CS525: Advanced Database Organization

Notes 7: Failure Recovery

Yousef M. Elmehdwi

Department of Computer Science

Illinois Institute of Technology

yelmehdwi@iit.edu

November 15, 2018

Slides: adapted from a courses taught by Shun Yan Cheung, Emory University, Hector Garcia-Molina, Stanford, & Jennifer Welch, Texas A&M, & Introduction to Database Systems by ITL Education Solutions Limited

Concurrency and Recovery

- DBMS should enable reestablish correctness of data in the presence of failures
    - System should restore a correct state after failure (recovery)
- DBMS should enable multiple clients to access the database concurrently
    - This can lead to problems with correctness of data because of interleaving of operations from different clients
        - System should ensure correctness (concurrency control)

Failure Modes

- Types of failure modes
  - Erroneous data entry: (e.g., wrong birth date entered)
    - Some erroneous data entry can be checked

      E.g.: value cannot be null or negative etc.
    - Most data entry errors cannot be prevented
  - Media failure (i.e.: disk crash)
    - General technique: Archiving
  - System failure — power outage
    - System failure will cause `inconsistent database states`

Failure Modes: System failure example

- Example: Transaction: `transfer $100 from account A to account B`

```
READ A
A.balance = A.balance - 100
WRITE A
**system fails here**
READ B
B.balance = B.balance + 100
WRITE B
```

- Then `A` would lose his $100 !!!
- This problem is solved by `logging`
- `Transaction` needs to be executed correctly

Modeling consistency of a database

- `Database element`: the unit of data accessed by the database system
  - Abstraction that will come in handy when talking about concurrency control and recovery
- `Database`: a collection of `database elements`
- Note:
  - Different DBMS uses different notion for database element
  - Possible units:
    - A relation
    - A disk block
    - A tuple in a relation

Modeling consistency of a database

- `Database state`: the collection of values of all `database elements` in the database
- `Database state` can be changed by changing one or more of the `database elements` in the database
- A `database state` can be
  - Consistent: satisfy all constraints of the database schema and implicit constraints
  - Inconsistent

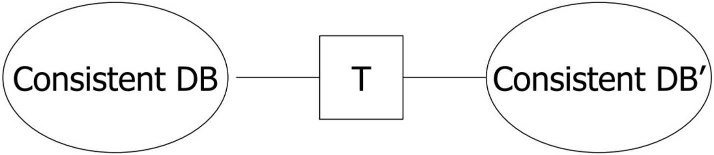
Modeling consistency of a database

- `Transaction`: a sequence of changes to one or more database elements
- Example: Transaction: `transfer $100 from account A to account B`

```
READ A
A.balance = A.balance - 100
WRITE A
READ B
B.balance = B.balance + 100
WRITE B
```

Modeling consistency of a database

- A more precise definition of transaction:
- `Transaction`: a sequence of changes to one or more database elements
- When all changes in a transaction are made to the database state:
  - The resulting database state is a `consistent state` (if the initial state is consistent)



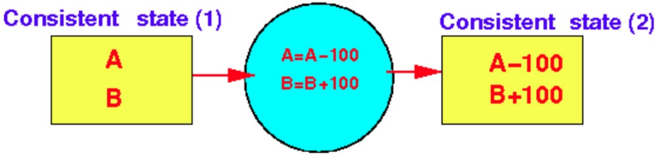
Causes of inconsistent database states

- There are 2 causes of inconsistent database states
    1. System failure
    2. Concurrent execution

How a system failure can result in an inconsistent DB state

- Consider the following transaction `transfer $100 from A to B`

```
READ A
A.balance = A.balance - 100
WRITE A
READ B
B.balance = B.balance + 100
WRITE B
```

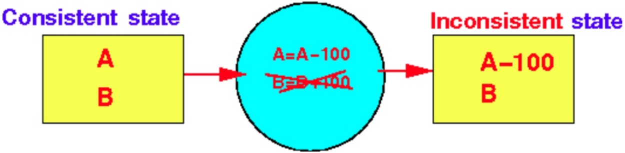


- There are 2 possible consistent states
  - Consistent state 1: `A` `B`
  - Consistent state 2: `A-100` `B+100`

How a system failure can result in an inconsistent DB state

- Consider the database state that result from the following system failure

```
READ A
A.balance = A.balance - 100
WRITE A
**system fails here**
READ B
B.balance = B.balance + 100
WRITE B
```



- The resulting database state is: `Database state = A-100     B`
- Not one of the 2 possible consistent states

How concurrent execution can cause inconsistent states

- Consider the following 2 transactions

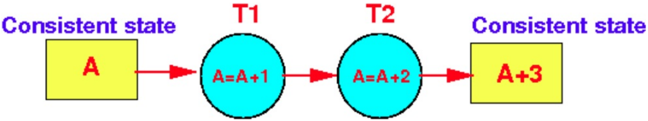
T1= Deposit $1 to A

```
READ A
A = A + 1
WRITE A
```

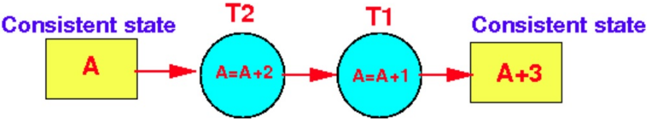
T2= Deposit $2 to A

```
READ A
A = A + 2
WRITE A
```

- The possible consistent database states for executing T1 and T2 are:
  - Case 1: T1 before T2



  - Case 2: T2 before T1



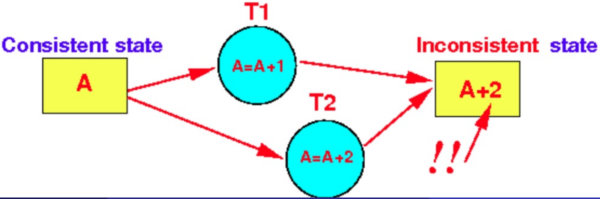
How concurrent execution can cause inconsistent states

- Consider the following concurrent execution of `T1` and `T2`

`T1= Deposit $1 to A`

T1	T2	
		initially: A = 10
READ A		
	READ A	
A = A + 1		(A = 11)
	A = A + 2	(A = 12)
WRITE A		Writes 11 to A
	WRITE A	Writes 12 to A

- Final database state: `A = 12 (= A + 2)`



Correctness “theory” of database transactions

- Assuming that the database is in a `consistent state`
- Then, a transaction will transform the database into a (another) consistent state if:
  - There are no system failures
  - There are no other transactions executing in the database system

Transactions

- `Transaction`: the (smallest) unit of execution of database operations (updates)
- Unit = whole thing, indivisible
- A `transaction` is:
  - executed completely or
  - nothing from the `transaction` is executed

Notation for a transaction

```
begin transaction
   ....
   .... operations performed by the transaction
   .... e.g.: read, compute, write
   ....
commit   // success
```

- or

```
begin transaction
   ....
   .... operations performed by the transaction
   .... e.g.: read, compute, write
   ....
abort   // failure
```

Notation for a `transaction`: Result

- All operations between

```
begin transaction
   ....
   ....   ALL operations executed
   ....
commit
```

  will be executed

- None of the operations between

```
begin transaction
   ....
   ....   NO operations executed
   ....
abort
```

  has been executed

The ACID properties of a transaction

- A `transaction` (must) have the following properties
  - Atomicity
    - Either all operations of the transaction are properly reflected in the database or none are
  - Consistency
    - Execution of a transaction in isolation preserves the consistency of the database.
  - Isolation
    - If two `transactions` are executing concurrently, each `transaction` will see the database as if the `transaction` was executing `sequentially` (in isolation)
  - Durability
    - After a `transaction` completes successfully, the changes it has made to the database persist (permanent), even if there are system failures

Technique to implement transactions

- `Logging`
  - Implements the `atomicity` property
  - Implements the `durability` property
- `Synchronization` (e.g.: locking)
  - Implements the `isolation` property
- The `consistency` property is assumed — otherwise, there is a bug in the `transaction`

Log or log file

- `Log`: an (append only) file containing `log records`
- `Log record`: a record in the `log file` that contains information needed to `undo` and/or `redo` the effects of a `transaction`
- `Log record format`:

  `(TransactionID, Action ,DB element, Value)`

- Creating and writing
  - `Log records` are first created in main memory
  - They are written to disk when convenient
  - Sometimes, the `log records` are forced onto disk

Primitive operations used by Transactions

- Three address spaces
  - The space of disk blocks holding the `database elements`
  - The main memory address space that is managed by the buffer manager
  - The local address space of `transaction`
- Primitive operations describe moving data between address spaces:

```
INPUT(X)    (X is a database element)
OUTPUT(X)   (X is a database element)
READ(X,t)   (X is a DB element,t is a program variable)
WRITE(X,t)  (X is a DB element,t is a program variable)
```

Primitive operations used by Transactions

- `INPUT(X)`
    - Copy the disk block containing the `database element X` to the buffer
- `READ(X,t)`
    - Copy the `database element X` to the `transaction`'s local variable `t`
        - If `database element X` is already in the buffer, then the value is copied to the local variable `t`
        - If `database element X` is not in the buffer, then an `INPUT(X)` is executed and then the value of `X` is copied to the local variable `t`
- `WRITE(X,t)`
    - Copy the value in the `transaction`'s local variable `t` to the `database element X`
        - If `database element X` is already in the buffer, then the value of `t` is copied to `X`
        - If `database element X` is not in the buffer, then an `INPUT(X)` is executed and then the value of `t` is copied to `X` in buffer
- `OUTPUT(X)`
    - Copy the buffer containing the `database element X` to disk

Primitive operations used by Transactions

- Assumption
  - `database element` $\leq$ `1 block`
  - This means, we only need `1 read/write` operation to `read/write` one `database element`
- `READ` and `WRITE` are issued by `transactions`
- `INPUT` and `OUTPUT` are issued by `buffer manager`
- `OUTPUT` can be initiated by `log manager` under certain conditions

Example: using primitive operations

- `Database elements`
  - `A = 8`
  - `B = 8`
- Constraint: `A=B` in all consistent states
- `T1` consists logically of two steps:

  $$\mathtt{A} = \mathtt{A} \times 2$$
  $$\mathtt{B} = \mathtt{B} \times 2$$

- We could express `T1` as a sequence of six relevant steps:

```
READ(A,t);
t =  t * 2;
WRITE(A,t);
READ(B,t);
t =  t * 2;
WRITE(B,t);
```

- In addition, `buffer manager` will eventually execute the `OUTPUT` steps to write these buffers back to disk

Steps of a transaction and it's effect on memory and disk

- Constraint: `A=B` in all consistent states

Action	**t**	**Mem A**	**Mem B**	**Disk A**	**Disk B**
`READ(A,t)`	8	8		8	8
`t = t * 2`	16	8		8	8
`WRITE(A,t)`	16	16		8	8
`READ(B,t)`	8	16	8	8	8
`t = t * 2`	16	16	8	8	8
`WRITE(B,t)`	16	16	16	8	8
`OUTPUT(A)`	16	16	16	16	8
`OUTPUT(B)`	16	16	16	16	16

Transaction manager

- the software sub-system in the DBMS that implements the behavior of a `transaction`
- Functions performed by the transaction manager:
  - Write `log records` to the `log` (file) when a `transaction` performs one of the following operations
    - Start a transaction:

      `Writes: <START T> to the log`
    - Read some data:

      `Writes: <READ ...> to the log`
    - Write (update) some data:

      `Writes: <WRITE ...> to the log`
    - Ends:

      `Writes: <COMMIT T>` or `<ABORT T> to the log`
  - Make sure that `concurrent execution` of `transactions` does not interfere with each other which can result in `inconsistent database state`

Key problem: Unfinished transaction

- Constraint: A=B in all consistent states

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t = t * 2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t = t * 2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
	system fails here				
OUTPUT(B)	16	16	16	16	8

- Need atomicity
  - execute all actions of a `transaction` or none at all

How to restore consistent state after crash?

- Desired state after recovery
  - Changes of committed transactions are reflected on disk
  - Changes of unfinished transactions are not reflected on disk
- After crash we need to
  - `Undo` changes of unfinished `transactions` that have been written to disk
  - `Redo` changes of finished `transactions` that have not been written to disk
- We need to store additional data to be able to `Undo/Redo`

Logging and Recovery

- We need to know
  - Which operations have been executed
  - Which operations are reflected on disk
- Log upfront what is to be done
- Next: We will discuss approaches for logging and how to use them in recovery

Logging: Type of logging techniques

- `Undo logging`
  - The `log file` contains `log records` that enable us to `undo` (roll back) the changes made by an incomplete `transaction`
- `Redo logging`
  - The `log file` contains `log records` that enable us to `redo` (roll forward) the changes made by a completed `transaction`
- `Undo/redo logging`
  - The `log file` contains both `undo` and `redo log records` and enable us to
    - `undo` (roll back) the changes made by an incomplete `transaction`
    - `redo` (roll forward) the changes made by a completed `transaction`

Logging: How is a log file written

- The `log file` consists of 2 parts
  - The older records of the `log file` are stored on disk
  - The newer `log records` of the `log file` are stored in memory
- When the `transaction manager` writes a new `log record`, the new `log record` is appended to the `log records` in the `memory buffer`
- When the `memory buffer` becomes full:
  - The entire buffer is written to disk
  - The buffer content is now a disk block which is appended to the `log file`

Logging: Log writing rules

- Each type of log file has a number of `log file writing rules`
- The `log file writing rules` specifies specific ordering of write operations to the disk
- The specific disk write ordering must be obeyed (or else, the logging technique will not work properly)

Logging: log flush operation

- Due to the `log writing rules` in the logging protocol, we must write the `log records` to disk even when the buffer is not completely full
- The `log flush` operation will force the `log records` in memory to be written to disk
- After a `log flush`, new `log records` can be appended to the log buffer in memory
- When the log buffer is full, we write the full buffer to disk and replace the partially filled data block

Undo logging

- Assume that the `log file` is append-only
  - The `log file` contains every record that has been written (No records has been deleted)
- Later, we will discuss (log) `checkpoint` that will truncate the `log file`
- `Log records` in an `undo log` are (solely) used to `undo` the changes made by a `transaction`

Undo logging: Record types in an undo-log:

- <START T>
  - Indicates that the transaction T has started
- <COMMIT T>
  - Indicates that the transaction T has completed successfully. (No more actions performed by transaction T will follow)
- <ABORT T>
  - Indicates that the transaction T has completed unsuccessfully. (No more actions performed by transaction T will follow)
- <T, X, v>
  - Indicates that the transaction T has updated the database element X.
  - The log record field v: the former value (the value before the update operation) of database element X.
  - The value v can be used to undo the change made by the transaction
  - The record <T, X, v> is generated by a WRITE(X) action by transaction T

Undo logging: Rules for writing an undo log

- Rule U1
  - If a transaction manager performs OUTPUT(X) (to write X to disk):
    - The transaction manager must first
      - Write log record <T, X, v> to disk
    - before writing the new value of X to disk using OUTPUT(X)
  - This ordering of actions will ensure guarantee that we can always undo the change made by OUTPUT(X) using the before value v in the log record

Undo logging: Rules for writing an undo log

- Rule U2
  - If a transaction T write a <COMMIT T> log record to disk
    - The transaction manager must write
      - OUTPUT(X_1) to disk
      - OUTPUT(X_2) to disk
      - ...
      - OUTPUT(X_k) to disk

      on all DB elements that have been updated by the transaction
    - before writing log record <COMMIT T> to disk
  - Because once the <COMMIT T> log record has been recorded on disk, we will not undo the transaction
  - Therefore, the database must have the all the new values to be consistent

Undo logging: Undo Write Rules

Algorithm 1: Undo Write Rules

```
Transaction manager executes an operation
// Undo log write rule U1
if operation = OUTPUT(X) (to disk) then
    FLUSH log (on disk); // This will OUTPUT <T, X, v> to disk
    OUTPUT(X) (on disk);
// Undo log write rule U2
else if operation = <COMMIT T> (to disk) then
    for each DB item D updated by transaction T do
        OUTPUT(D) // Write (new value) to disk
    end
    write <COMMIT T> to log
    FLUSH log (to disk)
else
    Execute operation
end
```

- Recall the `log flush` operation will force the `log records` in memory to be written to disk

Undo logging rules

1. For every action generate undo log record (containing old value)
2. Before `X` is modified on disk, log records pertaining to `X` must be on disk (write ahead logging: WAL)
3. Before commit is flushed to log, all writes of transaction must be reflected on disk

Actions and their log entries

Step	Action	t	M-A	M-B	D-A	D-B	M-log	D-log
1					8	8	<START T>	
2	READ(A,t)	8	8		8	8		
3	t = t * 2	16	8		8	8		
4	WRITE(A,t)	16	16		8	8	<T,A,8>	
5	READ(B,t)	8	16	8	8	8		
6	t = t * 2	16	16	8	8	8		
7	WRITE(B,t)	16	16	16	8	8	<T,B,8>	
8								
9	OUTPUT(A)							

- Notice the `undo write rule 1`, write undo log records (<T,A,8>,<T,B,8>) before writing database elements (OUTPUT(A),OUTPUT(B))

Actions and their log entries

Step	Action	t	M-A	M-B	D-A	D-B	M-log	D-log
1					8	8	<START T>	
2	READ(A,t)	8	8		8	8		
3	t = t * 2	16	8		8	8		
4	WRITE(A,t)	16	16		8	8	<T,A,8>	
5	READ(B,t)	8	16	8	8	8		
6	t = t * 2	16	16	8	8	8		
7	WRITE(B,t)	16	16	16	8	8	<T,B,8>	
8	FLUSH Log							<START T> <T,A,8> <T,B,8>
9	OUTPUT(A)	16	16	16	16	8		
10	OUTPUT(B)	16	16	16	16	16		
11							<COMMIT T>	
12	FLUSH Log							<COMMIT T>

- Notice the `undo write rule 1`, write undo log records (<T,A,8>,<T,B,8>) before writing database elements (OUTPUT(A),OUTPUT(B))
- Notice the `undo write rule 2`, write all database elements (OUTPUT(A),OUTPUT(B)) before writing the <COMMIT T> `log record`

Recovery using Undo logging

- Committed (completed successfully) transaction T:

  - a transaction T where its log record <COMMIT T> is written onto the disk
- Uncommitted transaction:

  - a transaction that does not have a log record <COMMIT T> in the log file
- Therefore, a <COMMIT T> log record (stored on disk) is the proof/evidence that the transaction T is completed
- Recovery manager:

  - the software component in the DBMS that is responsible for restoring the DB to a consistent state after a system failure
- Note: Committed transactions must survive the system failure after the recovery

Recovery using Undo logging

- If you find <COMMIT T> log record in the undo log, then all the data that was updated by transaction T has already been written to disk
- Therefore, we can ignore all undo log records for committed transactions T because we do not want to undo the changes made by committed transactions
- How to perform recovery from a system failure using an undo log
  1. Identify the uncommitted transactions
  2. Undo the actions (write operations) performed these uncommitted transactions

     - ''Roll back'' a transaction = undo-ing the updated made a transaction

Recovery Algorithm for an undo log

Algorithm 2: Recovery Algorithm for an undo log

```
for (every T_i with <START T_i > in Log) do
    if <COMMIT T_i > or <ABORT T_i > in Log then
        Do nothing
    else
        for all <T_i,X,v> in Log do
            // Undo the action
            WRITE(X,v)
            OUTPUT(X)
        end
        // mark the uncommitted transactions as failed
        Write <ABORT T_i >
    end
end
```

- Is this correct?

Recovery Algorithm for an undo log

Algorithm 3: Recovery Algorithm for an undo log

```
// Step 1:  identify the uncommitted transactions
Let S = set of uncommitted transactions in Log
// Step 2:  undo the uncommitted transactions in the reverse order
for (each <T_i, X, v> in Log in reverse order (latest → earliest)) do
    if T_i ∈ S then
        // Update X with the (before) value v
        WRITE(X,v)
        OUTPUT(X)
    end
end
for (each T_i ∈ S) do
    Write <ABORT T_i> to Log
end
Flush Log
```

Examples using an undo log

Step	Action	t	M-A	M-B	D-A	D-B	M-log	D-log
1					8	8	<START T>	
2	READ(A,t)	8	8		8	8		
3	t = t * 2	16	8		8	8		
4	WRITE(A,t)	16	16		8	8	<T,A,8>	
5	READ(B,t)	8	16	8	8	8		
6	t = t * 2	16	16	8	8	8		
7	WRITE(B,t)	16	16	16	8	8	<T,B,8>	
8	FLUSH Log							<START T> <T,A,8> <T,B,8>
9	OUTPUT(A)	16	16	16	16	8		
10	OUTPUT(B)	16	16	16	16	16		
11							<COMMIT T>	
12	FLUSH Log							<COMMIT T>
	System fails here (Crash)							

- Uncommitted transactions: None
- Action: Nothing to do
- Because the system has flushed the updated made by T to disk

Examples using an undo log

Step	Action	t	M-A	M-B	D-A	D-B	M-log	D-log
1					8	8	<START T>	
2	READ(A,t)	8	8		8	8		
3	t = t * 2	16	8		8	8		
4	WRITE(A,t)	16	16		8	8	<T,A,8>	
5	READ(B,t)	8	16	8	8	8		
6	t = t * 2	16	16	8	8	8		
7	WRITE(B,t)	16	16	16	8	8	<T,B,8>	
8	FLUSH Log							<START T> <T,A,8> <T,B,8>
9	OUTPUT(A)	16	16	16	16	8		
10	OUTPUT(B)	16	16	16	16	16		
	System fails here (Crash)							
11							~~<COMMIT T>~~	
12	~~FLUSH Log~~							~~<COMMIT T>~~?

- Uncommitted transactions: T
- Action caused by records in log
  - <T,B,8> $\Rightarrow$ restore B back to 8
  - <T,A,8> $\Rightarrow$ restore A back to 8

Examples using an undo log

Step	Action	t	M-A	M-B	D-A	D-B	M-log	D-log
1					8	8	<START T>	
2	READ(A,t)	8	8		8	8		
3	t = t * 2	16	8		8	8		
4	WRITE(A,t)	16	16		8	8	<T,A,8>	
5	READ(B,t)	8	16	8	8	8		
6	t = t * 2	16	16	8	8	8		
	System fails here (Crash)							
7	~~WRITE(B,t)~~	16	16	~~16~~	8	8	~~<T,B,8>~~	
8	~~FLUSH Log~~							~~<START T>~~ ~~<T,A,8>~~ ~~<T,B,8>~~
9	~~OUTPUT(A)~~	16	16	~~16~~	~~16~~	8		
10	~~OUTPUT(B)~~	16	16	~~16~~	~~16~~	~~16~~		
11							~~<COMMIT T>~~	
12	~~FLUSH Log~~							~~<COMMIT T>~~?

- The log has not been written (empty). There are no log records
- the database elements A and B have not been updated
- Uncommitted transactions: None
- Action caused by records in log None

System crash during recovery of the database

- What happens when the system crashes during a recovery procedure?
- `Idempotent operation`: an operation that produce the same result when the operation is applied any number of times.
- Operations used by a database recovery procedure are only `idempotent operations`
  - We restore the old value back into the `database element`
  - This operation is `idempotent`
- Therefore, if there is a system failure during a recovery procedure, we simply re-apply the log to the database again

Log Checkpointing

- Previously we made the following assumption
  - The log is never truncated
- This assumption simplifies the discuss on recovery
  - We can examine all transactions and determine which ones have committed
- Problem with `append-only log`: Too large
- We need to truncate the `log` from time to time.

Log Checkpointing

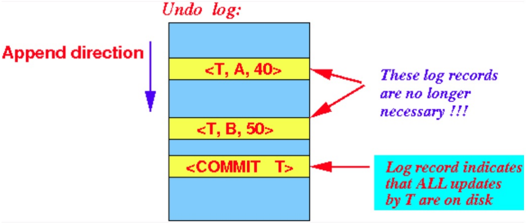
- In log checkpointing, we shorten the log file by "removing" the log records from `completed transactions`
- "remove": `log checkpointing` will remove `log records` logically
  - We will write a `LOG CHECK POINT log record` into the `log file`
  - Some `log records` before this `LOG CHECK POINT record` will not be examined in the `recovery procedure`
- Two ways to delete records from a `log file`
  - Physically
    - The `log records` are actually deleted from the `log file`
  - Logically
    - The `log file` is marked with a special "check point" record
    - Some portion of the `log file` will be discarded (ignored) when we use it in recovery
  - In practical, all log records are kept for the purpose of accounting (Especially in banking transactions)

Log checkpointing in practice

1. Special `check point (log) record` is written into the `log file`
2. Recovery operations will mostly use the portion of the `log file` that is written after the `check point record`

Checkpointing algorithms (for the undo log)

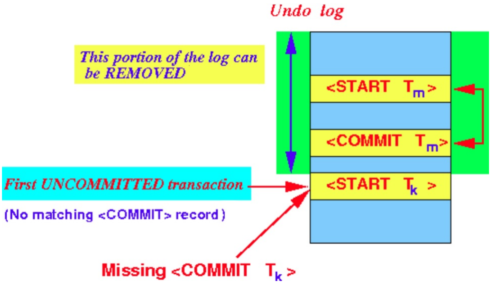
- When you find a <COMMIT T> record for `transaction T`, then the `undo log records` written on behalf of the `transaction T` are now unnecessary



- The log record <COMMIT T> is proof/evidence that `transaction T` has completed successfully

Checkpointing algorithms (for the undo log)

- How to truncate an `undo log`
  - Find the first uncommitted transaction in the (undo) log



  - You can remove everything before this $<$`START` $T_k >$ record, because all `log records` above the first `uncommitted transaction` belongs to a `committed transaction`
- How to apply the `undo log truncation` technique
  - `Quiescent check pointing`
  - `Non-quiescent check pointing`

Quiescent undo log check pointing

- Quiescent: inactive (no transactions are running)
- The quiescent check point algorithm on a undo log
  1. Make the DBMS stop accepting new transactions
  2. Wait until all currently active transactions to commit or abort (and have written a <COMMIT> or <ABORT> log record)
  3. Flush the log to disk
  4. Write <CKPT> (checkpoint) to log. Marks the "useful" boundary
  5. Flush the log
  6. Resume accepting new transactions

Quiescent undo log check pointing: Example

- Currently: T_1 and T_2 are active

  Undo log

  <START T_1>
  <T_1,A,4>
  <START T_2>
  <T_2,B,9>

- Now we want to perform a checkpoint
  - Wait until T_1 and T_2 commit or abort
  - Write <CKPT>
  - Flush log

Quiescent undo log check pointing: Example

- Possible continuation:

  <START T_1>
  <T_1,A,4>
  <START T_2>
  <T_2,B,9>
  <T_2,C,14>
  <T_1,D,19>
  <COMMIT T_1>
  <COMMIT T_2>
  <CKPT> --------- Useful ‘‘boundary’’
  <START T_3>
  <T_3,E,25>
  <T_3,F,30>

  - If we want to truncate an `undo log`, we can remove all log record prior to the <CKPT> record

Recovery procedure with checkpointing

- Key difference
  - We do not have to scan the entire `(undo) log file`
  - The (backwards) scan can stop when we find a <`CKPT`> record

Non-quiescent checkpointing

- Performing checkpointing without stopping the DMBS from accepting new transactions
- The Non-quiescent check point algorithm on a undo log
  1. Write a start checkpoint log record
     - <START CKPT($T_1, T_2, \ldots, T_k$)> to log file where $T_1, T_2, \ldots, T_k$ are the currently active transactions
  2. Flush-Log (optional)
  3. Wait until all of $T_1, T_2, \ldots, T_k$ to commit or abort (DBMS can accept new transactions)
  4. When all $T_1, T_2, \ldots, T_k$ have completed
     - Write <END CKPT> to log file
  5. Flush-Log (essential to keep the log file short)

Non-quiescent undo log check pointing: Example

- Currently: T_1 and T_2 are active

  `Undo log`

  <START T_1>
  <T_1,A,4>
  <START T_2>
  <T_2,B,9>

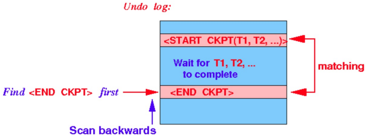
- Now we want to perform a checkpoint
  1. Write <START CKPT(T_1,T_2)>
  2. Flush log
  3. Wait until T_1 and T_2 commit or abort
  4. Write <END CKPT>
  5. Flush log

Non-quiescent undo log check pointing: Example

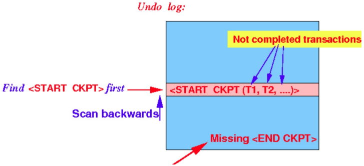
- Possible continuation:

<START T_1>
<T_1,A,4>
<START T_2>
<T_2,B,9>
<START CKPT(T_1,T_2)> --------- Flush Log
<T_2,C,14>
<START T_3> === New transactions can start
<T_1,D,19>
<COMMIT T_1>
<T_3,E,25>
<COMMIT T_2>
<END CKPT> --------- Flush Log
<T_3,F,30>

- When scanning the `log file` backwards, you can find one of 2 `possibilities`
  1. You find a <`END CKPT`> `log record` first.
     - This is the case when the (last) checkpoint operation has completed successfully

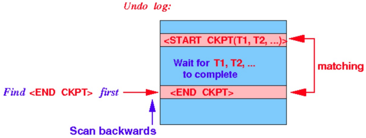


  2. You find a <`START CKPT`(T_1,T_2,...)> `log record` first.
     - This is the case when the system has crashed during the last checkpoint operation

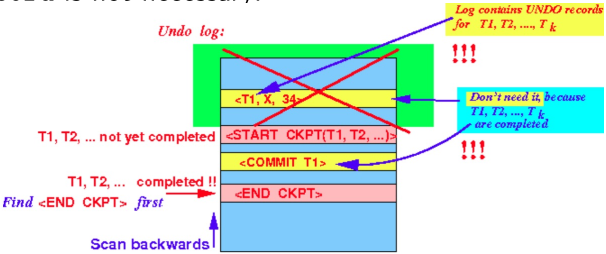


Recovering from case 1

- You find a <END CKPT> `log record` first.

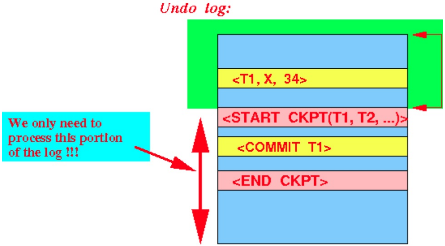


- We know (for sure) that all of the transactions $\mathtt{T}_1, \mathtt{T}_2, \ldots, \mathtt{T}_k$ have completed
- Therefore, the portion of the `undo log` before the <START CKPT...> `log record` is not necessary:

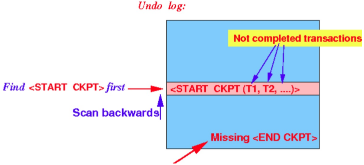


Recovering from case 1: How to recover:

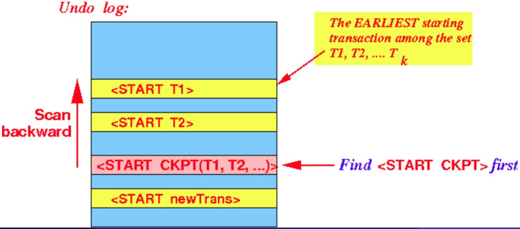
- We must undo all uncommitted transactions that has started after $<$START CKPT(T$_1$,T$_2$,...)$>$ record



- We found a <`START CKPT`(T_1,T_2,...)> `record` first.

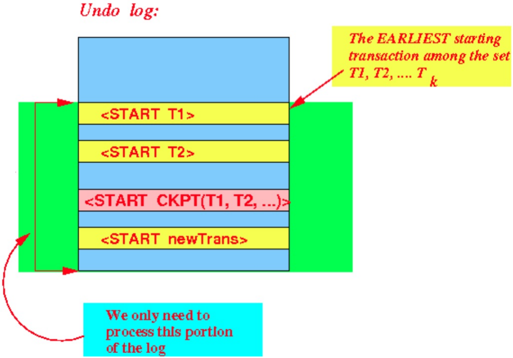


  - We know (for sure) that the only `transactions` that have not yet completed at the start of the check point are $T_1, T_2, \ldots, T_k$
- How far back in the `log file` do we need to look to find all `incomplete transactions`
  - Scan `log file` backward and locate the earliest <`START` T_i> `record`



Recovering from case 2: How to recover

- Portion of the `undo log` that is needed (contain the information) for recovery



Question

- How far back do we need to scan to find all <START CKPT> records?
  - The furthest back we need to scan the log file to find all of the <START T_1>, <START T_2>,..., <START T_k> records is the previous <START CKPT> record

Short-coming of undo logging

- Undo logging requires that data be written to disk immediately after a transaction finishes, perhaps increasing the number of disk I/O's that need to be performed

Comparing the undo and redo logging methods

- `Undo logging`
  - Is designed to `undo` (cancel) the effect of `incomplete transactions`
    - The recovery procedure will undo the effect of `uncommitted transactions`
  - The recovery procedure will ignore the `committed transactions`
- `Redo logging`
  - Is designed to `redo` (repeat) the effect of `complete transactions`
    - The recovery procedure will `redo` (repeat) the effect of `committed transactions`
  - The recovery procedure will ignore the `uncommitted transactions`

Redo Logging: Record Types in an Redo Log:

- <START T>
  - Indicates that the transaction T has started
- <COMMIT T>
  - Indicates that the transaction T has completed successfully. (No more actions performed by transaction T will follow)
- <ABORT T>
  - Indicates that the transaction T has completed unsuccessfully. (No more actions performed by transaction T will follow)
- <T, X, v>
  - Indicates that the transaction T has updated the database element X.
  - The log record field v: the after value (the value after the update operation) of database element X.
  - The value v can be used to redo the change made by the transaction
  - The record <T, X, v> is generated by a WRITE(X) action by transaction T

The Redo log update rules

- The transaction manager can only perform OUTPUT() operations for committed transactions
  - If the transaction manager performs OUTPUT(X)
    - The transaction manager must first write all log records of the transaction T to disk
  - i.e., All log records pertaining to the modification of database element X must be recorded on disk first. This include the <COMMIT> record
  - This will ensure that the effect of a committed transaction can be repeated
- Before we do OUTPUT(X_i), we must flush all log records of transaction T

Redo update rule expressed as algorithm

Algorithm 4: Redo Log Write Rule

```
Transaction manager executes an operation
// Redo log write rule
// Only update DB elements modified by committed transactions
if (operation = OUTPUT(X)) then
    // DB element X was updated by transaction T)
    if (T's state == COMMITTED) then
        // Write all log records to disk including the log records
           belonging to T. We made sure that all updates by T can be
           (re)done
        FLUSH log
        OUTPUT(X) // (When) we make one of the updates of T to disk
    else
        // Don't write data updated by uncommitted transaction to disk
        return
    end
else
    perform operation
end
```

- Recall the `log flush` operation will force the `log records` in memory to be written to disk

Example: using a redo log

Step	Action	t	M-A	M-B	D-A	D-B	M-log	D-log
1					8	8	<START T>	
2	READ(A,t)	8	8		8	8		
3	t = t * 2	16	8		8	8		
4	WRITE(A,t)	16	16		8	8	<T,A,16>	
5	READ(B,t)	8	16	8	8	8		
6	t = t * 2	16	16	8	8	8		
7	WRITE(B,t)	16	16	16	8	8	<T,B,16>	
8							<COMMIT T>	
9								
10	OUTPUT(A)	16	16	16	8	8		

- Before we update any database element (A or B), we must flush the log records
- Flushing the redo log will enable us to avoid possible partial updates of the DB elements (A or B, but not both) which will cause an inconsistent DB state

Example: using a redo log

Step	Action	t	M-A	M-B	D-A	D-B	M-log	D-log
1					8	8	<START T>	
2	READ(A,t)	8	8		8	8		
3	t = t * 2	16	8		8	8		
4	WRITE(A,t)	16	16		8	8	<T,A,16>	
5	READ(B,t)	8	16	8	8	8		
6	t = t * 2	16	16	8	8	8		
7	WRITE(B,t)	16	16	16	8	8	<T,B,16>	
8							<COMMIT T>	
9	FLUSH Log							<START T> <T,A,16> <T,B,16> <COMMIT T>
10	OUTPUT(A)	16	16	16	16	8		
11	OUTPUT(B)	16	16	16	16	16		

- Before we update any database element (A or B), we must flush the log records
- Flushing the redo log will enable us to avoid possible partial updates of the DB elements (A or B, but not both) which will cause an inconsistent DB state

Observation about the redo logging method

- Even when we see a <COMMIT T> record in the (redo) log file on disk, we cannot be certain that the updates (effects) of transaction T have been written to disk

Step	Action	t	M-A	M-B	D-A	D-B	M-log	D-log
1					8	8	<START T>	
2	READ(A,t)	8	8		8	8		
3	t = t * 2	16	8		8	8		
4	WRITE(A,t)	16	16		8	8	<T,A,16>	
5	READ(B,t)	8	16	8	8	8		
6	t = t * 2	16	16	8	8	8		
7	WRITE(B,t)	16	16	16	8	8	<T,B,16>	
8							<COMMIT T>	
9	FLUSH Log							<START T> <T,A,16> <T,B,16> <COMMIT T>
10								
11								

- The (redo) log on disk contains a <COMMIT T> record
- The DB elements A and B on disk has not been updated (still in the memory buffer)

Recovery Algorithm for an redo log

Algorithm 5: Recovery Algorithm for an undo log

// Step 1: identify the committed transactions

Let S = set of committed transactions in Log

// Step 2: redo the committed transactions in the forward order

for *(each $<T_i, X, v>$ in Log in forwards)* **do**

  if $T_i \in S$ **then**

    Update X with the (after) value v // Redo the change

  end

end

// Step 3: mark the uncommitted transactions as failed

for *(each T that is uncommitted)* **do**

  Write <ABORT T_i > to Log

end

Flush Log

Example: using a redo log

Step	Action	t	M-A	M-B	D-A	D-B	M-log	D-log
1					8	8	<START T>	
2	READ(A,t)	8	8		8	8		
3	t = t * 2	16	8		8	8		
4	WRITE(A,t)	16	16		8	8	<T,A,16>	
5	READ(B,t)	8	16	8	8	8		
6	t = t * 2	16	16	8	8	8		
7	WRITE(B,t)	16	16	16	8	8	<T,B,16>	
8							<COMMIT T>	
9	FLUSH Log							<START T> <T,A,16> <T,B,16> <COMMIT T>
10	OUTPUT(A)	16	16	16	16	8		
11	OUTPUT(B)	16	16	16	16	16		
	System fails here (Crash)							

- Committed transactions: T
- Action caused by records in log
  - Update DB element A to (new value) 16
  - Update DB element B to (new value) 16

Non-quiescent checkpointing for redo log

- Recall: `checkpointing a log`
  - In `log checkpointing`, we want to shorten the log by (logically) removing the `log records` of the completed `transactions`
    - Committed transactions, and
    - Aborted transactions
- Important facts about a redo log
  1. `Aborted (unsuccessful) transaction` will never perform any `OUTPUT()` operation in `redo logging`
     - `OUTPUT()` is performed after <`COMMIT T`> is written to disk
     - Therefore, we can ignore (remove) `log records` of the `aborted transactions` in the `redo log`
  2. The `OUTPUT()` operations `committed transactions` in `redo logging` can be delayed

Non-quiescent checkpointing for redo log

- We can discard (delete) the `log records` belonging to `uncommitted transactions`
- In order to remove the `redo records` (logically) belonging to the `committed transactions`
  - We must first incorporate all updates in `log records` <`T,X,v`> made by the `committed transactions` to disk
  - Because we cannot redo the updates after we (logically) remove these `log records`

Non-quiescent checkpointing for a redo log

- The `Non-quiescent check point` algorithm on a `redo log`
  1. Write a `start checkpoint log record`
     - <START CKPT($T_1, T_2, \ldots, T_k$)> to log file where $T_1, T_2, \ldots, T_k$ are the `currently active (uncommitted) transactions`
  2. Flush-Log
  3. Incorporate updates from `committed transactions`
     - Output all `database elements` that were updated by `committed transactions` to disk
  4. Write <END CKPT> to log file
  5. Flush-Log

Checkpointing a redo log: Example

- Currently: T_1 and T_2 are active

  Undo log

  <START T_1>
  <T_1,A,5>
  <START T_2>
  <COMMIT T_1>
  <T_2,B,10>

- Now we want to perform a checkpoint
  - Write <START CKPT(T_2)>

    Undo log
    <START T_1>
    <T_1,A,5>
    <START T_2>
    <COMMIT T_1>
    <T_2,B,10>
    <START CKPT(T_2)>

Checkpointing a redo log: Example

- Write the DB element A of committed transaction (T_1) to disk

Undo log

<START T_1>

<T_1,A,5> --- update by a committed transaction

<START T_2>

<COMMIT T_1>

<T_2,B,10>

<START CKPT(T_2)>

...

--- Write (A,5) to disk

...

Checkpointing a redo log: Example

- Write <END CKPT> log record

```
Undo log
<START T1>
<T1,A,5>          --- update by a committed transaction
<START T2>
<COMMIT T1>
<T2,B,10>
<START CKPT(T2)>
     ...
                  --- Write (A,5) to disk
     ...
<END CKPT>
```

- Flush log

Checkpointing a redo log: Example

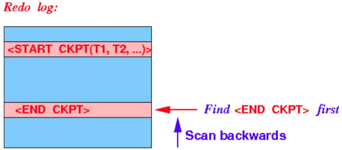
- Possible continuation:

```
Undo log
<START T1>
<T1,A,5>
<START T2>
<COMMIT T1>          --- T1 done
<T2,B,10>
<START CKPT(T2)>
<T2,C,15>            <---+ Between here:
<START T3>               | <OUTPUT(A,5)>
<T3,D,20>            <---+
<END CKPT>           --- Flush Log
<COMMIT T2>          --- T2 done
<COMMIT T3>          --- T3 done
```

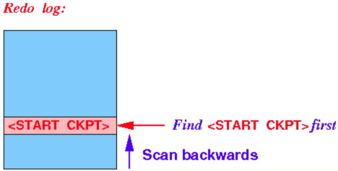
  - (<OUTPUT( )> for T_2 and T_3 happens later)

Recovery using non-quiescent checkpointing

- When scanning the `log file` backwards, you can find one of 2 `possibilities`
  1. You find a <`END CKPT`> `log record` first.
     - This is the case when the (last) checkpoint operation has completed successfully

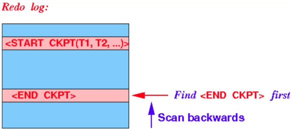


  2. You find a <`START CKPT`(T_1,T_2,...)> `log record` first.
     - This is the case when the system has crashed during the last checkpoint operation

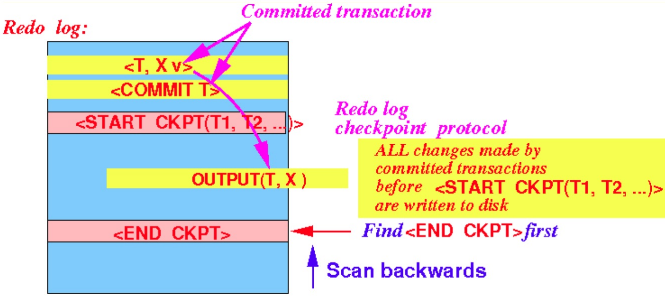


Recovering from case 1

- Given that we find a <`END CKPT`> `log record` first.

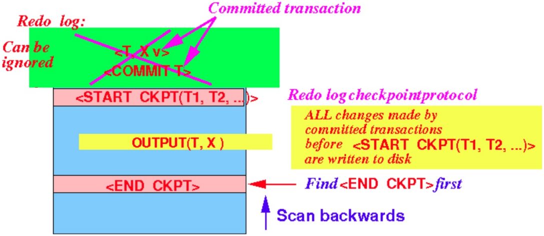


- By the checkpointing algorithm, all changes made by a `committed transaction T` prior to <`START CKPT(.)`> have been written to disk

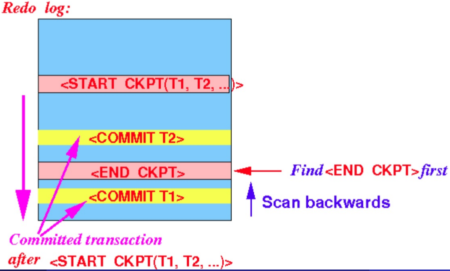


Recovering from case 1

- Therefore, in the recovery, we do not need to redo the changes made by committed transactions in this portion of the redo log



- We (still) have to redo the updates made by transactions that are committed after the <START CKPT(.)> record

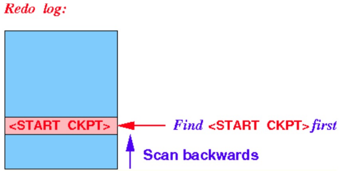


Summary: Recovering from case 1

- Find the earliest <START T_i> log record where T_i is in the check point transaction list
- Redo the updates made by the transactions that have committed after the <START CKPT> log record

Recovering from case 2

- We find a <START CKPT(T_1,T_2,...)> record first.



  - During the checkpointing, the check point protocol will write the changes made by committed transaction on disk
  - Without the <END CKPT> record, we do not know which updates have been written to disk (and which have not)
  - Worst case scenario, no updates made by committed transactions have been written to disk
- All we can do is go further back into the redo log, and use the recovery procedure for case 1

Key drawbacks

- Undo logging
    - cannot bring backup database copies up to date
- Redo logging
    - need to keep all modified blocks in memory until commit

The undo/redo log

- Format of a undo/redo log record
  - Undo/redo log record:
    - `<T,X,v,w>`
      `T` transaction ID
      `X`: DB element
      `v`: before (old) value
      `w`: after (new) value
- We write both the before value and the after value of an update operation in the log
- The `undo/redo log` is a combination of two logging approaches undo logging and redo logging

Undo/Redo update rule expressed as algorithm

Algorithm 6: Undo/Redo Log Write Rule

```
Transaction manager executes an operation
// Undo/redo log write rule
// First write <T,X,v,w> to disk then write X to disk
if (operation = OUTPUT(X)) then
    FLUSH log // Write all log records to disk including the <T,X,v,w>
        log record
    // We made sure that OUTPUT(X) can be undone
    OUTPUT(X)
else
    perform operation
end
```

- There is only `1 rule`, so `undo/redo logging` in more flexible

Non-quiescent checkpointing for a undo/redo log

- The `Non-quiescent check point` algorithm on a `undo/redo log`
  1. Write a `start checkpoint log record`
     - <`START CKPT`(T_1,T_2,...,T_k)> to log file where T_1,T_2,...,T_k are the `currently active (uncommitted) transactions`
  2. Flush-Log
  3. Write all `database elements` that were updated by **ALL** `transactions` that are still in memory buffers
  4. Write <`END CKPT`> to log file
  5. Flush-Log

Checkpointing a undo/redo log: Example

- Currently: T_1 and T_2 are active

  Undo log

  <START T_1>
  <T_1,A,4,5>
  <START T_2>
  <COMMIT T_1>
  <T_2,B,9,10>

- Now we want to perform a checkpoint
  - Write <START CKPT(T_2)>. (Do not include T_1 because T_1 has committed)
  - Write the DB element A and B to disk.
    - Even when T_1 has committed, the data written by T_1 may not have been written to disk
  - Write <END CKPT> log record
  - Flush log

Checkpointing a undo/redo log: Example

- Possible continuation:

```
Undo log
<START T1>
<T1,A,4,5>
<START T2>
<COMMIT T1>         --- T1 done
<T2,B,9,10>
<START CKPT(T2)>
<T2,C,14,15>       <---+ Between here:
<START T3>             | <OUTPUT(A)> and <OUTPUT(B)>
<T3,D,19,20>       <---+
<END CKPT>          --- Flush Log
<COMMIT T2>         --- T2 done
<COMMIT T3>         --- T3 done
```

  - (<OUTPUT( )> for T_2 and T_3 happens later)

Comment

- The checkpointing procedure will output all updated `DB elements` before writing the <END CKPT> `record`
- This will simplify the recovery of the `completed transactions`
- Example
  - If during a recovery we find that T_2 has committed (along with an <END CKPT> `record`), then we know for sure that:
    - All updates made by T_2 before the <START CKPT(...)> `record` has been written
    - To redo the actions by a `committed transaction`, we can start `redo-ing` the logged action from the <START CKPT(...)> `record`

Recovering using non-quiescent checkpointing

- Recovery algorithm (in general) in a `undo/redo log`
  - We need to `redo` the `committed transactions` and
  - We need to `undo` the `uncommitted transactions`

Recovering from case 1

- How to recover the committed transactions
  1. Scan up to the <START CKPT(T_1,T_2,...,T_k)> record
     - Identify all the committed transactions
  2. Redo all changes made by the committed transactions starting at the check point log record
- How to recover the uncommitted transactions
  1. Scan up to the earliest <START T_X> record where
     - T_X is one of the transactions in the check point record <START CKPT(T_1,T_2,...)>
     - T_X is an uncommitted transaction
  2. We must redo all actions for the uncommitted transactions after <START T_X> record

Recovering from case 2

- Recovering from case 2 (just like the redo log)

Summary

- Consistency of data
- One source of problems: failures
    - Logging
    - Redundancy
- Another source of problems: Data Sharing (Next)