Other kinds of logging

Overview over this video

In this video, we will look at other kinds of logging (redo and undo/redo) and how to get atomicity and durability from logging

Redo Logging

Logs activities with the goal of *restoring* committed transactions (ignores incomplete transactions).

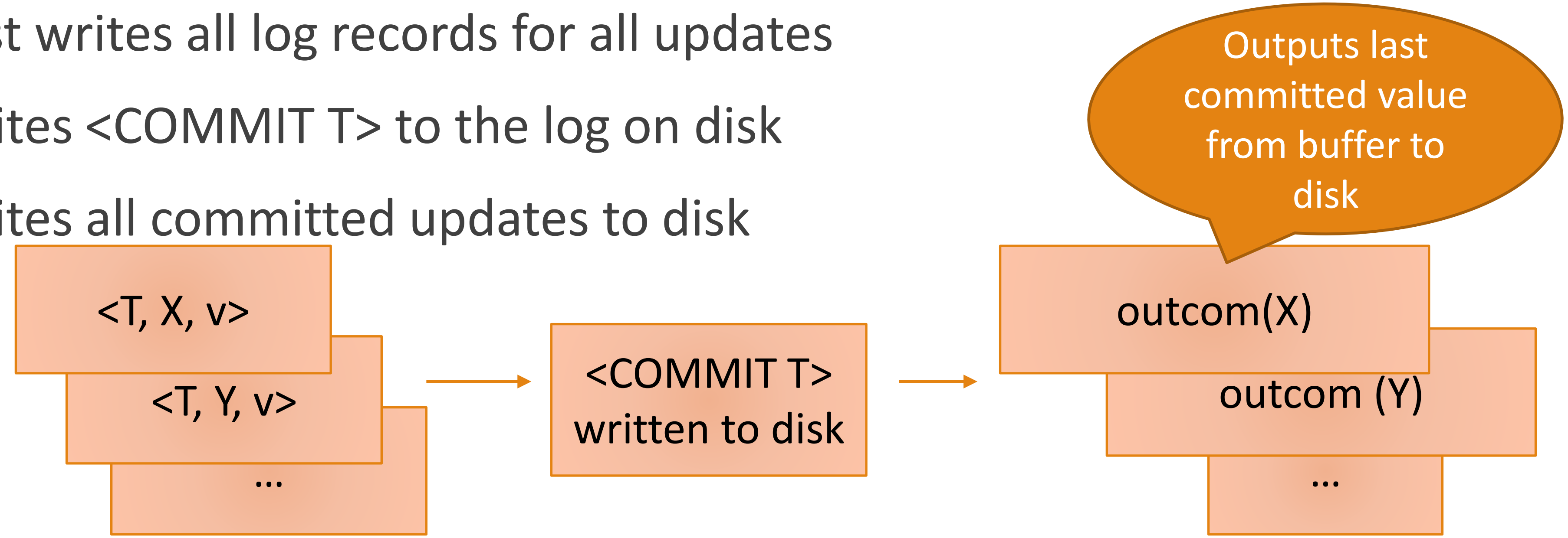
Log records:

- Same as before, but…
- New meaning of **<T, X, v>**: "Transaction T has updated the value of database item X & the new value of X is v."
  - Direct response to **write(X)**
  - Haven't changed X on disk yet!

Have to modify the logging procedure…

Redo Logging: Procedure

1. T first writes all log records for all updates
2. T writes <COMMIT T> to the log on disk
3. T writes all committed updates to disk

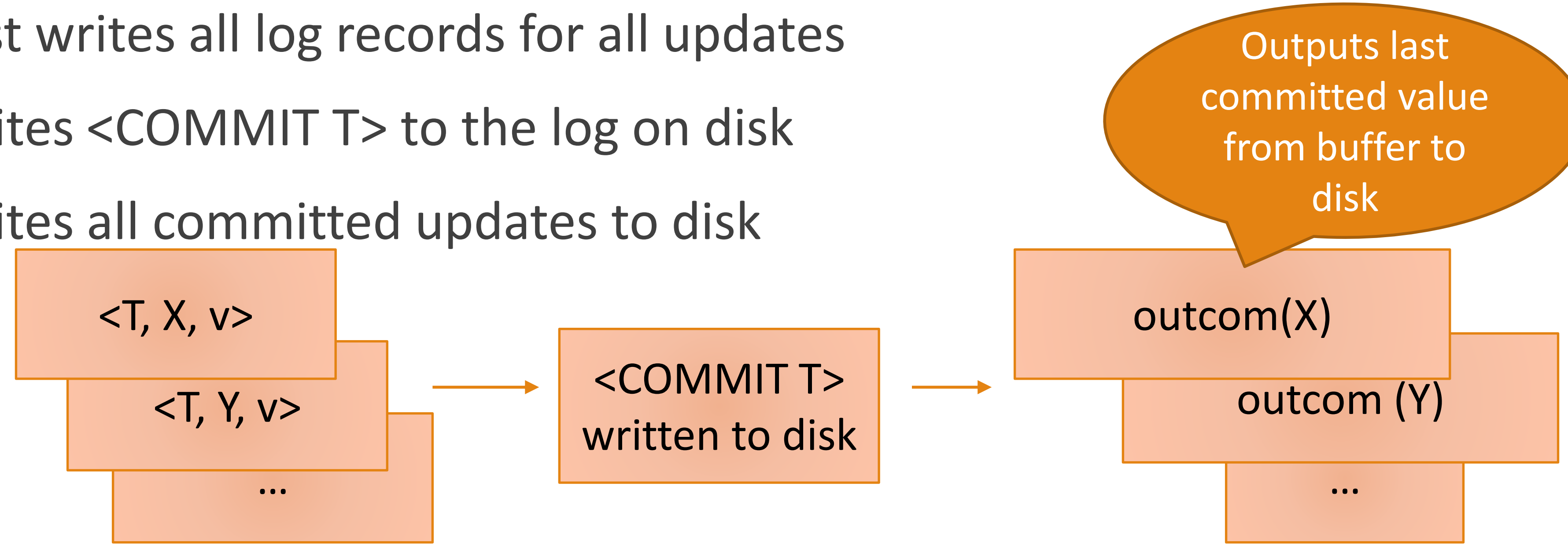


Example

		Local		Buffer		Database			
Time	**Transaction**	**X**	**Y**	**X**	**Y**	**X**	**Y**	**Log (buffer)**	**Log (disk)**
0						1	10	**<START T>**	
1	read(X)	**1**		**1**		1	10		
2	X := X*2	**2**		1		1	10		
3	write(X)	2		**2**		1	10	**<T, X, 2>**	
4	read(Y)	2	**10**	2	**10**	1	10		
5	Y := Y*2	2	**20**	2	10	1	10		
6	write(Y)	2	20	2	**20**	1	10	**<T, Y, 20>**	
7								**<COMMIT T>**	
8	**flush_log**								
9	**outcom(X)**	2	20	2	20	**2**	10		
10	**outcom(Y)**	2	20	2	20	2	**20**		

Redo Logging: Procedure

1. T first writes all log records for all updates
2. T writes <COMMIT T> to the log on disk
3. T writes all committed updates to disk



Fundamental property of redo logs:

- <COMMIT T> occurs in log → log contains complete information on T
- <COMMIT T> doesn't occur in log → T hasn't written anything to disk

Recovery With Redo Logs

Essentially: reverse of undo logging

Procedure:

- Identify all the transactions with a **COMMIT** log record.
- Traverse the log from first to the last item.
- If we see **<T, X, v>** and T has a COMMIT log record, then change the value of X on disk to v.
- For each *incomplete* transaction T, write **<ABORT T>** into the log on disk.

Example with Redo Logging

Time	Transaction T_1	Transaction T_2
1	read(X)	
2	X := X * 2	
3	write(X)	
4		read(X)
5	read(Y)	
6		X := X * 3
7		write(X)
8	Y := X + Y	
9	write(Y)	

X = 1
Y = 2

How does redo logging work on this schedule?

- Which **log entries** are written to buffer/disk & when?
- Which **other operations** must be executed & when?

Time	Transaction T_1	Transaction T_2	Log (buffer)	Log (disk)
0			<START T_1>	
1	read(X)			
2	X := X * 2			
3	write(X)		<T_1, X, 2>	
4			<START T_2>	
5		read(X)		
6	read(Y)			
7		X := X * 3		
8		write(X)	<T_2, X, 6>	
9	Y := X + Y			
10	write(Y)		<T_1, Y, 4>	
11			<COMMIT T_1>	
12	**flush_log**			
13	**outcom(X)**			
14	**outcom(Y)**			
15			<COMMIT T_2>	
16		**flush_log**		
17		**outcom(X)**		
18				
19				

X = 1
Y = 2

Undo/Redo Logging

Good properties of undo logging and redo logging

Log records:

- Same as before, but replace **<T, X, v>**
- **<T, X, v, w>**: “Transaction T has updated the value of database item X, and the **old**/**new** value of X is v/w.”

Procedure:

- Write all log records for all updates to database items first
- Then write updates to disk
- <COMMIT T> can be written to disk before or after all changes have been written to disk

Recovery needs to process log in both directions

Example with Undo/Redo Logging

Time	Transaction T_1	Transaction T_2
1	read(X)	
2	X := X * 2	
3	write(X)	
4		read(X)
5	read(Y)	
6		X := X * 3
7		write(X)
8	Y := X + Y	
9	write(Y)	

X = 1
Y = 2

How does undo/redo logging work on this schedule?

- Which **log entries** are written to buffer/disk & when?
- Which **other operations** must be executed & when?

Time	Transaction T_1	Transaction T_2	Local T_1		Local T_2		Buffer		Disk		Buffer log
			X	Y	X	Y	X	Y	X	Y	
0									1	2	<START T_1>
1	read(X)		1				1		1	2	
2	X := X * 2		2				1		1	2	
3	write(X)		2				2		1	2	<T_1, X, 1, 2>
4			2				2		1	2	<START T_2>
5		read(X)	2		2		2		1	2	
6	read(Y)		2	2	2		2	2	1	2	
7		X := X * 3	2	2	6		2	2	1	2	
8		write(X)	2	2	6		6	2	1	2	<T_2, X, 2, 6>
9	Y := X + Y		2	4	6		6	2	1	2	
10	write(Y)		2	4	6		6	4	1	2	<T_1, Y, 2, 4>
11			2	4	6		6	4	1	2	<COMMIT T_1>
12	**flush_log**		2	4	6		6	4	1	2	
13	**output(X)**		2	4	6		6	4	6	2	
14	**output(Y)**		2	4	6		6	4	6	4	
15			2	4	6		6	4	6	4	<COMMIT T_2>
16		**flush_log**	2	4	6		6	4	6	4	
17		**output(X)**	2	4	6		6	4	6	4	

Why are DBMS using Undo/Redo?

Undo without Redo

Undo essentially ensures Atomicity

Can ensure durability using Force

- **Force** the writing of updates to disk before commit
- (**No Force** is not to require this)
- Force is expensive in disk operations

Redo without Undo

Redo essentially ensures Durability

Can ensure atomicity using No Steal

- **No Steal** means that uncommitted data may not overwrite committed data on disk
- (**Steal** is not to require this)
- **No Steal** is expensive to ensure

Ensuring atomicity and durability

Could ensure Atomicity and Durability without log using No Steal/Force

- Very hard and expensive to ensure
- (Must write every change to disk made by the transaction while performing the commit statement!)

In practice:

- Want Steal/No Force (cheapest in time) → Use Undo/Redo

Summary

We covered Redo logging and Undo/Redo logging

- i.e. basically the kind of logging you should do to be able to do redos or undos and redos

Also covered why DBMS uses Undo/Redo logging (even if it is the more complex option)