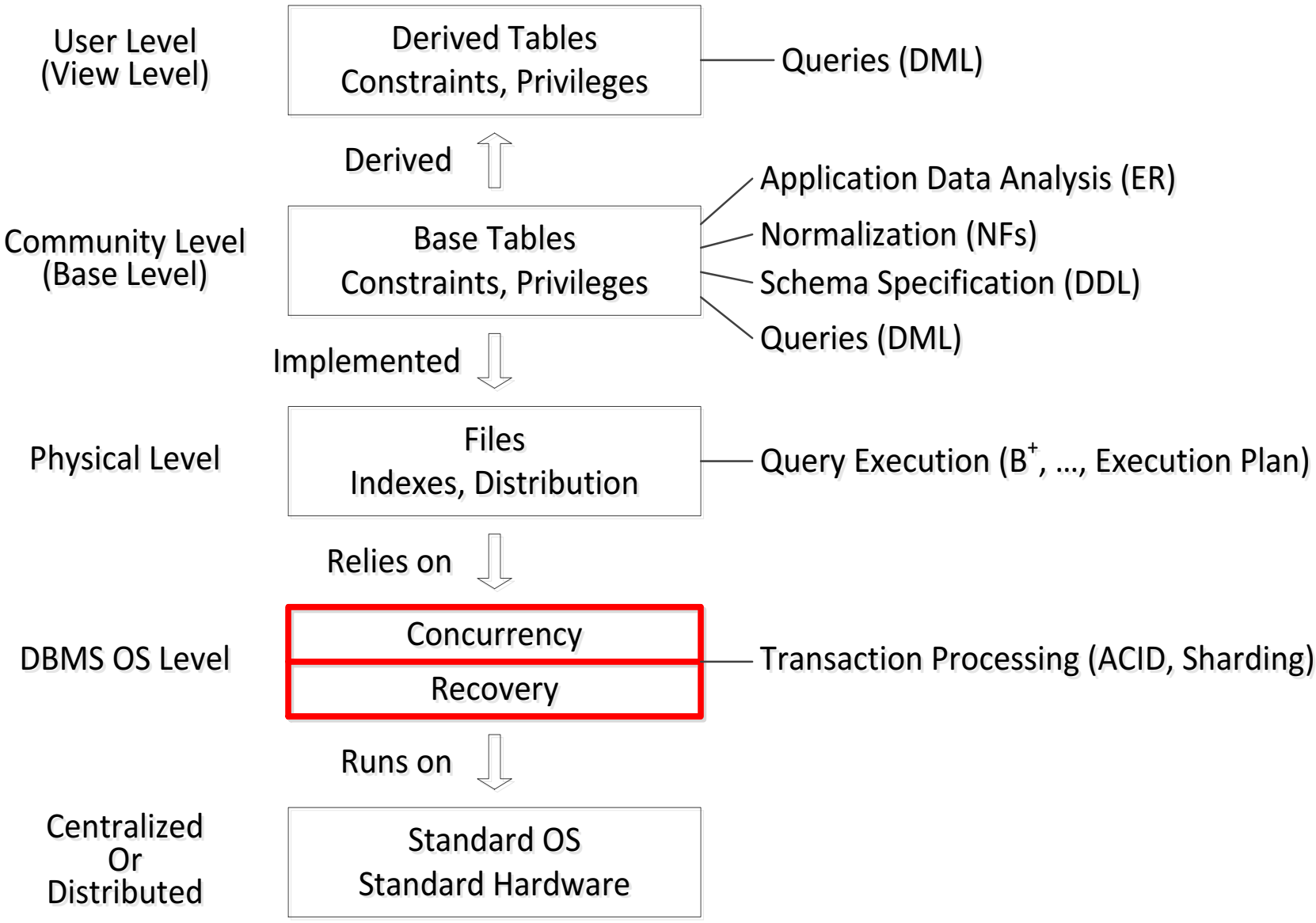
Unit 9
Transaction Processing: Recovery

Recovery in Context



Example: Money Transfer

- Transfer $5 from account a to b (items on the disk)

  1. transaction starts
  2. read a into xa (local variable in RAM)
  3. xa := xa − 5
  4. write xa onto a
  5. read b into xb (local variable in RAM)
  6. xb := xb + 5
  7. write xb onto b
  8. transaction ends

- If initial values are a = 8 and b = 1

  then after the execution a = 3 and b = 6

Example: Money Transfer

- Sample values of database variables at various points in a completed execution

		a	b
		8	1
1.	transaction starts	8	1
2.	read a into xa	8	1
3.	xa := xa − 5	8	1
4.	write xa onto a	3	1
5.	read b into xb	3	1
6.	xb := xb + 5	3	1
7.	write xb onto b	3	6
8.	transaction ends	3	6

Example: Money Transfer

- If the state of RAM is lost between instructions 1 and 8, it is not known which was the last instruction executed
- Thus in general, neither of the following naïve recovery procedures will work:
  - **re-execute** the transaction

    Wrong, because if the transaction crashed after instruction 4, incorrect values (a = 3,...) will exist in the database.
  - **do not re-execute** the transaction

    Wrong, because if the transaction crashed before instruction 7, incorrect values (..., b = 1) will exist in the database

Transactions

- Transaction is an execution of a user's program
- In the ***cleanest and most important*** model a transaction is supposed to satisfy the ***ACID*** conditions
- ***Atomic***
  - It is not logically divisible into smaller units of executions
  - It is either executed to completion (was ***committed***), or not executed at all (was ***aborted****, also called* ***rolled back***)
- ***Consistent***
  - It preserves the consistency of the database, when running by itself (without any other transactions executing concurrently)
  - If started on a "correct" database, and "successfully" finished, it will leave a "correct" database
  - "Correctness" means: satisfies integrity constraints as specified to the database
  - This is somewhat weak, as correctness presumably should model real world not just internal consistency

Transactions

- ***Durable***
  - Once it completed “successfully,” the values it produced will never be forgotten and “in effect” will be installed in the database
  - On the disk, which is the only thing that counts as a disk is considered persistent storage, i.e., “data on it cannot be lost accidentally”
  - So we assume that disks are perfect: this is OK to understand the issues
  - RAM can be lost when the system crashes
  - Of course the values can be overwritten by transactions that come later
- ***Isolated***
  - A transaction is given the illusion of running on a dedicated system, so errors because of concurrent execution cannot be introduced
  - It does not interact with other transactions running concurrently
  - It does not see what they do, they do not see what it does: effects as if each executed on the database by itself

Recovery and Concurrency Management

- The job of these recovery/concurrency modules of the database operating system is to assure the ***ACID*** properties and to handle other related issues
- The two modules are quite intertwined and cooperate during their execution
- Recovery is more fundamental and also applicable to a single user systems
- Recovery management will be “helped” by concurrency management, as we will see in the “Concurrency” unit

Recovery Management

- The job of recovery is to make sure that the transaction satisfies ***ACD*** properties, as ***I*** is not relevant
- So the job is not to allow partial executions and to make sure that transactions once executed do not “disappear” (i.e., their “effects” remain, unless these “effects” become obsolete)
- If a failure occurred while a transaction was executing, we cannot continue, and therefore need to restore the database to the state before the failed transaction started
- If a failure occurred after a transaction finished executing (we will be more precise about the meaning of “finished executing”), the state must continue reflecting this transaction

Storage Model

- Again, we need a model
- Two level storage
  - RAM
  - Disk
- RAM (volatile storage)
  - There will be no errors but all the information on it can disappear
    - Electricity goes down
- Disk (stable storage)
  - Atomic reads/writes of blocks
  - No failures
    - This is ideal but implementable in practice, through RAIDs, disk-to-disk-to-tape backups, etc..
- While the hierarchy is much larger (registers, primary caches, secondary caches, RAM, disk, tape, etc.), we only need to be concerned about RAM vs. disk
  - Failure of RAM implies failure of all storage “closer” to computing, such as registers, etc.

Failure Types

- "Global" problem in execution
  - E.g., deadlock: a transaction needs to be removed in the middle of an execution to let the system proceed
- Transaction failure
  - Transaction's execution may result in violating CHECKs and CONSTRAINTs; then the transaction needs to be failed, that is ***aborted***
  - This is still OK as aborted transactions do not have to be correct, because their effects are going to be removed from the execution
- Machine reboots spontaneously

Our Failures: RAM State Loss

- RAM failure: the state of the computation is lost and we do not know what were the values of the variables in RAM
- This is the most interesting case and we will focus on it
- Other cases will be solved similarly, which we will not focus on
- Typical scenario: electricity goes down and the computer shuts down and then reboots forgetting everything, other than whatever has been written to the disk

Lifecycle Of A Transaction

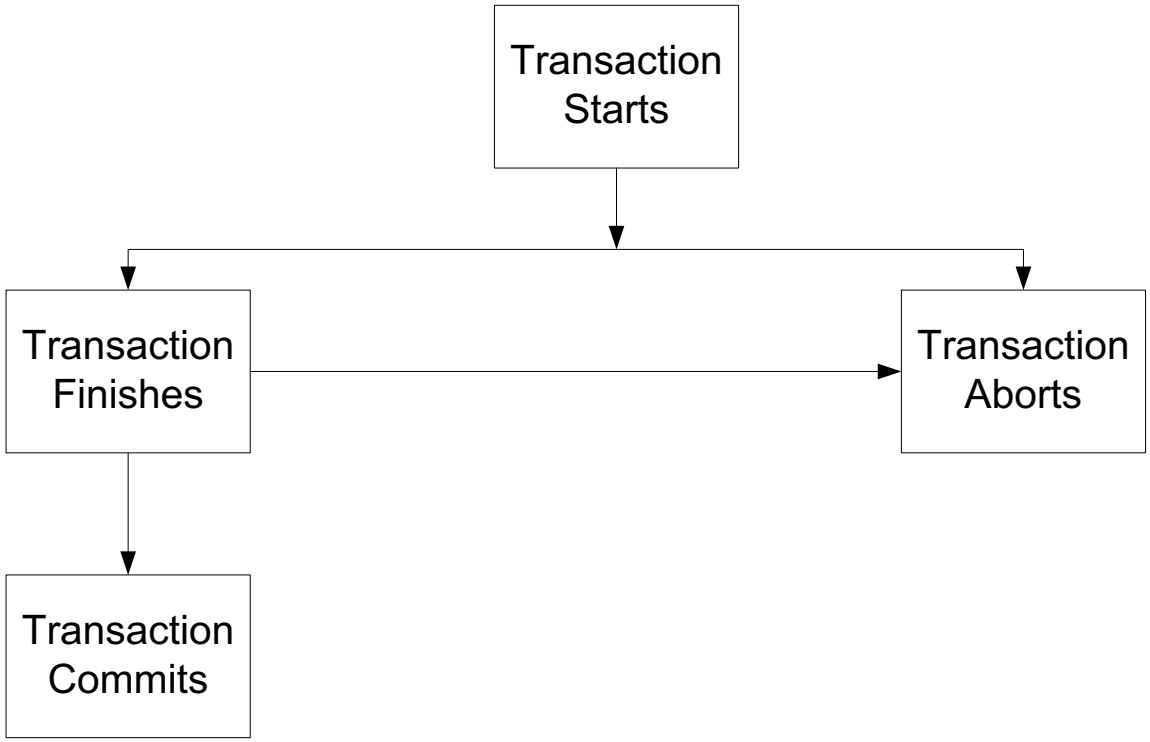
1. Start
2. Run
3. Finish
4. Check for deferred consistency requirements, if any
   - Requirement that can be violated during execution (but not after it), such as during the movement of money from checking account to savings account
   - Such deferrals can be specified using SQL DDL
5. Commit if steps 1 – 4 are done successfully: this means the transaction's effects are "durable"

   But a transaction can be aborted for any reason (or no reason) before commit

- Of course an aborted transaction can be re-executed, especially if this was not "its fault," but then this is really another transaction

The Fate Of A Transaction

- The system either succeeded in moving money from my savings account to my checking account or not



- To reiterate informally
  - **committed**: completed and permanent
  - **aborted**: never executed

History (Schedule)

- This is ***a trace of execution of transactions in a system***
  - This is somewhat imprecise at this point
  - We will make it more formal as needed
- Example at the beginning of the unit was a trace
- At different time we will “trace” different actions
  - Most commonly which transaction accessed which item in which mode
  - E.g. T1 R x; meaning transaction T1 Reads item x
  - E.g. T1 W x; meaning transaction T1 Writes item x

- We will start with some basic, but **extremely important**, concepts, constrains, and principles that need to be satisfied by any recovery mechanism

Recoverable Histories

- A history is ***recoverable*** if for every transaction T that commits, the Commit of T follows the Commit of every transaction from which T read
  - We do not formally define here what it means "from which T read," but it is pretty obvious intuitively: T read the result of writing by that other transaction
- ***Histories must be recoverable, otherwise a transaction may operate based on non-existent past***

- A history that ***is not recoverable***

T1	T2
W x	
	R x
	W y
	Commit

Non-Recoverable Histories

- Here is what could happen next

T1	T2
W x	
	R x
	W y
	Commit
Abort	

- And now what to do about T2?
  - It operated using a non-existent value of x (here y could have been x + 1)
  - It ***cannot*** be aborted because it has already committed

More On Non-recoverable Histories

- We need to understand what is wrong with the following

T1	T2
W x	
	R x
	Commit
Abort	

- It may seem that there is no problem because T2 only read
- Consider the following case:
  - Initially x = 0 and y = 0
  - T1 is: x := 1
  - T2 is: **if** x = 0 **then** y := 1
- Then the value of x did matter and depending on whether T1 existed or not the behavior of T2 is different

Cascading Aborts

- A history ***avoids cascading aborts*** if every transaction reads only values produced by transactions that have already committed
- Of course it can read the initial state unless overwritten previously by another transaction
  - We will assume, to simplify discussion, some transaction T0, running by itself and creating the initial state of the database
  - T0 is recoverable, automatically
- A history that ***does not avoid cascading aborts***

T1	T2
W x	
	R x
	W y

Cascading Aborts

- Here is what could happen next

T1	T2
W x	
	R x
	W y
Abort	

- Then we must do:

T1	T2
W x	
	R x
	W y
Abort	
	Abort

Strict Histories

- History is ***strict*** if
  - Satisfies the condition for avoiding cascading aborts
  - For every transaction, if it writes an item, all the transactions that previously wrote that item have already committed or aborted
- If we need to abort a transaction that wrote a value, the most convenient thing to implement is just to restore the value that existed previously
- If a history is not strict, this is not good enough

- A history that ***is not strict***

T1	T2
W x	
	W x

Strict Histories

- Here is what could happen next

T1	T2
Wx	
	Wx
Abort	

- Even though T1 aborted, we must not do anything to x!

Strict Histories

- Here what could happen next

T1	T2
Wx	
	Wx
Abort	
	Abort

- Because T2 aborted, we must restore the value of x that existed before T1 wrote, sometime in the past

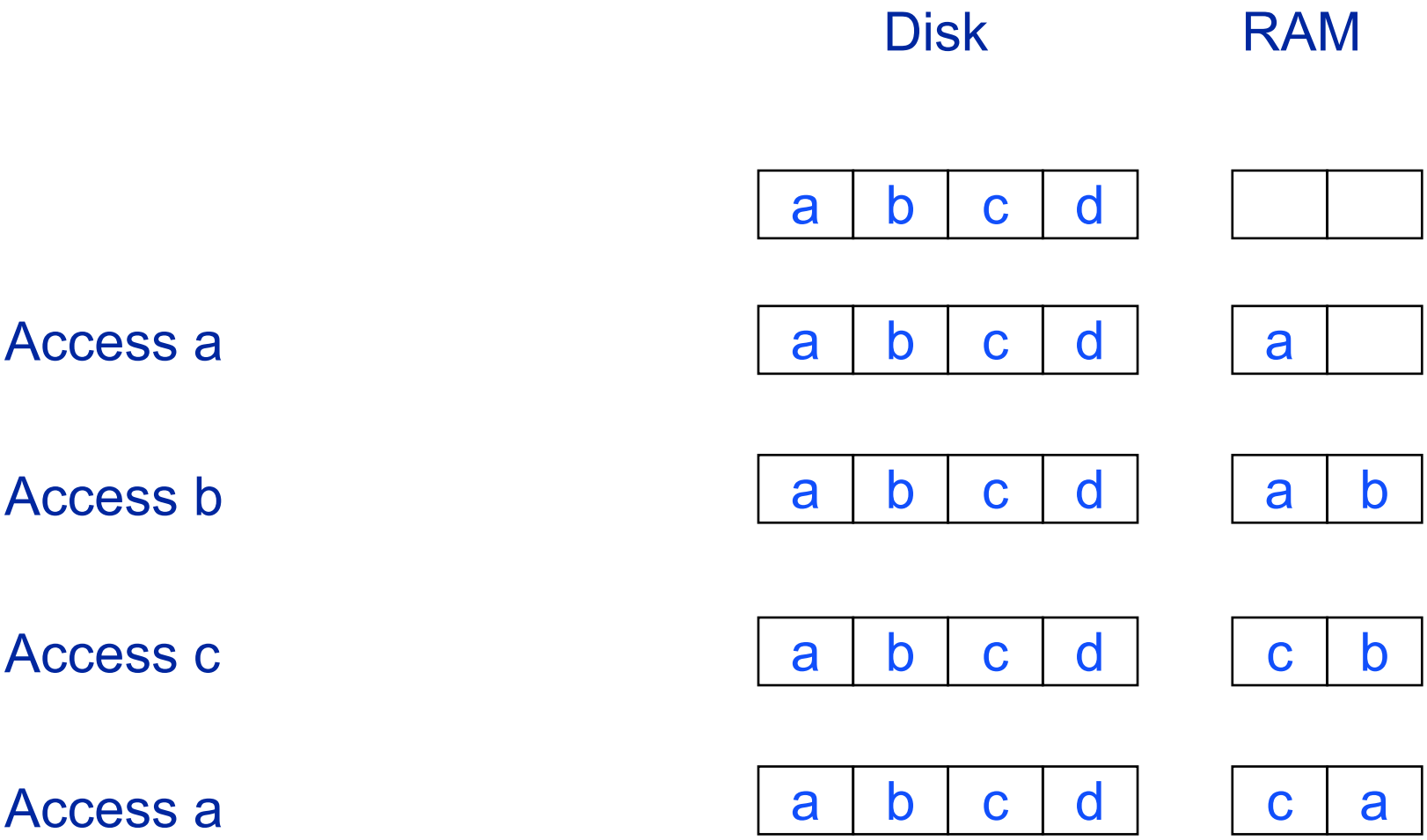
- Very complicated; need to maintain several values for x

Relations

- ***Strict implies no cascading aborts***
- ***No cascading aborts implies recoverable***

- ***So we like strict, which really means:***
  - ***Every transaction reads only values produced by transactions that have already committed***
  - ***Every transaction, if it writes an item, all the transactions that previously wrote that item have already committed or aborted***
- We will assume that the DB OS will make sure that all histories will be strict
- This will be automatically ensured "magically" by concurrency management (next unit)

Reminder On Virtual Memory: Paging In And Out



- "Access" means read or write
- 4 blocks on the disk and 2 page slots in the RAM cache

The Scenario And The Meta-Principle

- Our scenario: when the system crashes
  1. The state of the RAM is lost
  2. The information on the disk is not corrupted by the crash

- ***Our meta-principle: The information on the disk must be sufficient to produce a consistent state of the database, which***
  1. ***Reflects all the committed transactions***
  2. ***Does not reflect any of the non-committed transactions***

General Setting And "Principles"

- A transaction modifies "items" by changing an "old" value into a "new" value

1. ***At any time before the commit, the old values must be in stable storage (on the disk)***
   - Because old values must be remembered if the RAM fails
   - Very important note: on the disk but not necessarily in the database
2. ***At any time after the commit, the new values must be in stable storage (on the disk)***
   - Because the new values must be remembered if the RAM fails
   - Very important note: on the disk but not necessarily in the database
3. ***The transaction commits exactly when this "fact" is written in stable storage (on the disk)***
   - Because we must remember this if the RAM fails

Principles Applied in Practice

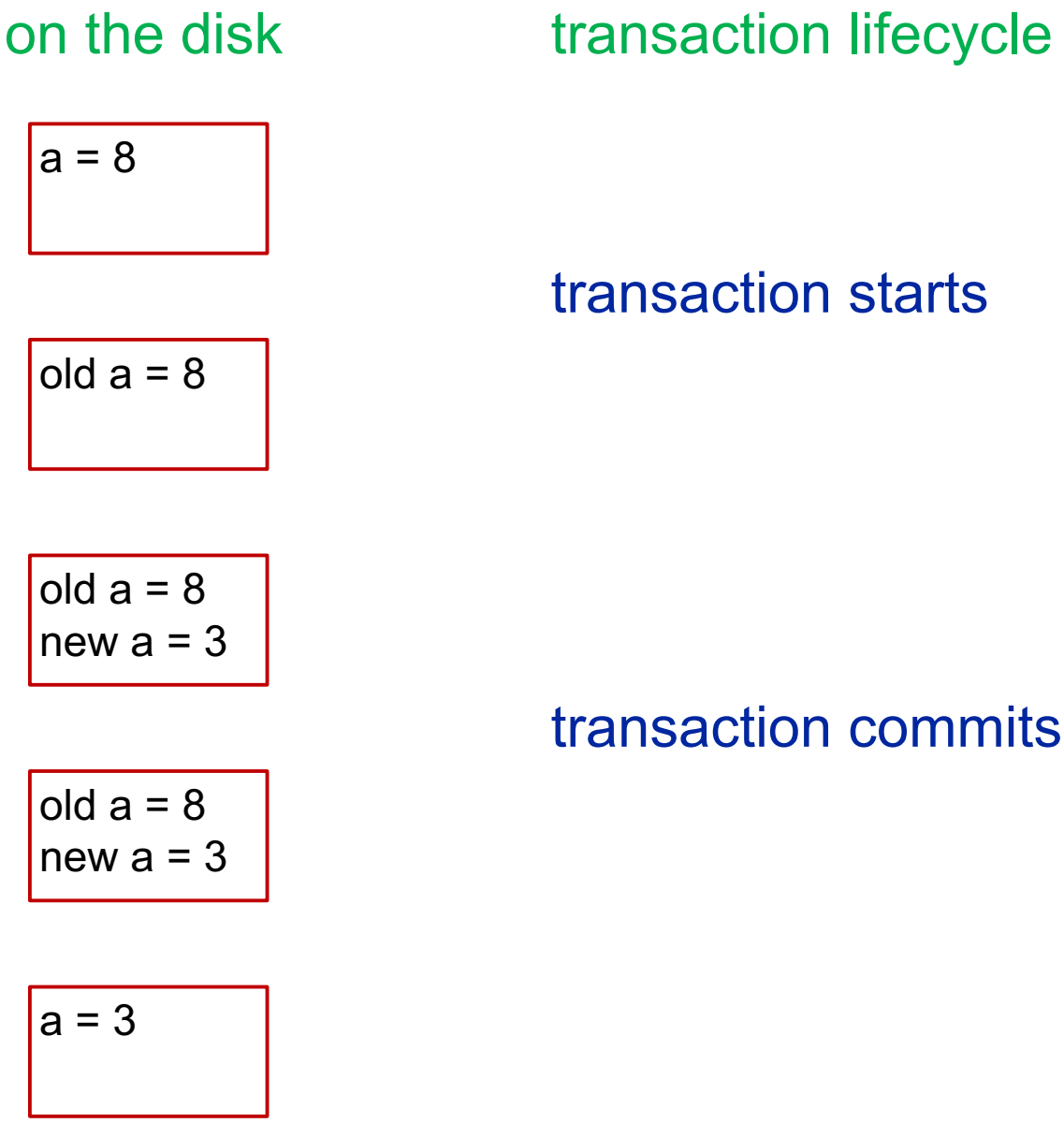
- While the principles are clear, their implementations are ultimately very complex
- The reason for this is efficiency
- We will not go to the ultimate implementations but will stop "at the tip of the iceberg"

- We will start with a very simple example and then we will discuss more realistic scenarios and implementations

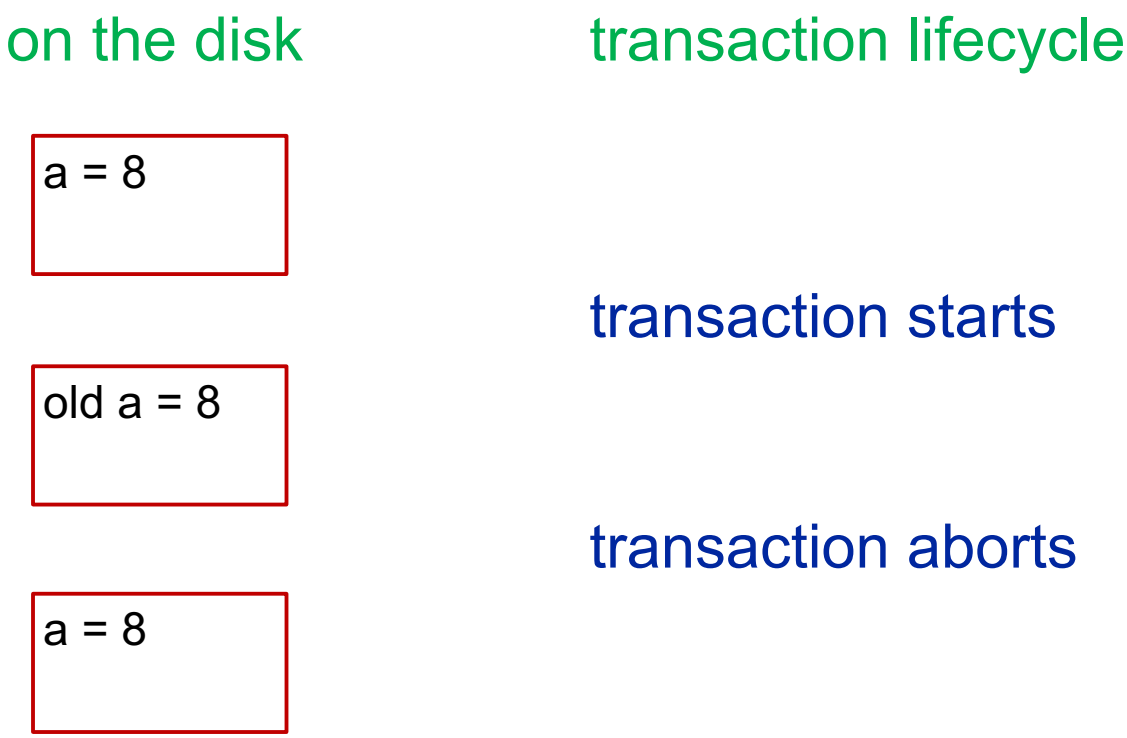
A Very Simple Transaction And a Very Simple Scenario

- Normal execution
  - The transaction reads item a from the disk, which happens to be 8
  - The transaction changes a to be 3
  - The transaction commits
- During execution
  - We need to keep the old value of a until (at least) the instance of commit
  - We need to keep the new value of a (at least) since the instance of commit
- Only one action can be done in a single instance of time
  - So for some period of time we may need to keep both the old and the new values

A Simplified History
Transaction Committed



A Simplified History
Transaction Aborted "Early"



A Simplified History
Transaction Aborted "Late"

on the disk

transaction lifecycle

a = 8

transaction starts

old a = 8

old a = 8
new a = 3

transaction aborts

old a = 8
new a = 3

a = 8

Simplified Scenario: Immediate Writes To Disk

- We will discuss first a simplified scenario, which is the conceptual foundation to what happens in real systems
- We ignore virtual memory and assume that writes to the database on the disk happen "immediately" after the values are changed in RAM
- We will explain the most common method of managing recovery: **Write Ahead Log**
- The basic idea

  Record how you are going to modify the database before you actually do it

- We return to our old money transfer example and consider the situation when
  - RAM values "disappear" because the system crashes
  - Disk values "do not disappear"

Example: Money Transfer

- Sample values of database variables at various points in a completed execution
- We move 5 from a to b

	a	b
	8	1
1. transaction starts	8	1
2. read a into xa	8	1
3. xa := xa − 5	8	1
4. write xa onto a	3	1
5. read b into xb	3	1
6. xb := xb + 5	3	1
7. write xb onto b	3	6
8. transaction ends	3	6

RAM and Disk

- We will have the following:

- Database on disk with two items: a and b
- RAM with two items xa and xb
- Log: a *sequential* file on the disk consisting of records of the following types:

- [T starts]; abbreviated as [T s]
- [T commits]; abbreviated as [T c]
- [T item old-value new-value]

We only consider one transaction, but in general there are many transactions, so “T” needs to be written in the log

Trace Of Execution: Items And Log

xa	xb	a	b	log
?	?	8	1	

Trace Of Execution: Items And Log

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]

Trace Of Execution: Items And Log

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]

Trace Of Execution: Items And Log

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]

Trace Of Execution: Items And Log

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]

Trace Of Execution: Items And Log

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]
3	?	3	1	[T s][T a 8 3]

Trace Of Execution: Items And Log

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]
3	?	3	1	[T s][T a 8 3]
3	1	3	1	[T s][T a 8 3]

Trace Of Execution: Items And Log

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]
3	?	3	1	[T s][T a 8 3]
3	1	3	1	[T s][T a 8 3]
3	6	3	1	[T s][T a 8 3]

Trace Of Execution: Items And Log

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]
3	?	3	1	[T s][T a 8 3]
3	1	3	1	[T s][T a 8 3]
3	6	3	1	[T s][T a 8 3]
3	6	3	1	[T s][T a 8 3][T b 1 6]

Trace Of Execution: Items And Log

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]
3	?	3	1	[T s][T a 8 3]
3	1	3	1	[T s][T a 8 3]
3	6	3	1	[T s][T a 8 3]
3	6	3	1	[T s][T a 8 3][T b 1 6]
3	6	3	6	[T s][T a 8 3][T b 1 6]

Trace Of Execution: Items And Log

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]
3	?	3	1	[T s][T a 8 3]
3	1	3	1	[T s][T a 8 3]
3	6	3	1	[T s][T a 8 3]
3	6	3	1	[T s][T a 8 3][T b 1 6]
3	6	3	6	[T s][T a 8 3][T b 1 6]
3	6	3	6	[T s][T a 8 3][T b 1 6][T c]

Recovery Under Various Conditions
Look Only At The Log

```
a    b    log

8    1
```

- Do nothing

Recovery Under Various Conditions Look Only At The Log

```
a    b    log

8    1    [T s]
```

- Do nothing

Recovery Under Various Conditions Look Only At The Log

```
a   b   log

?   1   [T s][T a 8 3]
```

- Write 8 into a

Recovery Under Various Conditions Look Only At The Log

```
a   b   log

?   ?   [T s][T a 8 3][T b 1 6]
```

- Write 1 into b and write 8 into a

Recovery Under Various Conditions Look Only At The Log

```
a   b   log

?   ?   [T s][T a 8 3][T b 1 6][T c]
```

- Write 3 into a and write 6 into b

Recovery Procedure

- The system crashes (RAM “disappears”)
- We look at the log.
- If there is [T s] but no [T c], we copy the old values of a and b from the log onto the database on the disk
- If there is [T c] (and therefore also [T s]) we copy the new values of a and b from the log onto the database on the disk

- We then continue: reboot the database and continue processing
- If we restored old values, we re-execute the transaction

Delaying Writes

- It is not necessary to update the database with the new values by any particular deadlines, as long as the most recent value is in RAM
- Not performing database updates is likely to speed up performance

- A transaction is always given the current value of an item
- If it is in RAM, from RAM
- If it is not in RAM, then it is read from disk into RAM

- If there is a need to recover, this is done based on the log which has everything that is needed

- So next, we revisit our example, when one update to the disk was not done

Example: Money Transfer With Delayed Writes

- Sample values of database variables at various points in a completed execution
- We move 5 from a to b

	a	b
	8	1
1. transaction starts	8	1
2. read a into xa	8	1
3. xa := xa − 5	8	1
4. write xa onto a	3	1
5. read b into xb	3	1
6. xb := xb + 5	3	1
7. write xb onto b	3	6
8. transaction ends	3	6

Trace Of Execution: Items And Log With Delayed Writes

xa	xb	a	b	log
?	?	8	1	

Trace Of Execution: Items And Log With Delayed Writes

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]

Trace Of Execution: Items And Log With Delayed Writes

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]

Trace Of Execution: Items And Log With Delayed Writes

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]

Trace Of Execution: Items And Log With Delayed Writes

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]

Trace Of Execution: Items And Log With Delayed Writes

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]

Trace Of Execution: Items And Log With Delayed Writes

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]
3	1	8	1	[T s][T a 8 3]

Trace Of Execution: Items And Log With Delayed Writes

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]
3	1	8	1	[T s][T a 8 3]
3	6	8	1	[T s][T a 8 3]

Trace Of Execution: Items And Log With Delayed Writes

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]
3	1	8	1	[T s][T a 8 3]
3	6	8	1	[T s][T a 8 3]
3	6	8	1	[T s][T a 8 3][T b 1 6]

Trace Of Execution: Items And Log With Delayed Writes

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]
3	1	8	1	[T s][T a 8 3]
3	6	8	1	[T s][T a 8 3]
3	6	8	1	[T s][T a 8 3][T b 1 6]
3	6	8	6	[T s][T a 8 3][T b 1 6]

Trace Of Execution: Items And Log With Delayed Writes

xa	xb	a	b	log
?	?	8	1	
?	?	8	1	[T s]
8	?	8	1	[T s]
3	?	8	1	[T s]
3	?	8	1	[T s][T a 8 3]
3	1	8	1	[T s][T a 8 3]
3	6	8	1	[T s][T a 8 3]
3	6	8	1	[T s][T a 8 3][T b 1 6]
3	6	8	6	[T s][T a 8 3][T b 1 6]
3	6	8	6	[T s][T a 8 3][T b 1 6][T c]

Note: a later update was reflected in the database but the earlier was not

Recovery Under Various Conditions Look Only At The Log

```
a    b    log

8    1
```

- Do nothing

Recovery Under Various Conditions Look Only At The Log

```
a    b    log

8    1    [T s]
```

- Do nothing

Recovery Under Various Conditions Look Only At The Log

```
a    b    log

?    1    [T s][T a 8 3]
```

- Write 8 into a

Recovery Under Various Conditions Look Only At The Log

```
a   b   log

?   ?   [T s][T a 8 3][T b 1 6]
```

- Write 1 into b and write 8 into a

Recovery Under Various Conditions Look Only At The Log

```
a    b    log

8    6    [T s][T a 8 3][T b 1 6][T c]
```

- Write 3 into a and write 6 into b

Recovery Procedure: Same as Before

- The system crashes (RAM “disappears”)
- We look at the log.
- If there is [T s] but no [T c], we copy the old values of a and b from the log onto the database on the disk
- If there is [T c] (and therefore also [T s]) we copy the new values of a and b from the log onto the database on the disk

- We then continue: reboot the database and continue processing
- If we restored old values, we re-execute the transaction

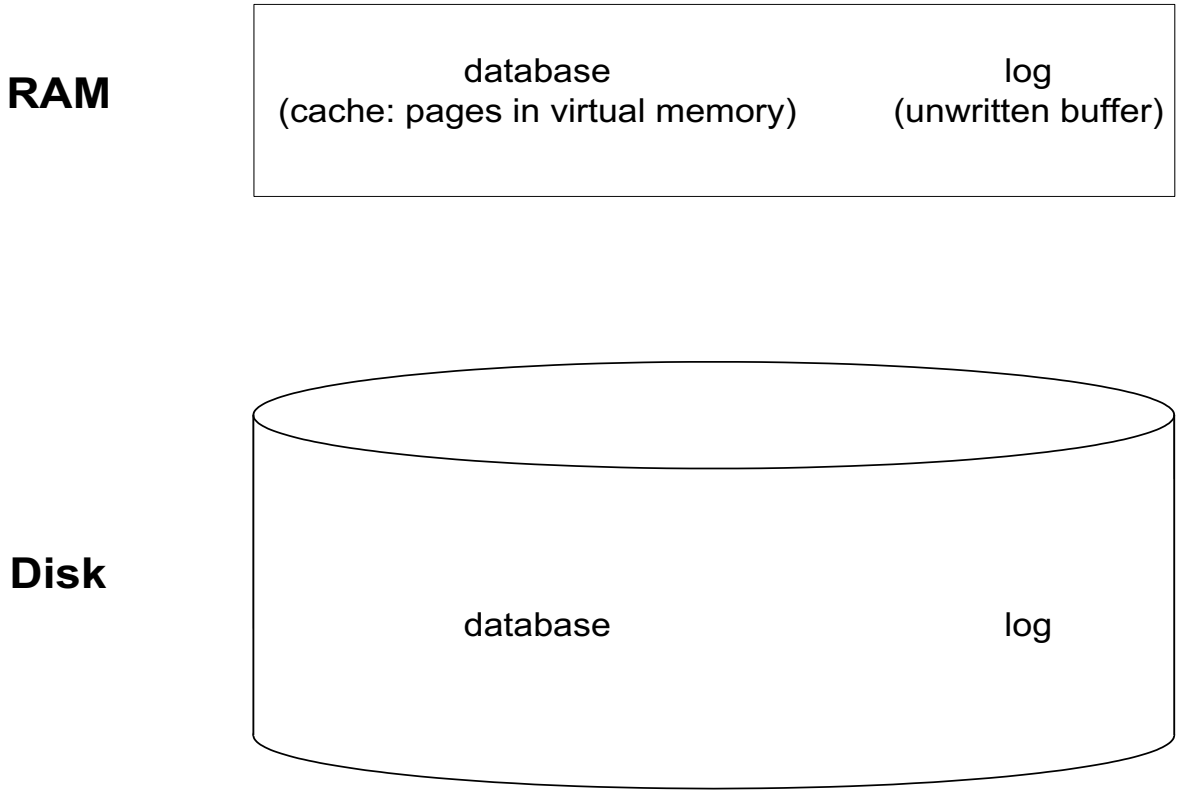
- Next: Elaborate on the protocol with virtual memory and checkpoints

Write Ahead Log

- We will discuss the most common way of handling recovery
- The log will be a sequential file, which will be produced by the DB OS
- It will describe the history of what has happened (more precisely: what is going to happen)
- It will have 4 types of records
  - T starts; this records that transaction “T” started
  - T x a b; this records that transaction “T” modified “x” with the old value being “a” and the new value being “b”
  - T abort; this records that transaction “T” was aborted
  - T commit; this records that transaction “T” was committed
- To simplify discussion, we will assume that no transaction is aborted: the only failure is the failure of the RAM
  - Once we finish, we will essentially know what to do if transactions are aborted

RAM and Disk

- "Real log" is log on disk + unwritten buffers
- "Real database" is database on disk + some more up-to-date pages in RAM (in virtual memory)
- But of course, after a failure we only see what is on the disk



Assumptions And Remarks

- Note that if a write instruction is executed, the appropriate value is generally written in a buffer in the RAM.
- At some later point in time it will be written on the disk
- If RAM fails between these two points in time, the new value does not exist on disk and therefore is lost
- We will use the term ***actually written*** to indicate that the value is written on disk (from RAM)
- As usual, in virtual memory systems, not every update to a page is written on the disk, only “sometimes” the disk is updated
- In general, only a small part of the database pages can be kept in virtual memory
- For pages in virtual memory the current value is always in virtual memory and maybe also on disk

Additional Assumptions On Execution

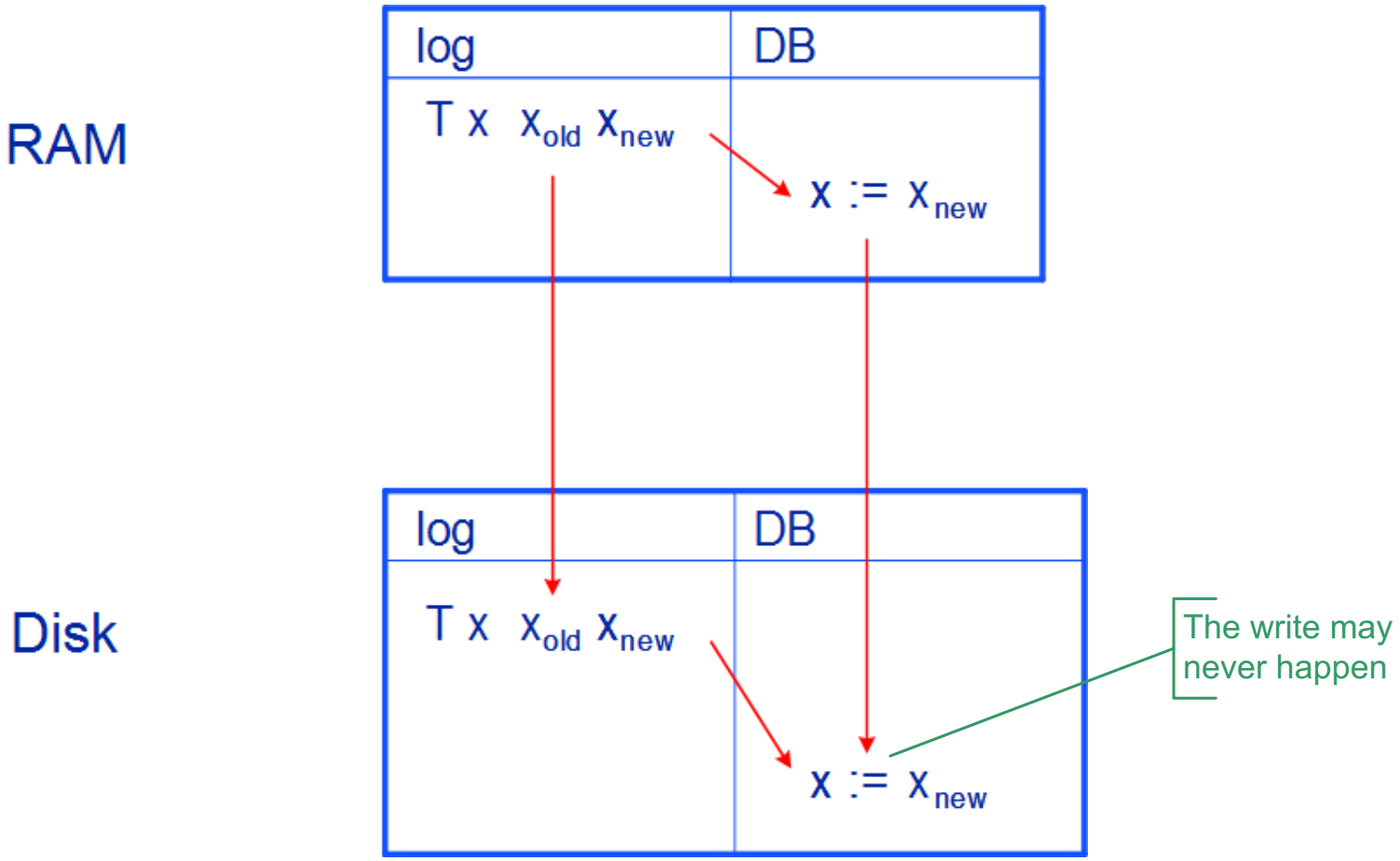
- ***The log always runs ahead of the execution in RAM*** *(not strictly necessary, but convenient to assume)*
- So if x_{old} is replaced by x_{new} in RAM, then in order
  1. The log record "T x x_{old} x_{new}" is written in the log buffer (in RAM)
  2. x_{new} is written in the appropriate virtual page (in RAM)
- ***On the log, values are written in the same sequential order in which the write instructions are issued because it is a sequential file***

- If the transaction executes a read of an item x, it gets the latest value
  - It obtains it from the virtual memory pool or if not there, from the database itself

Additional Assumptions On Execution

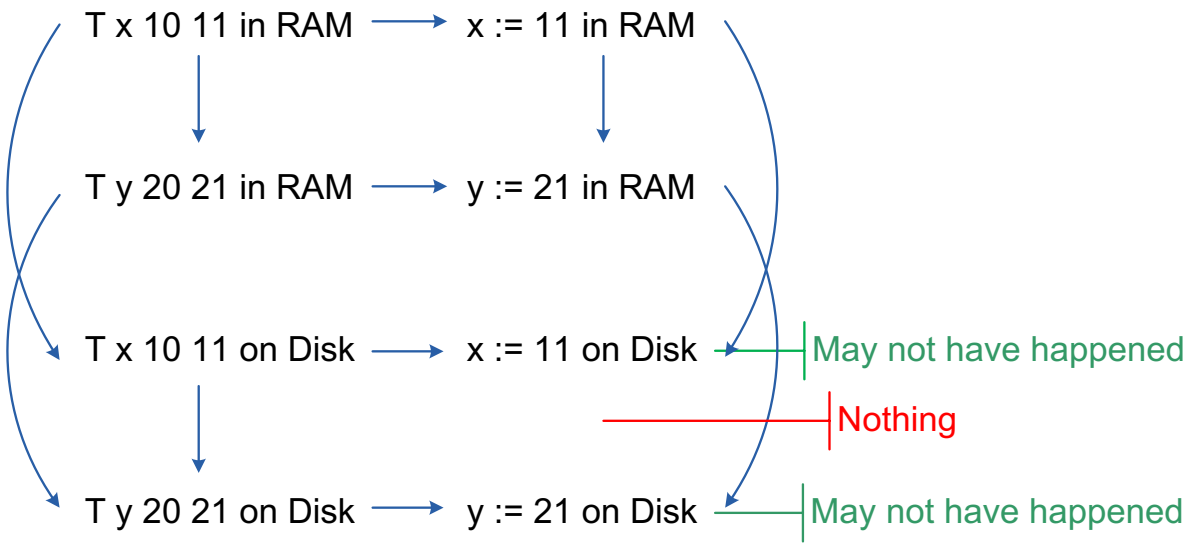
- ***The log always runs ahead of the execution on the Disk***
- So if x_{old} is replaced by x_{new}, then in order
  1. The log record "T x x_{old} x_{new}" is actually written in (actually appended to) the log (but this may not take place if there is a failure before this is done
  2. x_{new} is actually written on the disk (but this may actually never happen even if there are no failures, we will see why later)
- On the log, values are actually written in the same sequential order in which the write instructions are issued because the log is a sequential file

Temporal Ordering of Constraints On Execution



"Tracing" Two Writes And Their Temporal Ordering Constraints

- Originally, x = 10 and y = 20
- In the execution:
  1. x := 11
  2. y := 21
- Eight actions
  a. x := 11 in RAM
  b. y := 21 in RAM
  c. x := 11 on Disk
  d. y := 21 on Disk
  e. T x 10 11 in RAM
  f. T y 20 21 in RAM
  g. T x 10 11 on Disk
  h. T y 20 21 on Disk



Two Transactions (Programs)

- Transactions T1 and T2
- Initial values: a = 100, b = 300, c = 5, d = 80, e = 60, f = 70
- The instructions labeled with roman numerals are really the writing of values into the database (first in RAM then later perhaps on the Disk)

T1	T2

read(a);
read(b);
i: b := a + b;
read(c);
ii: c := 2c;
iii: a := a + b + c + 50;
	read(e);
	iv: e := e − 10;
	read(a);
	v: a := a + 10;
	read(d);
	read(b);
	vi: d := d + b;

A History Of An Execution In RAM

T1	T2

read(a);
read(b);
i: b := a + b;
 	
read(c);
ii: c := 2c;
 	
iii: a := a + b + c + 50;
	read(e);
	
	iv: e := e − 10;
	
	read(a);
	v: a := a + 10;
	read(d);
	read(b);
	vi: d := d + b;

Write Instructions For The History

- Log records

1. T1 starts
2. T1 b 300 400
3. T2 starts
4. T1 c 5 10
5. T2 e 60 50
6. T1 a 100 560
7. T1 commits
8. T2 a 560 570
9. T2 d 80 480
10. T2 commits

- Database items

(i)	b	400
(ii)	c	10
(iv)	e	50
(iii)	a	560
(v)	a	570
(vi)	d	480

Temporal Constraints

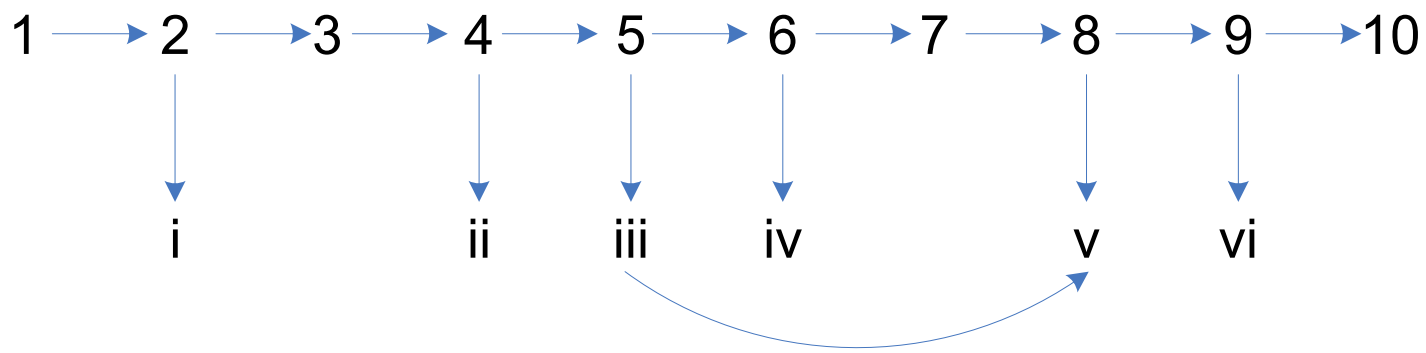
- There are certain temporal constraints, which we indicate by "**<**"

  If ***action1 < action2***, ***where the actions are actual writes***, this means that necessarily, in any execution, action1 has to come before action2

- 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10
  - Because actual write to a sequential file (the log), are processed in order of issuing of "non-actual" writes
- 2 < i, 4 < ii, 5 < iii, 6 < iv, 8 < v, 9 < vi
  - Because actual writing to the log is ahead of actual writing to the database

Temporal Constraints

- iii < v
  - Because an “older” value of a could not have existed in RAM after a “newer” value was produced, therefore
  - If v is actually written, it has been in RAM (for some time) and iii is not longer in RAM.
- Note however that iii need not have taken place!
  - Because, the disk is only “sometimes” updated (when there is no room in virtual memory or maybe when we “force” update of the blocks on the disk)



A Possible Order of Actions

- The order of actual writes might be:

log	database
1	
2	
3	
4	
	ii
	i
5	
6	
7	
8	
9	
	iv
	vi
	v
10	

Possible Situation After "write d" Was Issued

Disk

DB	log
a = 100 b = 400 c = 10 d = 80 e = 60 f = 70	1. T1 starts 2. T1 b 300 400 3. T2 starts 4. T1 c 5 10 5. T2 e 60 50 6. T1 a 100 560 7. T1 commits 8. T2 a 560 570

RAM

DB	log
a = 570 b = 400 d = 80	9. T2 d 80 480

- Actual writes processed in order: 1, 2, 3, 4, ii, i, 5, 6, 7, 8
- Notes concerning "write d":
  - It has already been reflected in the log in RAM
  - It has not yet been reflected in the DB in RAM
  - It has not yet been reflected in the log on Disk
  - It has not yet been reflected in the DB on Disk

Disaster Strikes

- Assume a RAM failure occurs (“crash”)
- The log on the disk is examined: all of it “from the beginning of time”
- Various actions are taken for the variables in the database
- There are two possibilities for each transaction in the log
  1. ***It has a commit record; therefore it has committed***
  2. ***It does not have a commit record; therefore it has not committed***
- Recall: we assume no transaction was aborted during execution

Recovery

- Transaction that has ***not committed***

  In this case we make sure that the values it produced ***are not reflected*** in the database

  We perform ***undo*** = copy old values from log to database (and later, after recovery ends, rerun the transaction—we need to do it for all "correct" submitted transactions that have not completed as it is not their fault that they have been "undone")

- Transaction that ***has committed***

  In this case, we make sure that the values it produced ***are reflected*** in the database

  We perform ***redo*** = copy new values from log to database

Consider Our Example

- If the last instruction on the log is 8 (as in our example), then we have to do the following
  - undo T2
  - redo T1
- Because we know that
  - T2 has not committed, but the database may be contaminated by some values produced by T2 and we need to restore the state as if T2 never existed (in fact in our example no such contamination took place, but could have)
  - T1 has committed and the log and the database together contain all the information describing the state after T1 finished

The Algorithm With Some Details Missing Will Explain Later As Part Of Checkpointing

- Undo all transactions for which the log has “start” but no “commit”
- Redo all transactions for which the log has both "start" and “commit”
- Remarks:
  - If the system crashes during the recovery stage, recovery needs to be started again; this may have to be done repeatedly
  - All such “recoveries” together must be equivalent to a single recovery: ***recovery must be idempotent***
  - In this algorithm, a large number of transactions need to be redone, since we do not know how far behind the log the database updates are
- Note: We have to scan the log between (a) the point it was started: i.e., when the system was turned on last, and (b) its end
  - We assume that before the system is turned on (in the current session), the Disk DB is completely up to date

Checkpointing

- A technique to obviate the need to look at the complete log and thus reduce the work during recovery
- During the execution in addition to the activities of the previous algorithm, periodically perform checkpointing in this precise order:
  1. Stop processing (suspend transactions)
  2. Force log buffers on Disk log (“force” means “actually write”)
  3. Force database buffers on Disk database
  4. Write on the log the list of transaction currently running (currently suspended)
  5. Write CHECKPOINT DONE on the log
  6. Resume processing
- Checkpointing in fact synchronizes the database with the log
  - The database reflects everything actually written in the log by this time

Checkpointing

- We will simplify and assume that we can write in one block a checkpoint record that both list the transactions and that checkpointing was done
- So we combine steps 4 and 5
- During the execution in addition to the activities of the previous algorithm, periodically perform checkpointing in this precise order:
  1. Stop processing
  2. Force log buffers on Disk log (“force” means “actually write”)
  3. Force database buffers on Disk database
  4. Write and force “checkpoint record” on log
     - List of transactions running at this time
     - The fact that we have done checkpointing
  5. Resume processing

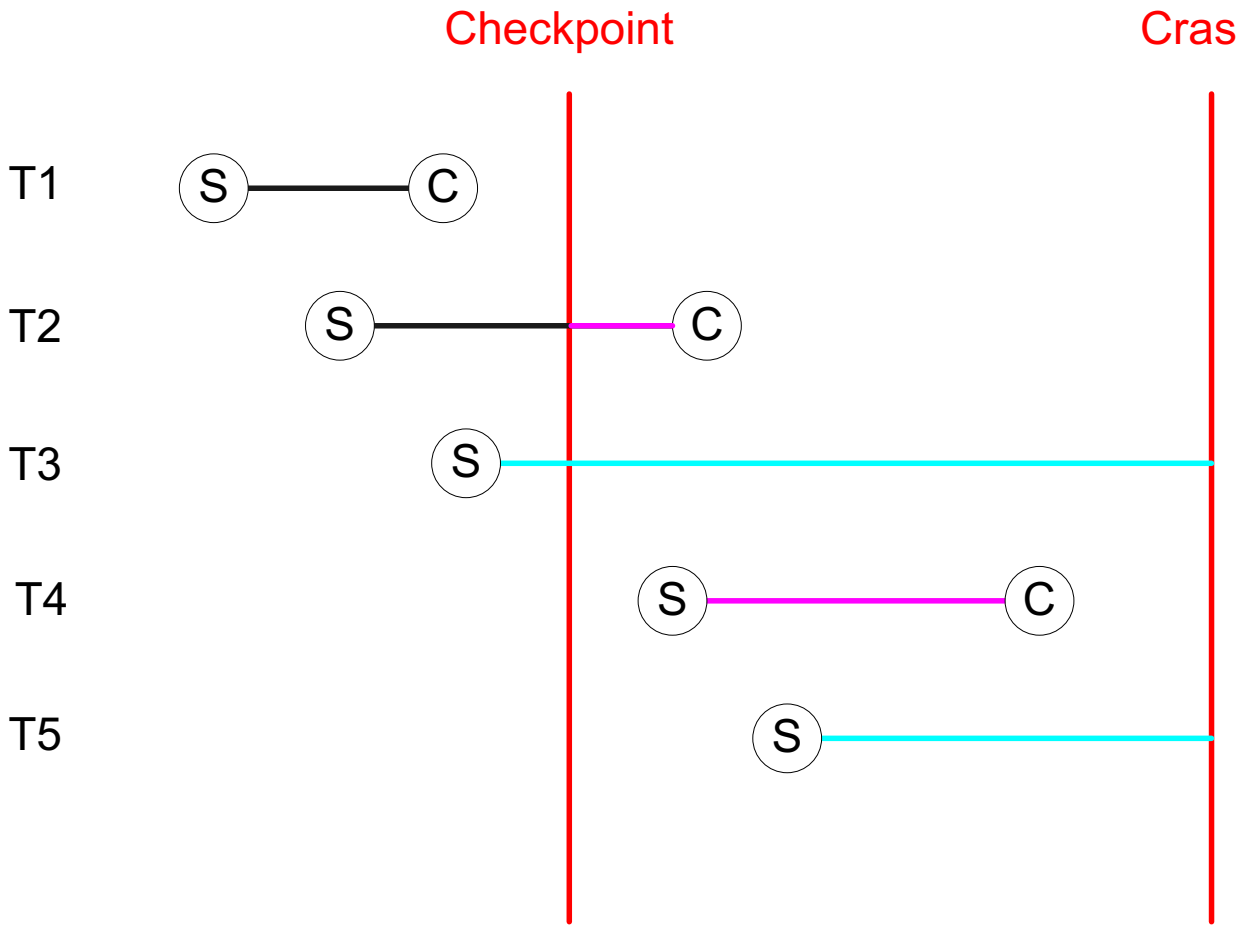
Checkpointing

- This is not efficient enough
- We should not stop processing for checkpointing because then transactions (which may be interactive) need to be suspended
- There are a little more complex checkpointing mechanisms, of incremental nature, that are significantly more efficient
- We do not cover them here

Recovery With Checkpointing

- Start scanning the log you see on the disk ***backwards*** from the end until you reach the first checkpoint record you see, producing 2, initially empty, lists (sets)
  - Undo list
  - Redo list
- For every transaction for which you have a commit record, add it to the redo list
- For each transaction for which you have a start record but not a commit record, add it to the undo list
- For each transaction that is listed in the checkpoint record for which there is no commit record, add it to the undo list
- Revised recovery algorithm:
  1. ***undo all transactions in the undo list***
  2. ***redo all transactions in the redo list (but only from the checkpoint to the end of the log)***

Recovery With Checkpointing



S Transaction started

C Transaction committed

Ignore T1
Undo T3 (all the way) and T5
Redo T2 (only from checkpoint on) and T4

Recovery With Checkpointing

1. Going backwards from the end of the log, for each record belonging to an “undo” transaction perform undo
   - can stop when all “start” records for transaction in the “undo” list have been seen
2. Going forwards from the checkpoint record to the end of the log, for each record belonging to a “redo” transaction perform redo

- If you do not do checkpointing, use the above procedure, but you have to use the beginning of the log as if it were a checkpoint record
  1. Undo going from end to beginning
  2. Redo going from beginning to end

Example Log On The Disk After A Crash

T1 starts
T1 a 0 10
T1 commits
T2 starts
T2 b 0 10
T3 starts
T3 c 0 10
T3 c 10 20
checkpoint T2 T3
T4 starts
T2 a 10 20
T4 d 0 10
T5 starts
T6 starts
T6 e 0 10
T6 aborts (needs discussing we will ignore it here)
T2 commits
T5 a 20 500
T4 commits

Practice Recovery On This Log

- We ignore (for simplicity of discussion): transaction T6 and item e
- Possible values on the disk after crash:
  - a = 10 or 20 or 500
  - b =10
  - c = 20
  - d = 0 or 10
- Undo list: T5, T3
- Redo list: T4, T2
- Actual writes
  - a := 20
  - c := 10
  - c := 0
  - a := 20
  - d := 10

Practice Recovery On This Log

- Final values
  - a = 20
  - b = 10
  - c = 0
  - d = 10
- Transactions existed (reflected): T1, T2, T4
- Transaction did not exist (not reflected): T3, T5

Big Recovery Example

```
Initial values: a = b = c = d = e = f = g = h = i = j = k = 0
On the log after the crash:
T1 STARTS
T1 a, 0, 1
T2 STARTS
T2 b, 0, 1
T3 STARTS
T3 c, 0, 1
T4 STARTS
T4 d, 0, 1
T4 e, 0, 1
T4 COMMITS
T5 STARTS
T5 f, 0, 1
T6 STARTS
T6 g, 0, 1
T2 d, 1, 2
CHECKPOINT T1, T2, T3, T5, T6
T2 h, 0, 1
T2 COMMITS
T5 h, 1, 2
```

Big Recovery Example

```
T7 STARTS
T7 i, 0, 1
T5 COMMITS
T8 STARTS
T8 d, 2, 3
T8 j, 0, 1
T8 COMMITS
T3 d, 3, 4
```

CRASH

We look at the log

Big Recovery Example

undo list: T3, T7, T6, T1

redo list: T8, T5, T2

Values of database items at different points in time

	a	b	c	d	e	f	g	h	i	j	k
initial	0	0	0	0	0	0	0	0	0	0	0
after checkpoint	1	1	1	2	1	1	1	0	0	0	0
after crash possible	1	1	1	2	1	1	1	0	0	0	0
				3				1	1	1	
				4				2			
undo	0		0	3			0		0		
redo								1			
				3				2		1	
from checkpoint		1			1	1					
unmodified											0
after recovery	0	1	0	3	1	1	0	2	0	1	0

After the recovery: effects of committed transactions

	a	b	c	d	e	f	g	h	i	j	k
T4				1	1						
T2		1		2				1			
T5					1			2			
T8				3						1	

A Few Points

- A transaction should be acknowledged, and its definition stored on the log, when accepted by the system
- The system must at some point execute it, unless it fails on its own
- Thus
  - A transaction that was aborted due to a RAM failure must be re-executed.
  - A transaction that was undone during recovery must be re-executed.
- Note that even a “completed” transaction maybe need to be redone if the commit record has not been written to the log, which can happen even if all the new values are written both to the log and to the database
  - Because we do not know that all the new values have been reflected on the disk

Problem With Interactive Transactions

- Interactive transactions are difficult to handle satisfactorily.
- How can you rollback a message to the user?
- How can you recall $100 an automatic teller has already handed out?
- Some workarounds
  - Forbid interactive transactions, or break them into smaller units of consistency that are transactions on their own
  - Send all messages after commit
    - But what if after the crash you do not know if all the messages have been sent (you cannot simultaneously send messages and record that you have done it, so danger of inconsistency); do you “send $100” again?

SQL Support

- Transaction is started implicitly, by executing a "reasonable" SQL statement
  - Or BEGIN WORK
- Transaction is ended explicitly by issuing one of the two instructions
  - COMMIT, or
  - ROLLBACK
- If the instruction is ROLLBACK, the transaction is aborted (by DB OS)
- If the instruction is COMMIT
  - Every ASSERTION and CHECK that has been declared as DEFERRABLE (in the SQL DDL specification), and therefore was not being checked during the transaction, is now automatically checked
  - If there is a failure of such a consistency requirement, the COMMIT is automatically converted to ROLLBACK and the transaction is aborted
  - If everything is OK, the transaction is committed (by DB OS)

SQL Support

- QUIT or EXIT generates implicit COMMIT

- DDL statements issue an implicit COMMIT after their execution
- This implies that change to database schema is committed too
- We do not discuss this further here

- SAVEPOINT can be added to save partial work
- This may be useful for implementing subtransactions (smaller units of consistency)
- We do not discuss this further here

Optional Material
Advanced Protocols

Refining The Model

- It is interesting and important to consider the efficiency implications of using recovery mechanism
- Some systems, such as Microsoft Access do not support recovery
- In systems that do not support recovery mechanism, at the very least, all the new data needs to be written to the database before a transaction commits
- We will refine the model and study the implications of the recovery mechanism costs
- We will still assume our standard write-ahead logging

Refining The Model

- As usual, the physical unit of access to the disk is a block, which is the same size as page in the virtual memory pool
- The logical unit of access to a file is a record, which is likely to be much smaller than a block
- So, what the log could contain is a sequence of tuples of the form

  transaction_name, record_identifier, old_value, new_value

- Such tuples (together with control tuples, such as commit records) are the records of the sequential log file

- Records of the log are buffered in RAM and when a full page/block is assembled, it is written out to the disk and a new page/block of the log is started

Refining The Model

- Records of the log are likely to be small compared to the size of the block/page
  - So many of them fit in a block

Scenario

- We will consider a stream of frequent transactions, each of them modifying one tuple of a relation, that is one record of the file storing the relation
- These transactions are submitted by interactive users (perhaps bank customers at ATM machines)
- So, the transactions have to be acknowledged as successful relatively quickly
- So, the transactions have to be committed relatively quickly

Without Recovery

- Each transaction modified only one record
- But a ***block of the database*** (containing this record) must be written before commit
- So, to commit a transaction a block must be written

With Recovery

- Each transaction modified one record
- But a ***block of the log*** containing information about the record (transaction_name, record_identifier, old_value, new_value) must be written before commit
- So, to commit a transaction this block must be written
- ***But this block contains information for many transactions***
- So a single block write commits many transactions
- So the overhead of recovery may result in much more efficient execution!

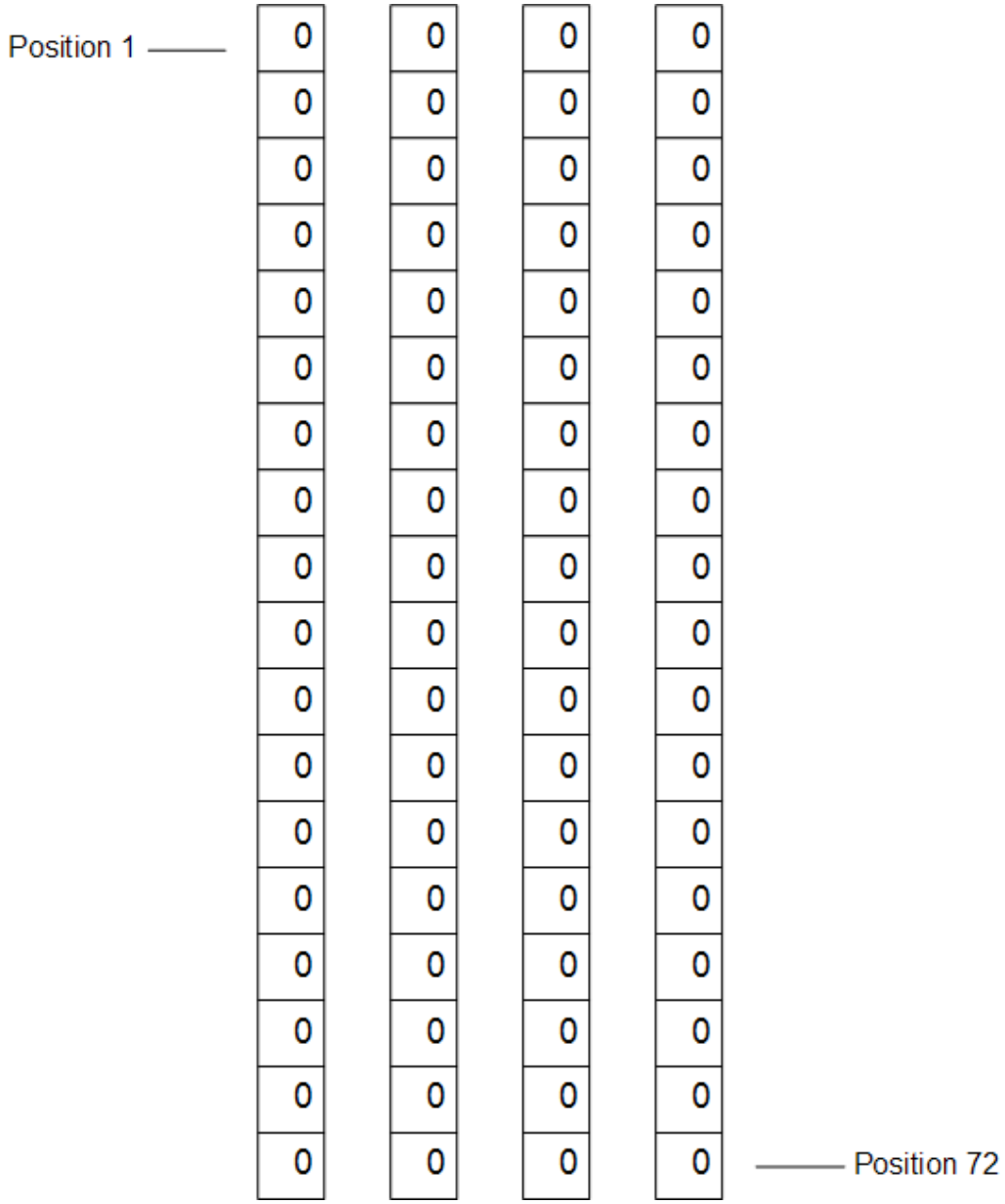
Example

- We will consider a simple example
- Our log, to simplify, will not contain transaction names
- The database is a vector of 72 integers
- A block contains 18 records, each of one integer
- There is room in the RAM for 5 blocks
- The whole database can be kept in RAM for processing, but when a transaction commits, “something” must be written to the disk

- There will be 6 transactions

Initial State Of The Memory Pool

- The vector needs 4 blocks, of 18 integers in the RAM virtual memory pool

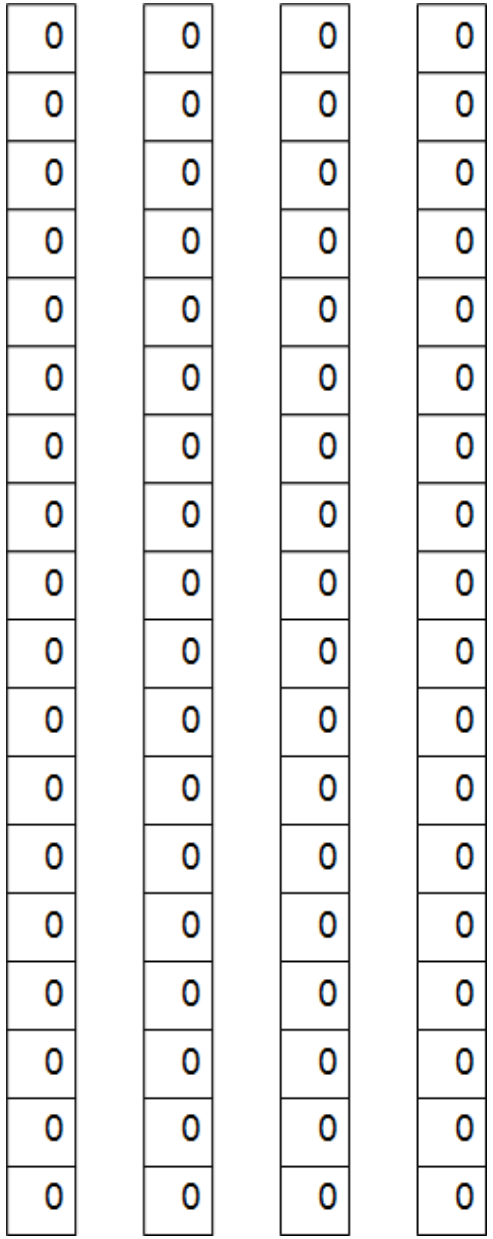


Without Recovery Mechanism

- We need to write one block to commit each transaction

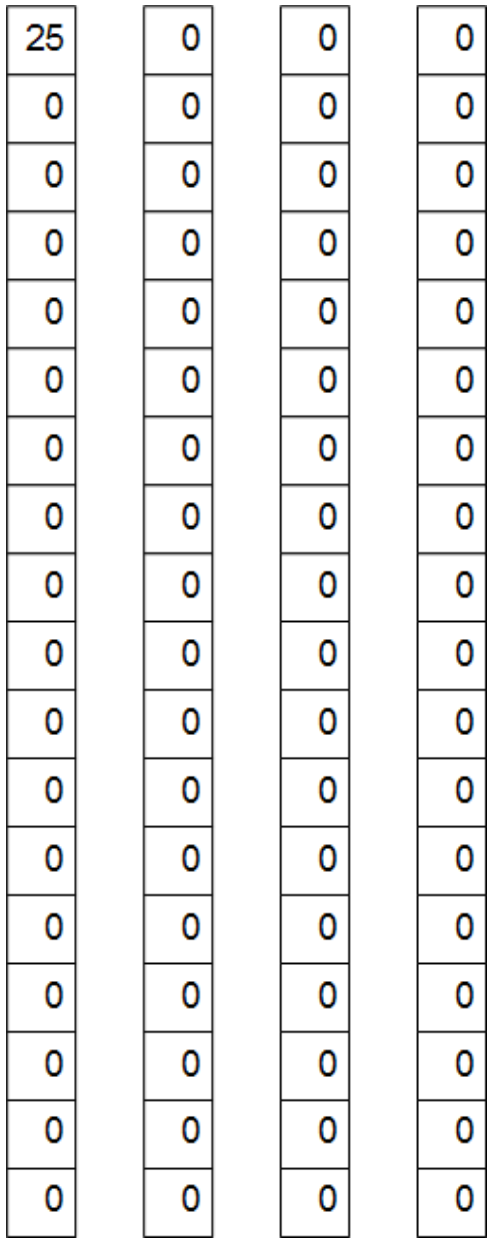
Initial State Of Memory Pool

- Initial database in RAM



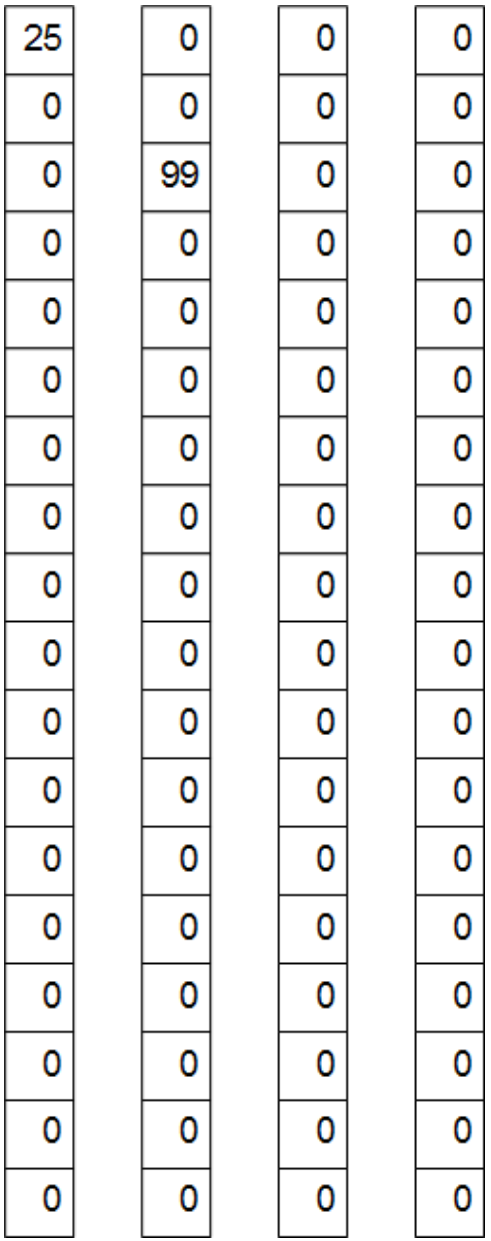
After Transaction 1

- We need the new values to be in stable storage before commit
- After an update of position 1, block 1 needs to be written out to the database before commit can be done and reported to the user



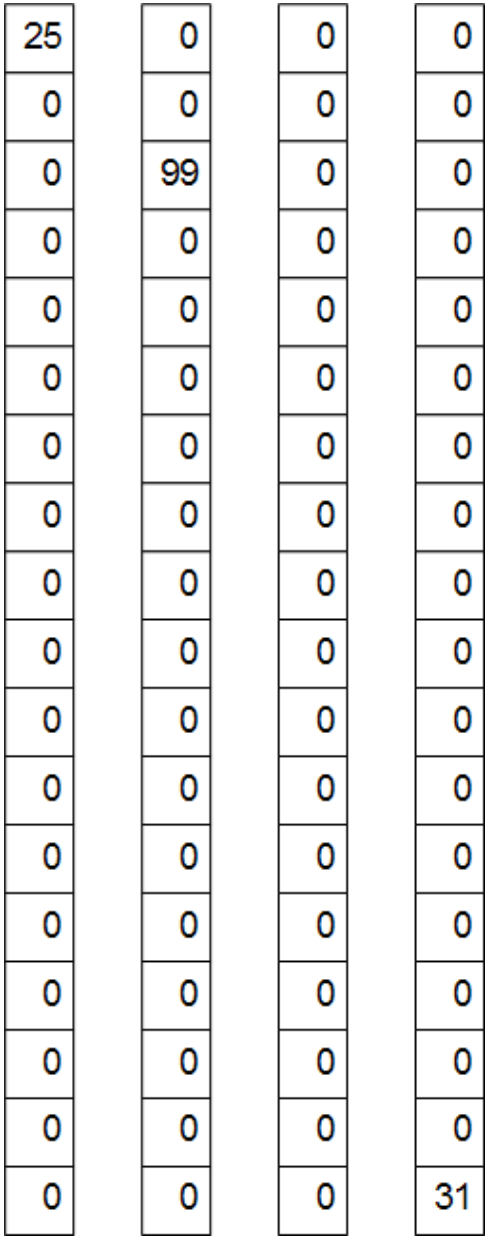
After Transaction 2

- We need the new values to be in stable storage before commit
- After an update of position 21, block 2 needs to be written out to the database before commit can be done and reported to the user



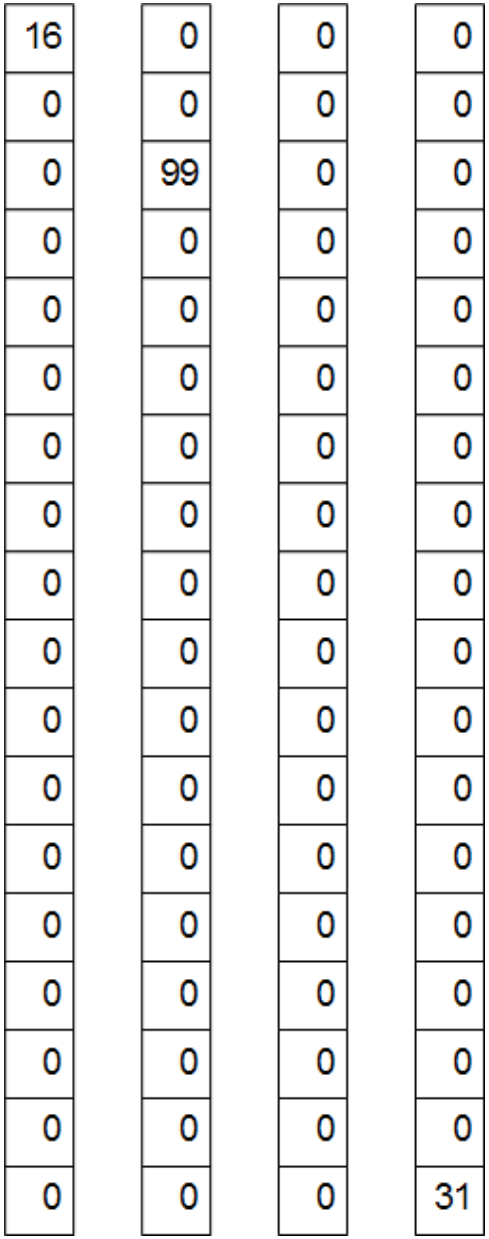
After Transaction 3

- We need the new values to be in stable storage before commit
- After an update of position 72, block 4 needs to be written out to the database before commit can be done and reported to the user



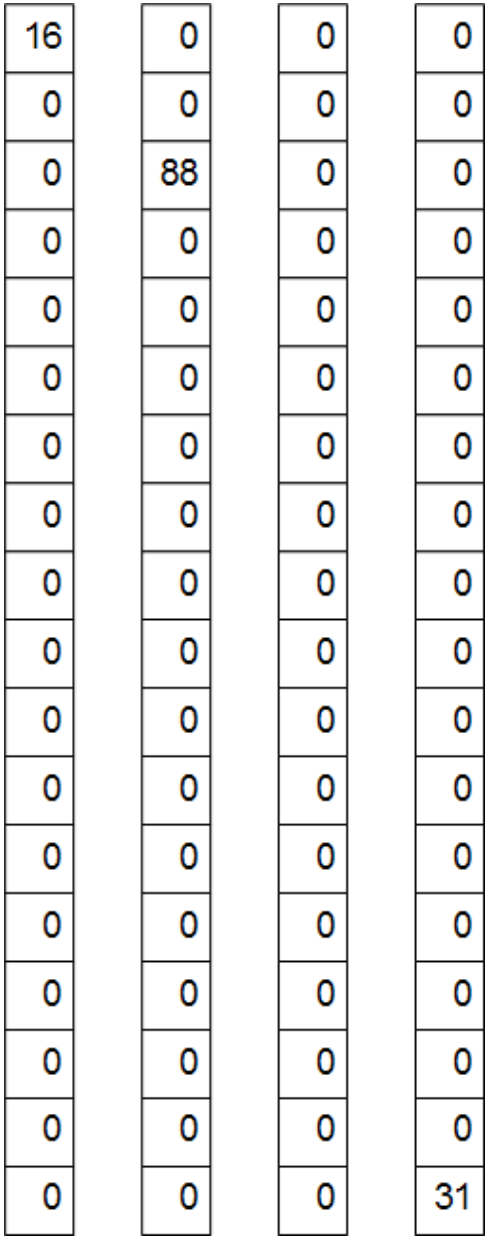
After Transaction 4

- We need the new values to be in stable storage before commit
- After an update of position 1, block 1 needs to be written out to the database before commit can be done and reported to the user



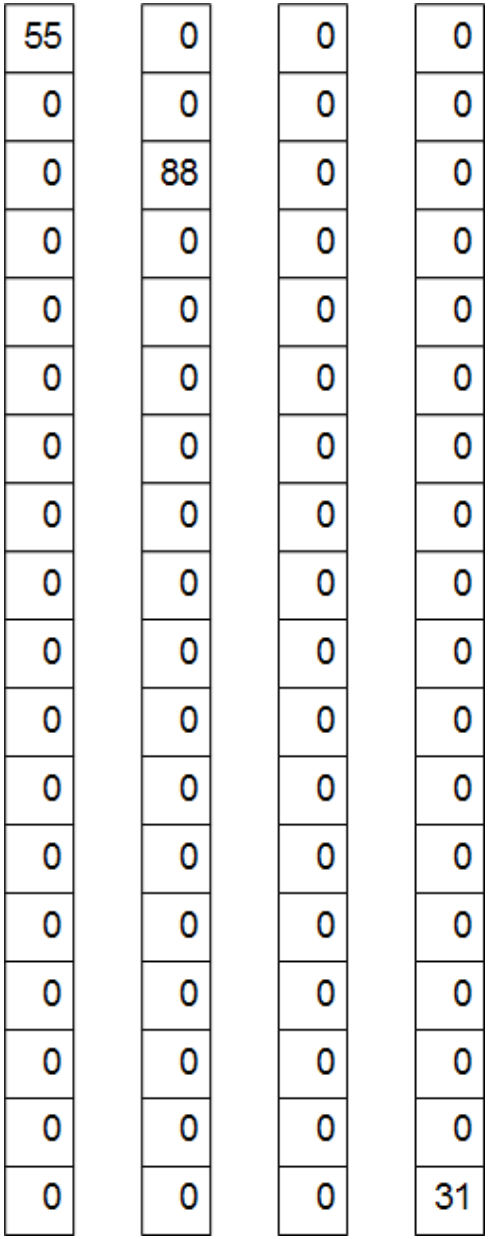
After Transaction 5

- We need the new values to be in stable storage before commit
- After an update of position 21, block 2 needs to be written out to the database before commit can be done and reported to the user



After Transaction 6

- We need the new values to be in stable storage before commit
- After an update of position 1, block 1 needs to be written out to the database before commit can be done and reported to the user

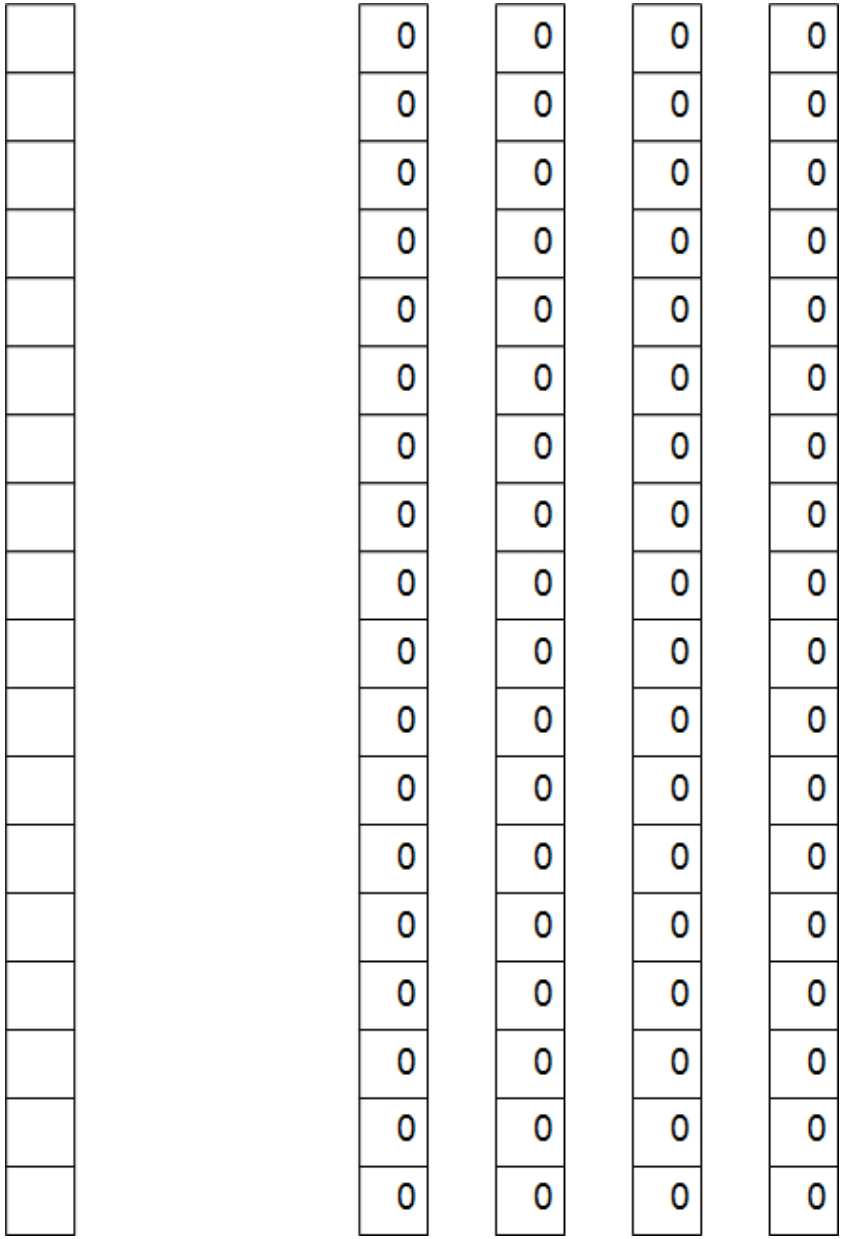


With Recovery Mechanism

- We do not need to write one block to commit each transaction
- We write one full log block to commit “many” transactions

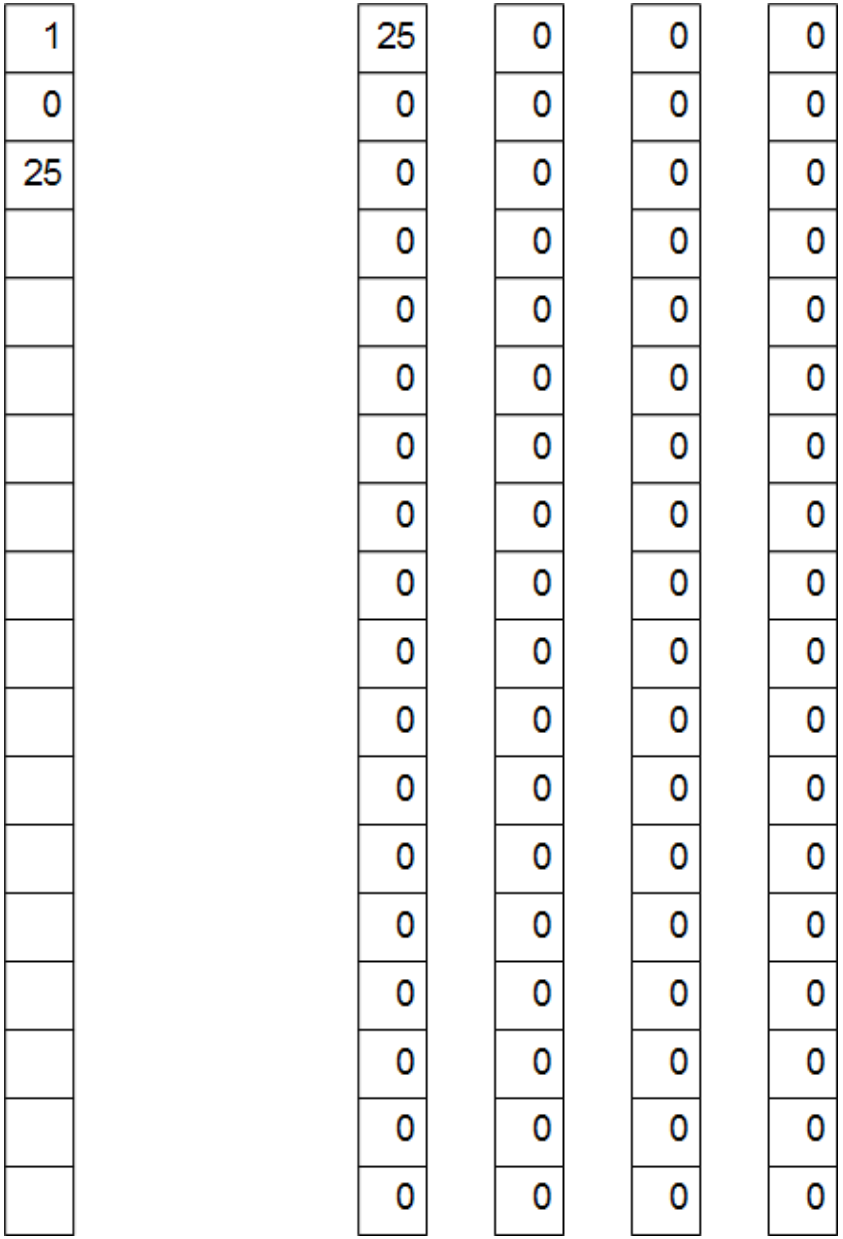
Initial State Of Memory Pool

- Empty log block and initial database in RAM



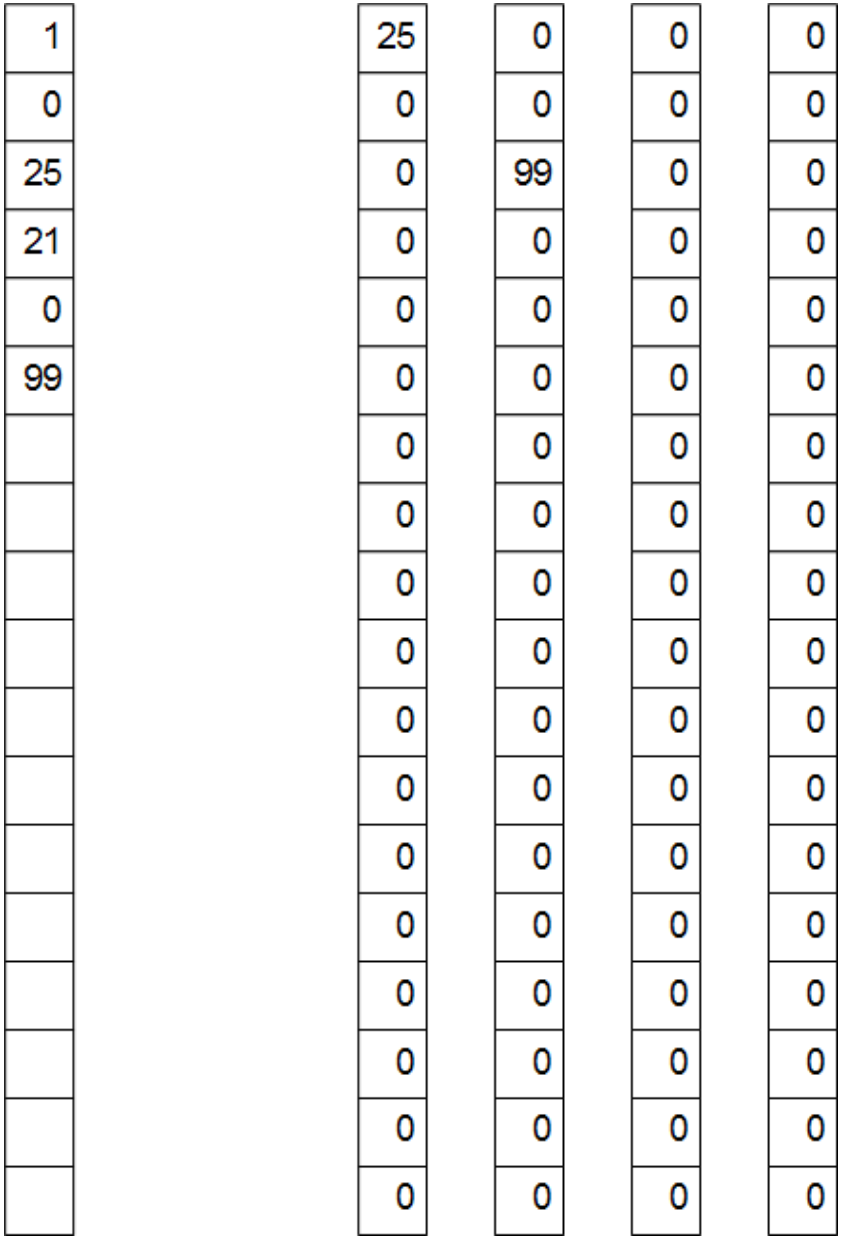
After Transaction 1

- We need the new values to be in stable storage before commit
- After an update of position 1, log block describing this update needs to be written out to the database before commit can be done and reported to the user
- If transactions come frequently, we can wait until log block is full



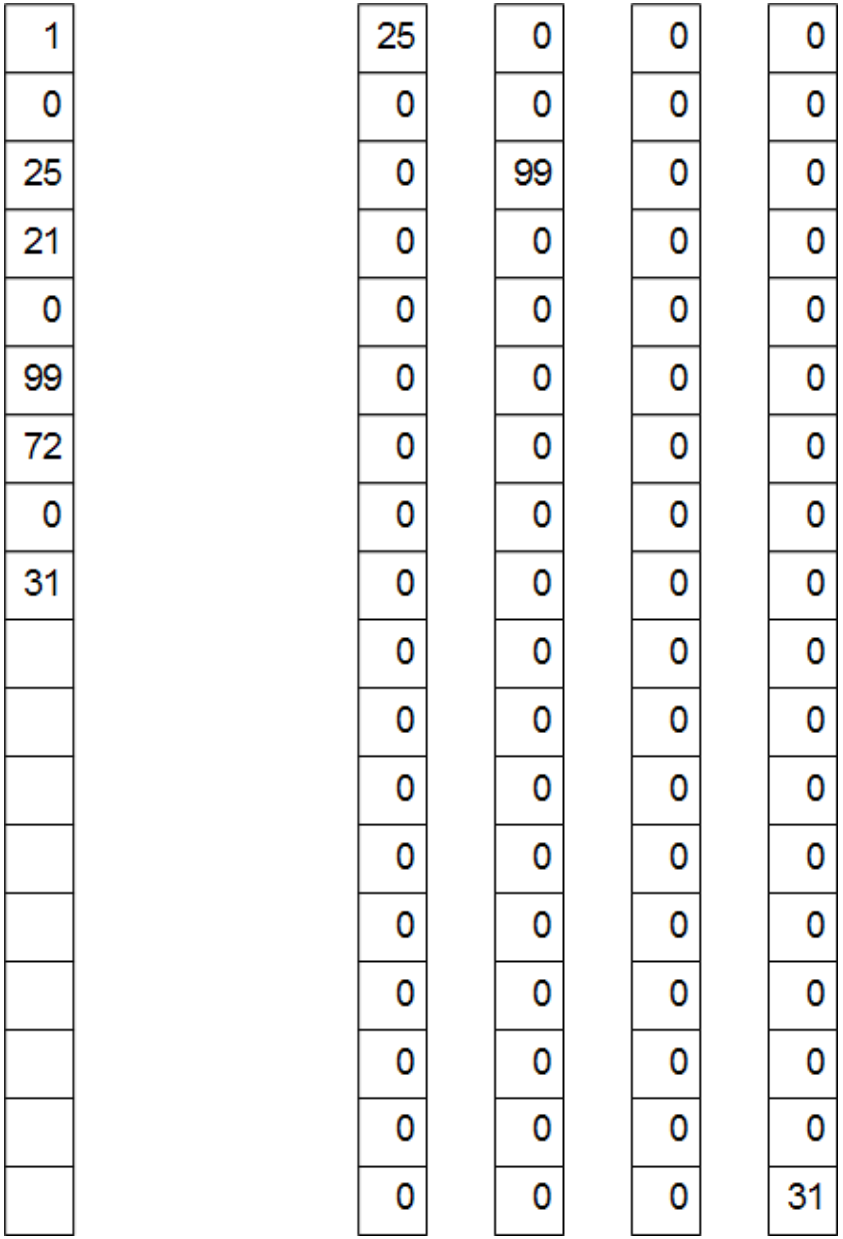
After Transaction 2

- We need the new values to be in stable storage before commit
- After an update of position 21, log block describing this update needs to be written out to the database before commit can be done and reported to the user
- If transactions come frequently, we can wait until log block is full



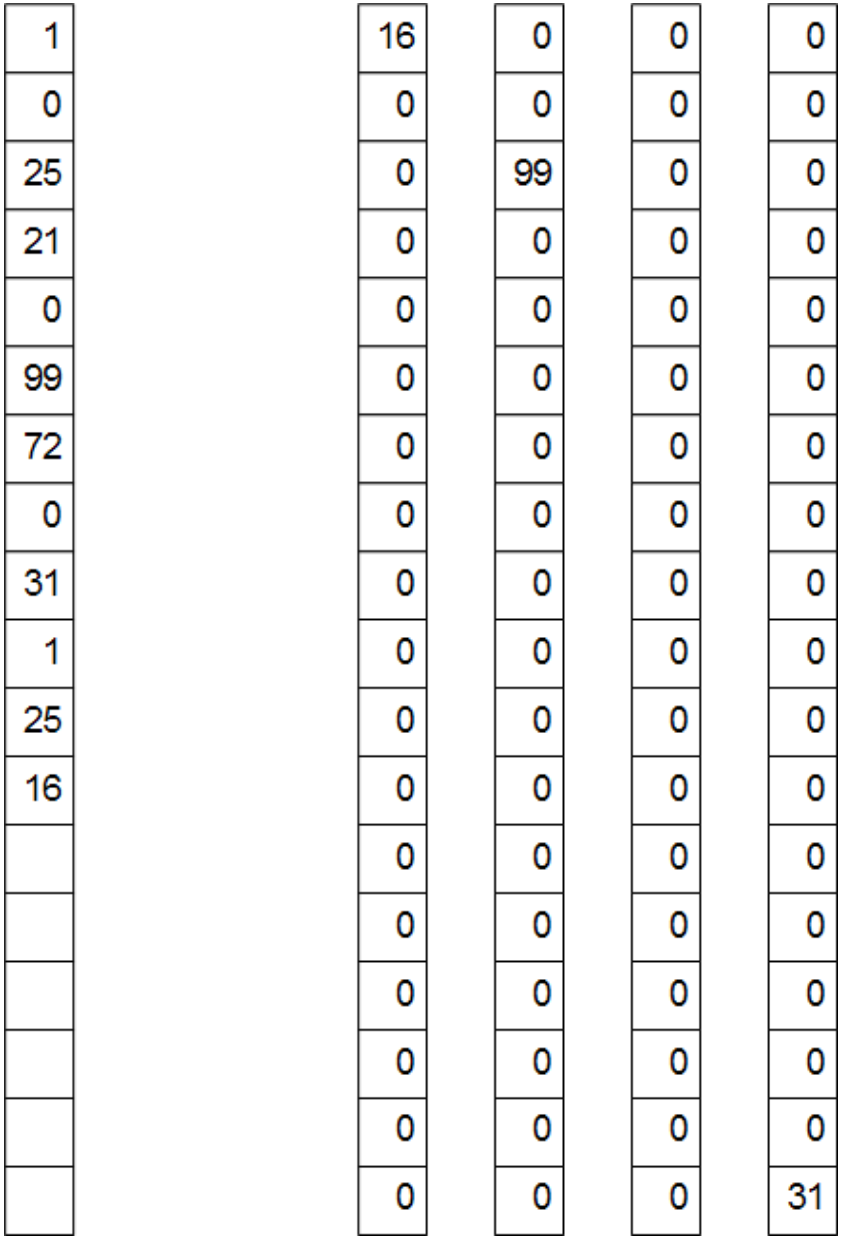
After Transaction 3

- We need the new values to be in stable storage before commit
- After an update of position 72, log block describing this update needs to be written out to the database before commit can be done and reported to the user
- If transactions come frequently, we can wait until log block is full



After Transaction 4

- We need the new values to be in stable storage before commit
- After an update of position 1, log block describing this update needs to be written out to the database before commit can be done and reported to the user
- If transactions come frequently, we can wait until log block is full



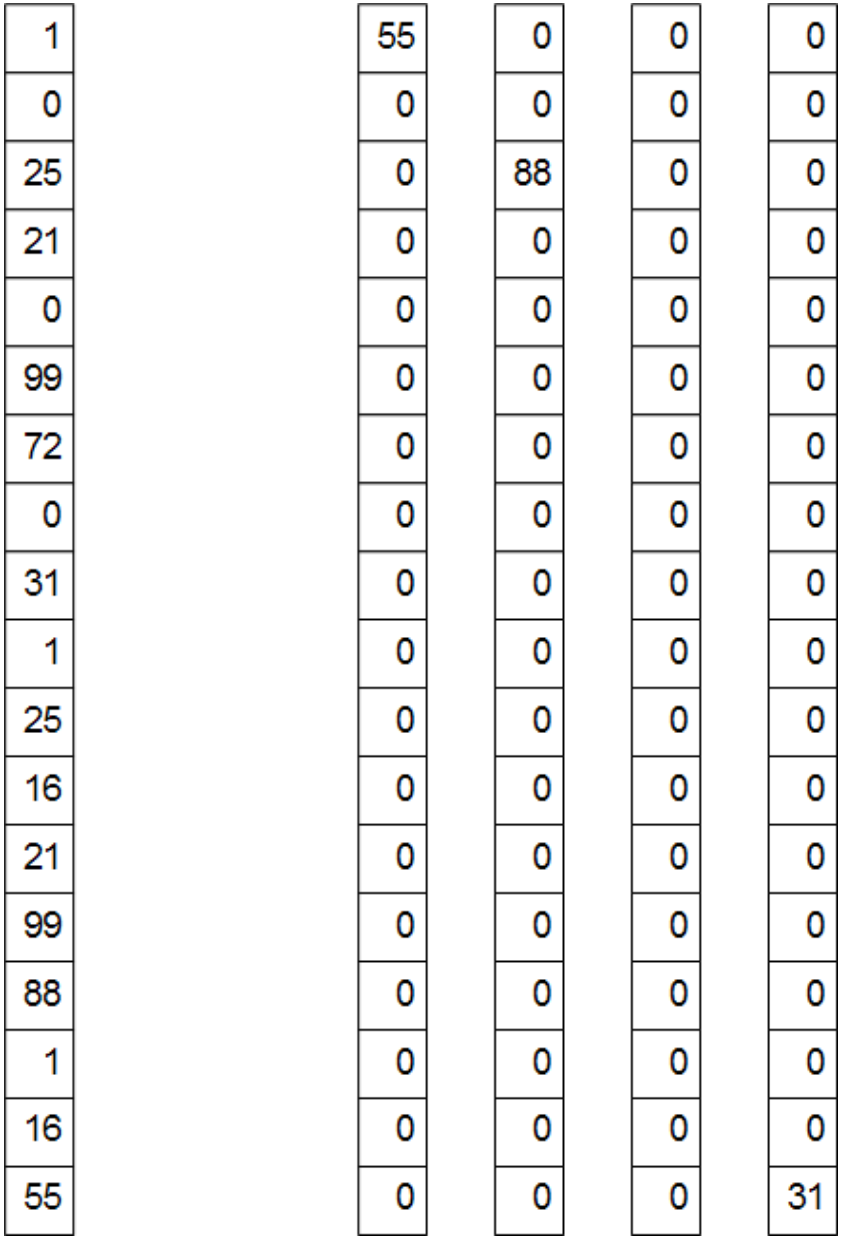
After Transcation 5

- We need the new values to be in stable storage before commit
- After an update of position 21, log block describing this update needs to be written out to the database before commit can be done and reported to the user
- If transactions come frequently, we can wait until log block is full

1	16	0	0	0
0	0	0	0	0
25	0	88	0	0
21	0	0	0	0
0	0	0	0	0
99	0	0	0	0
72	0	0	0	0
0	0	0	0	0
31	0	0	0	0
1	0	0	0	0
25	0	0	0	0
16	0	0	0	0
21	0	0	0	0
99	0	0	0	0
88	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	31

After Transaction 6

- We need the new values to be in stable storage before commit
- After an update of position 1, log block describing this update needs to be written out to the database before commit can be done and reported to the user
- If transactions come frequently, we can wait until log block is full

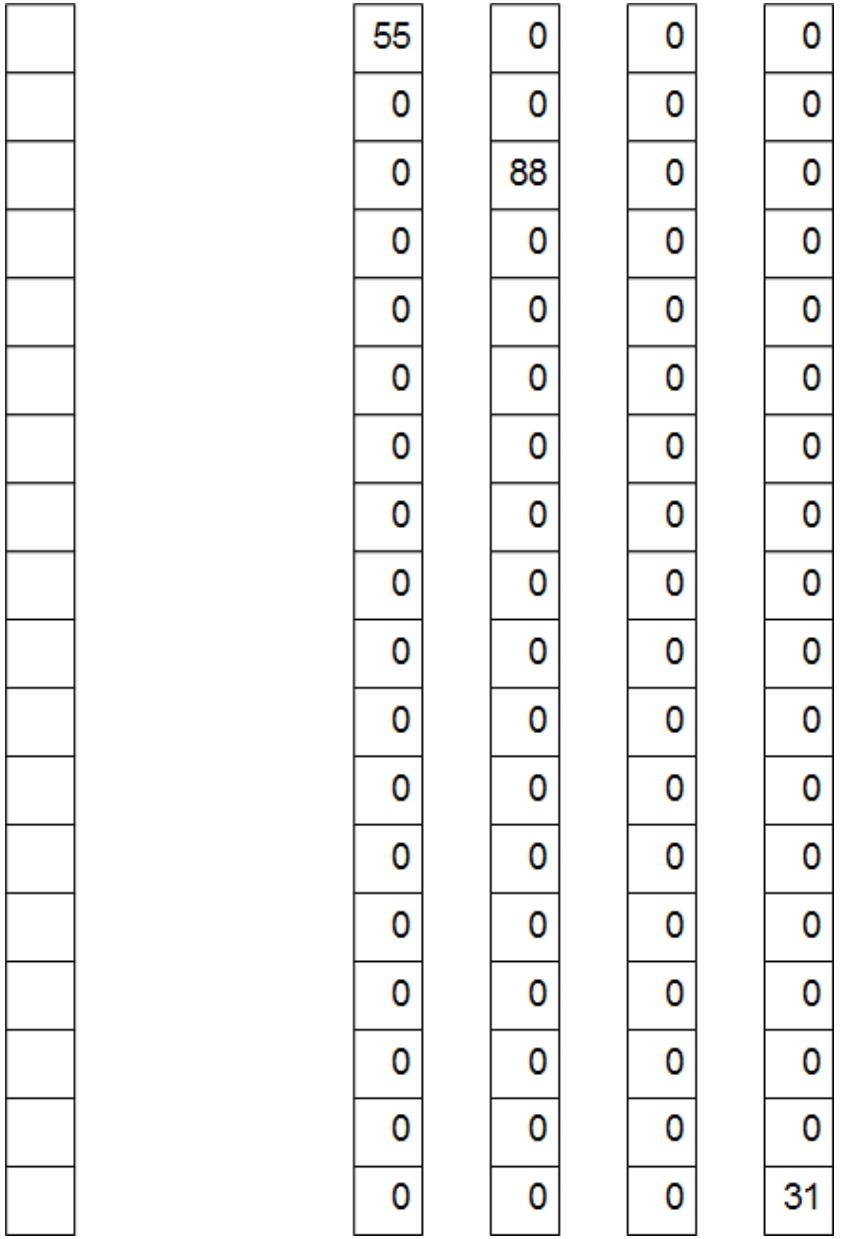


Committing The Transactions

- The block of the log is written out, committing the transactions
- Nothing is done to the four blocks of the database residing in RAM (at least for a while)

New State Of Memory Pool

- Ready for new transactions



Conclusion

- Running with recovery resulted in significant reduction of block writes
- In addition, of course, to supporting recovery well

Key Ideas

- ACID properties
- Need for recovery
- History (schedule)
- Recoverable histories
- Cascading aborts
- Strict histories
- Write ahead log
- Algorithm for recovery
- Checkpointing
- SQL support
- Advanced material: further optimizations