东西做一次flush。当然有一些corner case，比如我在flush清理page cache的时候，有一个事务执行到一半挂了，这个我们先不管。
- For undo logging, we only need to wait for TXs to finish to discard all its log entries 先不接受新的事务请求，等当前所有事务做完，这个时候所有事务就只需要redo entry了，然后我们再按照上面的方法清一下page cache，整体就完事了。

## 2. How CKPT? (for both undo && redo)



**Basic approach**

1. Wait till no TXs are in progress
2. Flush the page cache
3. Discard all the logs

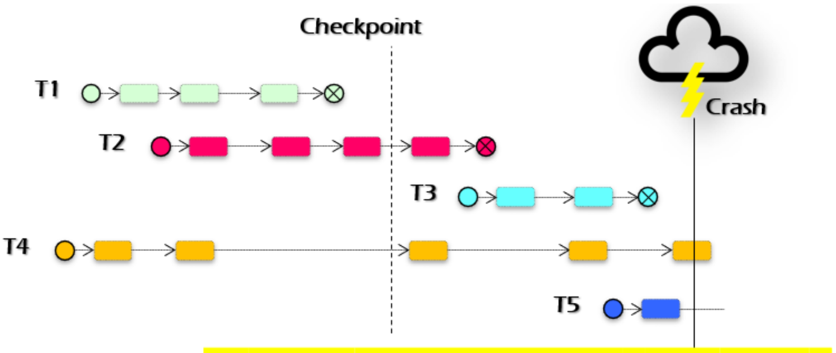
**Question 这个方法的问题就是我们需要stop the world!**

- What if a TX is doing a long time? Can we allow ongoing TXs?
- We need to reserve the log for ongoing TXs !
- 有没有办法一边跑，一边做checkpoint?
- 我只要保留这些正在跑的事务的log不管，处理前面已经做完的事务的log就行了呀！

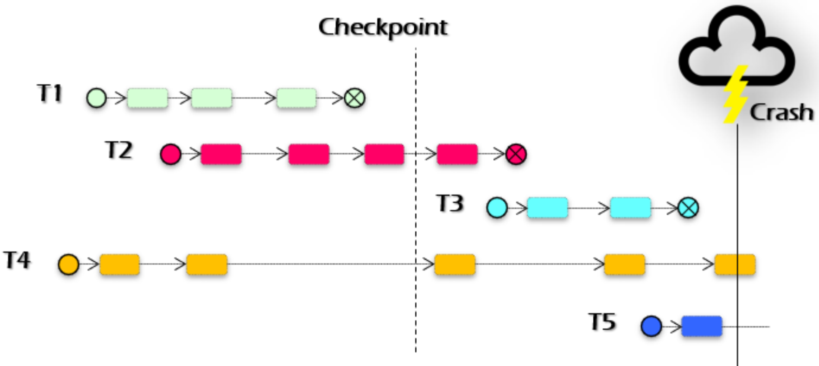
**How to checkpoint?**

1. Wait till no ~~transactions~~ actions are in progress
2. Write a **CKPT** record(check point record) to log
    - Contains a list of all transaction in process and their logs
3. Flush the page cache
4. Discard all the log records except the CKPT record

## 3. Recovery in all

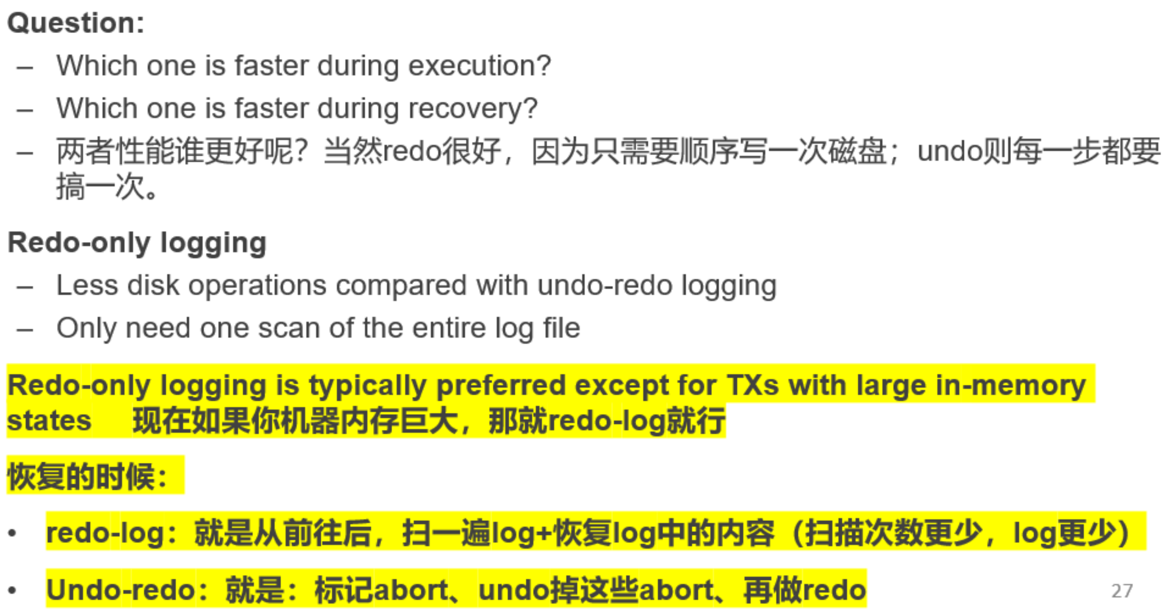


1. T1: do nothing 已经checkpoint了，状态都更新到文件里了，甚至log也都丢掉了
2. T2: redo its update based on the log in its checkpoint crash的时候有log，但是光有ckpt后的log还不够，还需要ckpt里的log；所以是undo还是redo？因为已经commit了，redo
3. T3: redo its updates based on the log entries 提交了，所以是redo
4. T4: undo its updates based on the log in both CKP & log entries 还没提交呢，所以是undo；你得去checkpoint里面把一部分log拿出来，和现有的log拼接在一起。
5. T5: undo its updates based on the log entries 没有提交，直接undo



1. Read the latest checkpoint
   → T2, T4 are ongoing transactions
2. Read log → T2, T3 are committed, T5 are ongoing
3. Undo ongoing TXs & redo committed TXs

## 4. Undo-Redo v.s. Redo



**Question:**

- Which one is faster during execution?
- Which one is faster during recovery?
- 两者性能谁更好呢？当然redo很好，因为只需要顺序写一次磁盘；undo则每一步都要搞一次。

**Redo-only logging**

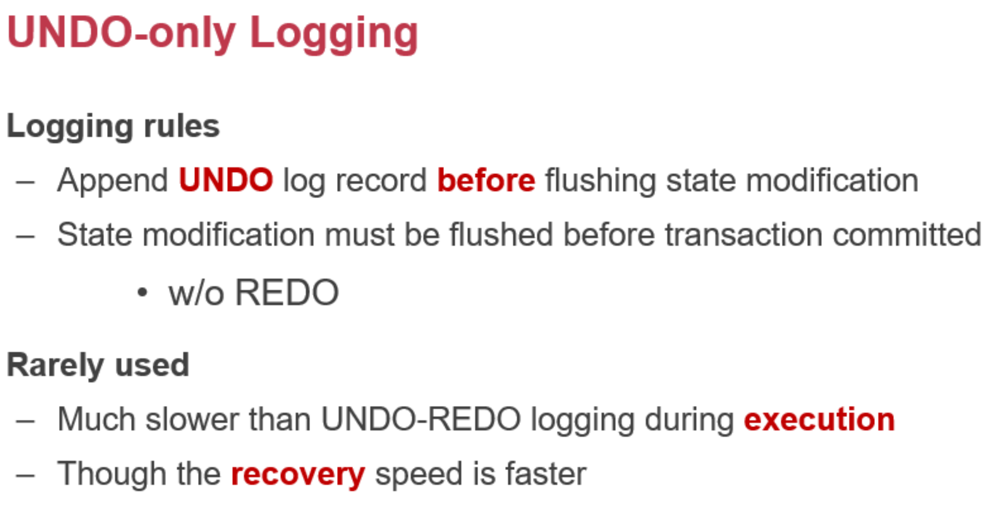
- Less disk operations compared with undo-redo logging
- Only need one scan of the entire log file

**Redo-only logging is typically preferred except for TXs with large in-memory states　现在如果你机器内存巨大，那就redo-log就行**

**恢复的时候：**

- **redo-log：就是从前往后，扫一遍log+恢复log中的内容（扫描次数更少，log更少）**
- **Undo-redo：就是：标记abort、undo掉这些abort、再做redo**

27



## UNDO-only Logging

**Logging rules**

- Append **UNDO** log record **before** flushing state modification
- State modification must be flushed before transaction committed
  - w/o REDO

**Rarely used**

- Much slower than UNDO-REDO logging during **execution**
- Though the **recovery** speed is faster

# 4. Take away messages

其实本讲涉及到了事务ACID特性的两个部分：atomicity和durability。

Logging：

- redo-logging：commit logging
  - 只使用redo-logging有两个问题：第一是所有修改都必须要buffer在内存里面，对内存是很大的压力；第二是logging会一直增长上去。
  - 为了解决第一个问题，我们允许事务将未提交的数据直接写到磁盘上，因此引入了undo-logging（比如，page是会被刷回去的，如果一个page被刷回去了，但是另一个数据所在的page没有，然后这个时候系统崩溃了，我们就需要undo整个事务）
  - 为了解决第二个问题，我们引入了ckpt
- Undo-redo logging：write ahead logging

- undo-logging：没人用，别管了

Duability：持久化。意思是commited action's data on durable storage。logging本身就是实现持久化的方式——已经写到logging里面去了呀。

这些技术在文件系统、系统、数据库里面都有类似的版本。