Atomic Actions

Atomic actions

Atomic action: an action that either is executed in full or has no effects at all

- Atomic actions in distributed systems:

  - an action is generally executed at more than one node

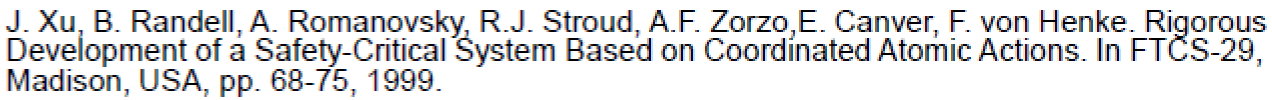
  - nodes must cooperate to guarantee that

  - either the execution of the action completes successfully at each node or the execution of the action has no effects

- The designer can associate fault tolerance mechanisms with the underlying atomic actions:

  - limiting the extent of error propagation when faults occur and

  - localizing the subsequent error recovery



J. Xu, B. Randell, A. Romanovsky, R.J. Stroud, A.F. Zorzo,E. Canver, F. von Henke. Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions. In FTCS-29, Madison, USA, pp. 68-75, 1999.

An example: Transactions in databases

- Transaction: a sequence of changes to data that move the data base from a consistent state to another consistent state.

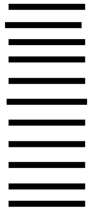
- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items

- Transactions must be atomic:

  all changes are executed successfully or data are not updated

Transactions in databases

Transaction T



T can access data at distinct nodes : distributed transaction

Transactions in databases

COMMIT
Termination with success of the transaction
All operations are executed and changes to the database are persistent

Transfer of 50 from account A to account B

```
t: begin transaction
                    UPDATE account
                    SET balance=balance + 500
                      WHERE account_number=45;

                    UPDATE account
                    SET balance=balance - 500
                    WHERE account_number=35;
      commit
     end transaction
```

Transactions in databases

ABORT or ROLLBACK
abort of the transaction
None operation is exected

Transfer of 50 from account A to account B. Abort if balance of A less than 500.

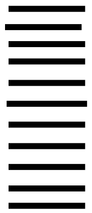
```
t: begin transaction
        UPDATE account
        SET balance=balance + 500   WHERE account_number=45;

        UPDATE account
        SET balance=balance – 500    WHERE account_number=35;

        SELECT balance INTO V FROM account WHERE account_number=35;
        if V >= 0 then commit else abort;
end transaction
```

Transactions in databases and failures

Transaction T



1) A failure before the termination of the transaction, results into a rollback (abort) of the transaction

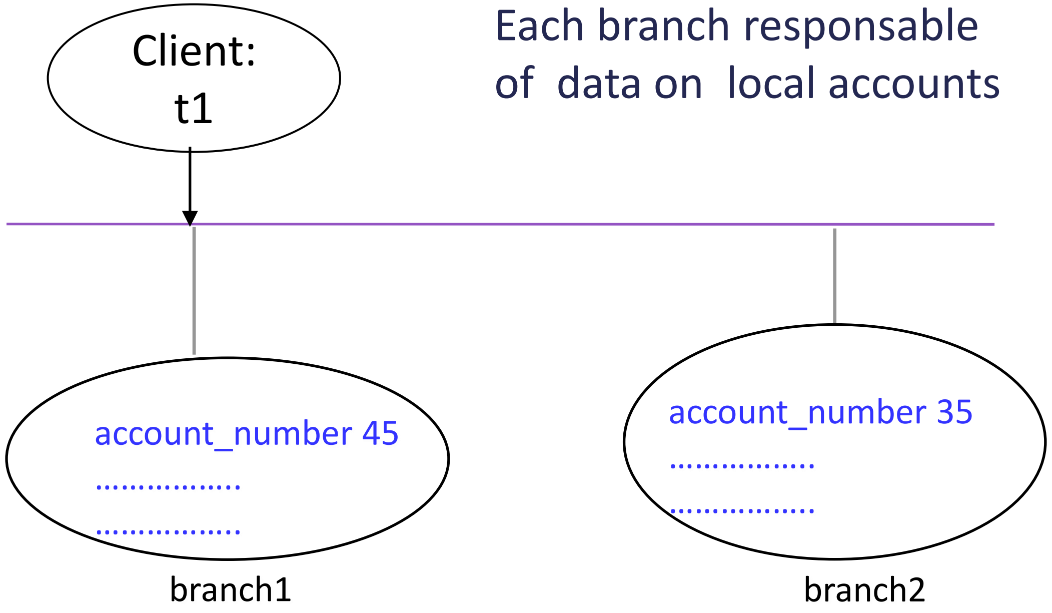
2) A failure after the termination with success (commit) of the transaction must have no consequences

Banking application

Account =(account_name, branch_name, balance)

t1: distributed transaction (access data at different sites)

```
t1: begin transaction
        UPDATE account
        SET balance=balance + 500
          WHERE account_number=45;
        UPDATE account
        SET balance=balance - 500
        WHERE account_number=35;
    commit
   end transaction
```



Each branch responsable of data on local accounts

t1

```
t11: UPDATE account
       SET balance=balance + 500
       WHERE account_number=45;
              site1
```

```
t12:UPDATE account
       SET balance=balance - 500
       WHERE account number=35;
              site2
```

Atomicity requirement

- **Atomicity requirement**
  - if the transaction fails after the update of 45 and before the update of 35, money will be “lost” leading to an inconsistent database state
  - the system should ensure that updates of a partially executed transaction are not reflected in the database

A main issue: atomicity in case of **failures of various kinds, such as hardware failures and system crashes**

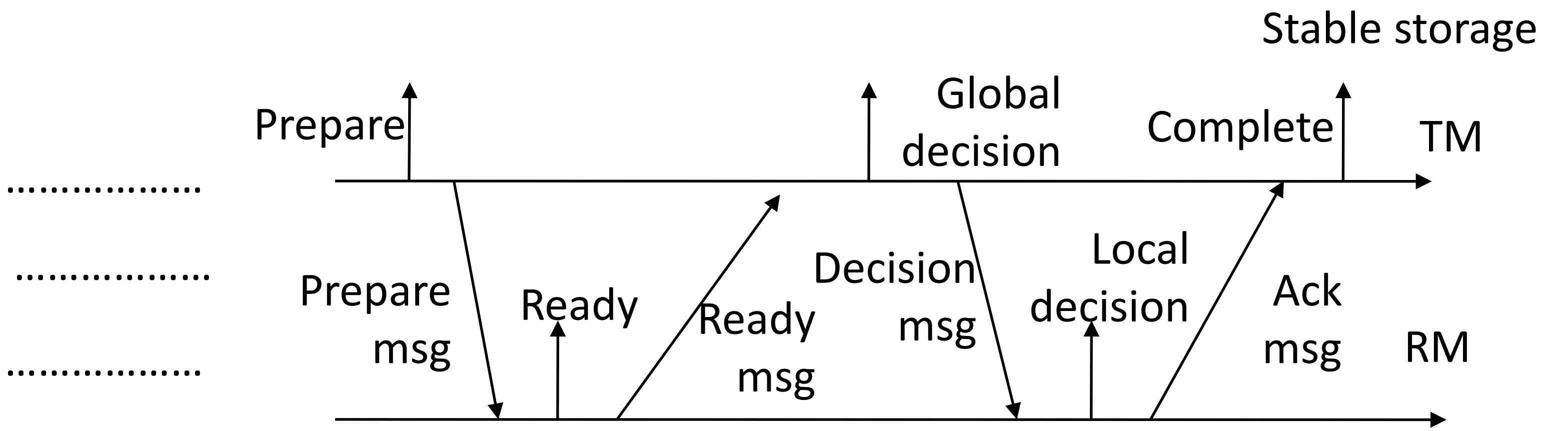
- Atomicity of a transaction:

Commit protocol + Log in stable storage + Recovery algorithm

A programmer assumes atomicity of transactions

Two-phase commit protocol

- One transaction manager TM
- Many resource managers RM
- Log file (persistent memory)
- Time-out



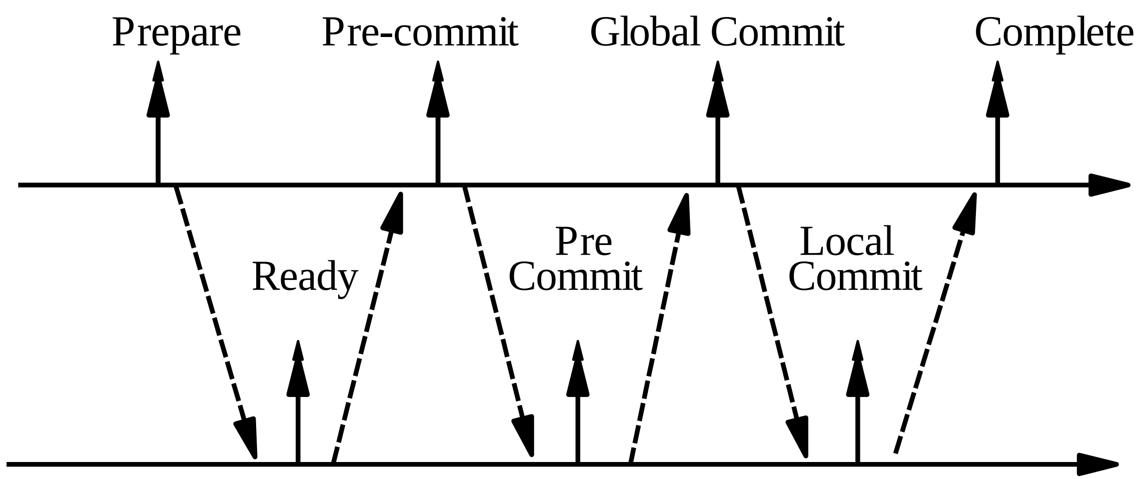
Tolerates: loss of messages

crash of nodes

Uncertain period:

if the transaction manager crash, a participant with Ready in its log cannot terminate the transaction

Three-phase commit



Precommit phase is added. Assume a permanent crash of the coordinator.
A participant can substitute the coordinator to terminate the transaction.

A participant assumes the role of coordinator and decides:

- Global Abort, if the last record in the log Ready
- Global Commit, if the last record in the log is Precommit

Recovery and Atomicity

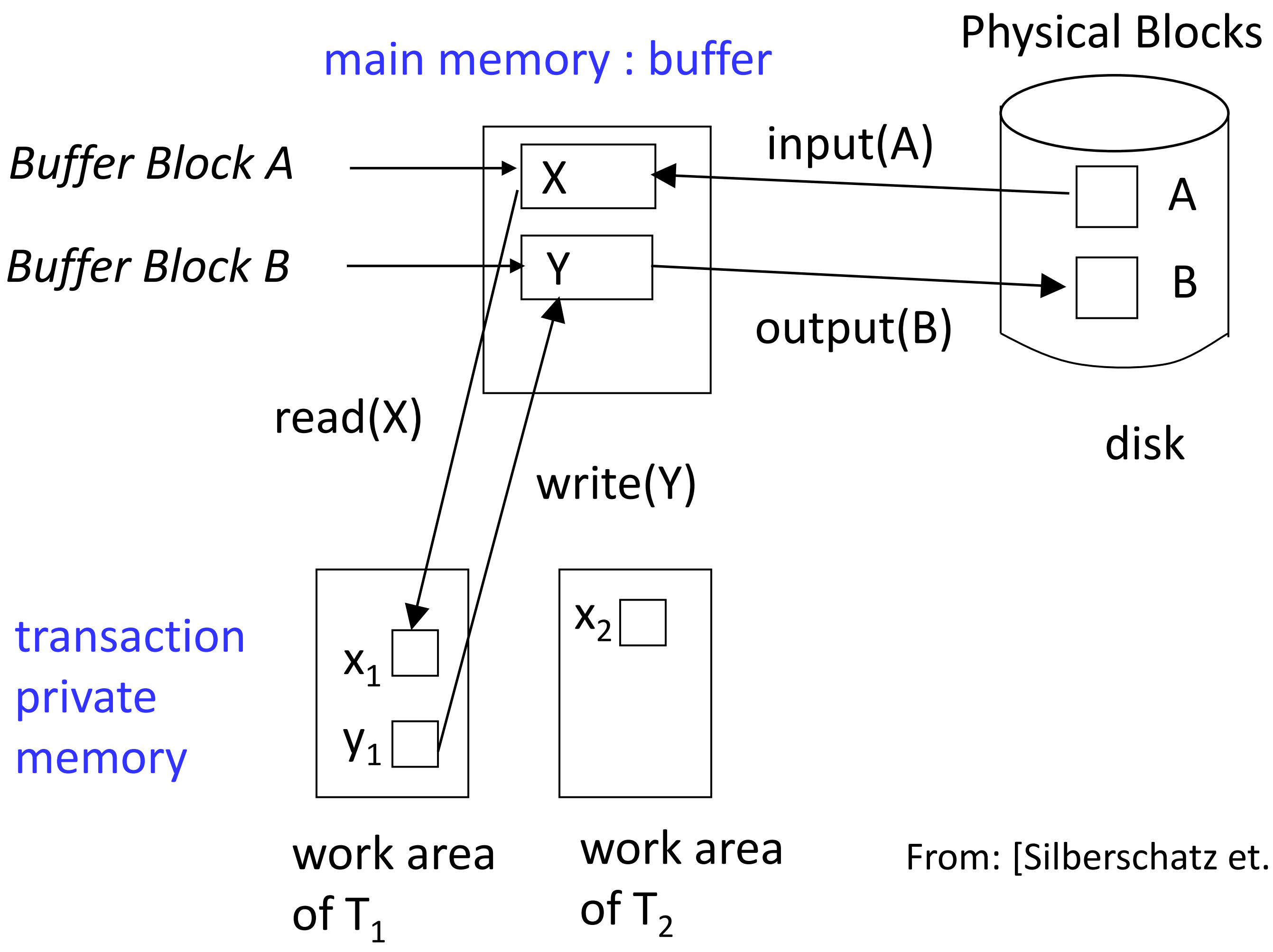
Physical blocks: blocks residing on the disk.
Buffer blocks: blocks residing temporarily in main memory

Block movements between disk and main memory through the following operations:
- **input**(*B*) transfers the physical block *B* to main memory.
- output(*B*) transfers the buffer block *B* to the disk

Transactions
- Each transaction *Ti* has its private work-area in which local copies of all data items accessed and updated by it are kept.
-perform **read**(*X*) while accessing *X* for the first time;
-executes **write**(*X*) after last access of X.

System can perform the **output** operation later.
Let B_X denote block containing *X*.
output(B_X) need not immediately follow **write**(*X*)

Data Access



From: [Silberschatz et. al,2005]

Recovery and Atomicity

- Several output operations may be required for a transaction
- A transaction can be aborted after one of these modifications have been made permanent (transfer of block to disk)
- A transaction can be committed and a failure of the system can occur before all the modifications of the transaction are made permanent
- To ensure atomicity despite failures, we first output information describing the modifications to a Log file in stable storage without modifying the database itself

Log-based recovery

Log file

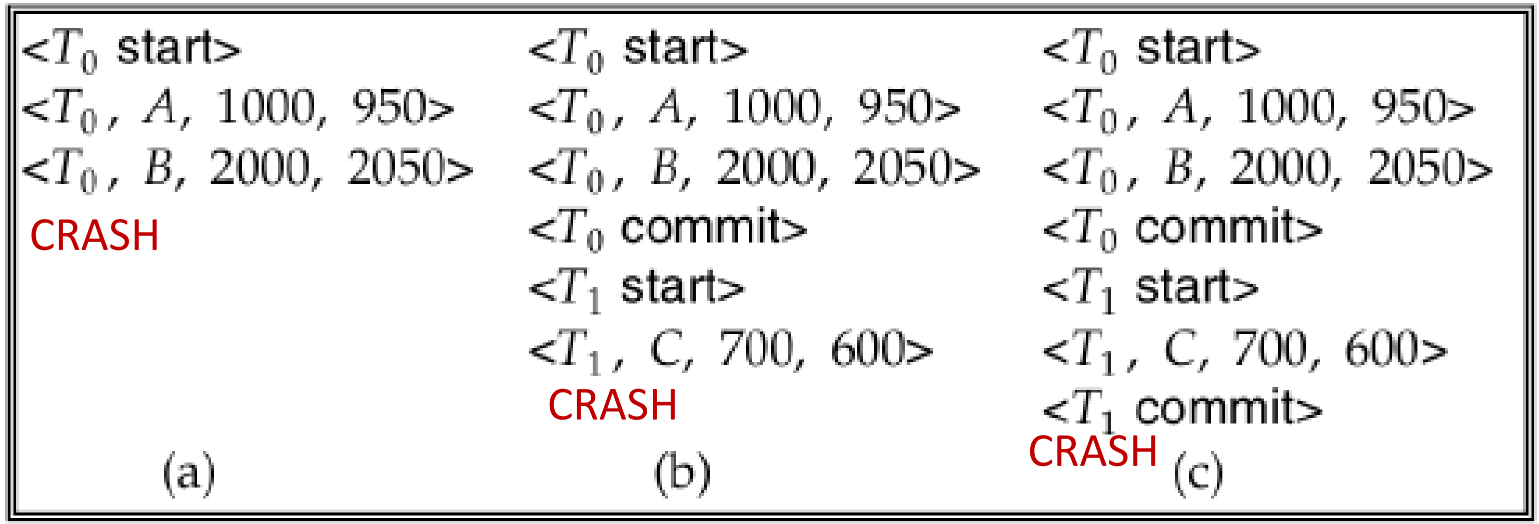
Records in the log file:

Start:	<T_1 start>		
Update:	<T_1, C, 700, 600>	previous value:700	new value 600
Commit:	<T_1 commit>		
Abort:	<T_1 abort>		

DB Modification: an example

Log	**Write**	**Output**
<T_0 **start**>		
<T_0 , A, 1000, 950>		
	A = 950	
<T_o , B, 2000, 2050>		
	B = 2050	
		Output(B_B)
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		Output(B_C)

Transactions executed in sequence



$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$	$<T_0, A, 1000, 950>$
$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$	$<T_0, B, 2000, 2050>$
CRASH	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 700, 600>$	$<T_1, C, 700, 600>$
	CRASH	$<T_1$ commit>
		CRASH
(a)	(b)	(c)

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

DB Modification: concurrent transactions

Log	**Write**	**Output**
<T_0 **start**>		
<T_0 , A, 1000, 950>		
	A = 950	
<T_o , B, 2000, 2050>		
	B = 2050	
		Output(B_B)
<T_1 **start**>		
<T_0 **commit**>		
<T_1, C, 700, 600>		
	C = 600	
		Output(B_C)
CRASH		

Recovery actions

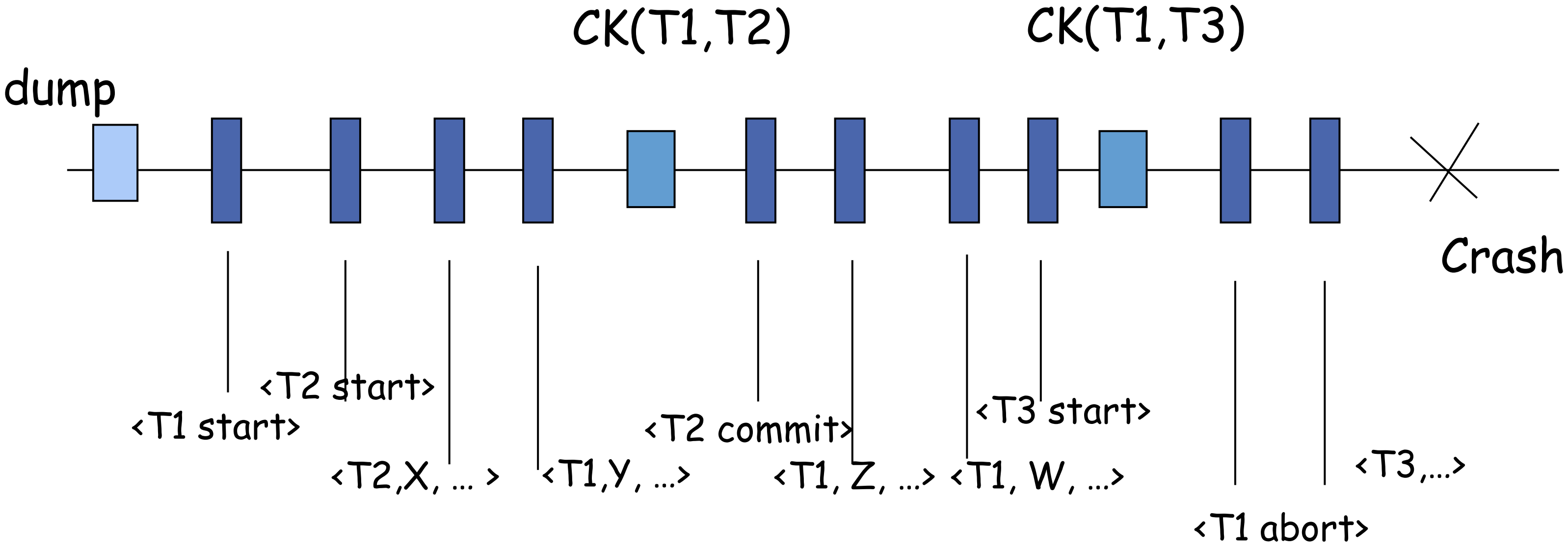
- undo (T_1) A reset to 950
  B reset to 2050
- redo (T_0) C is restored to 700

Checkpointing

CHECKPOINT operation:

output all modified buffer blocks of committed transactions to the disk

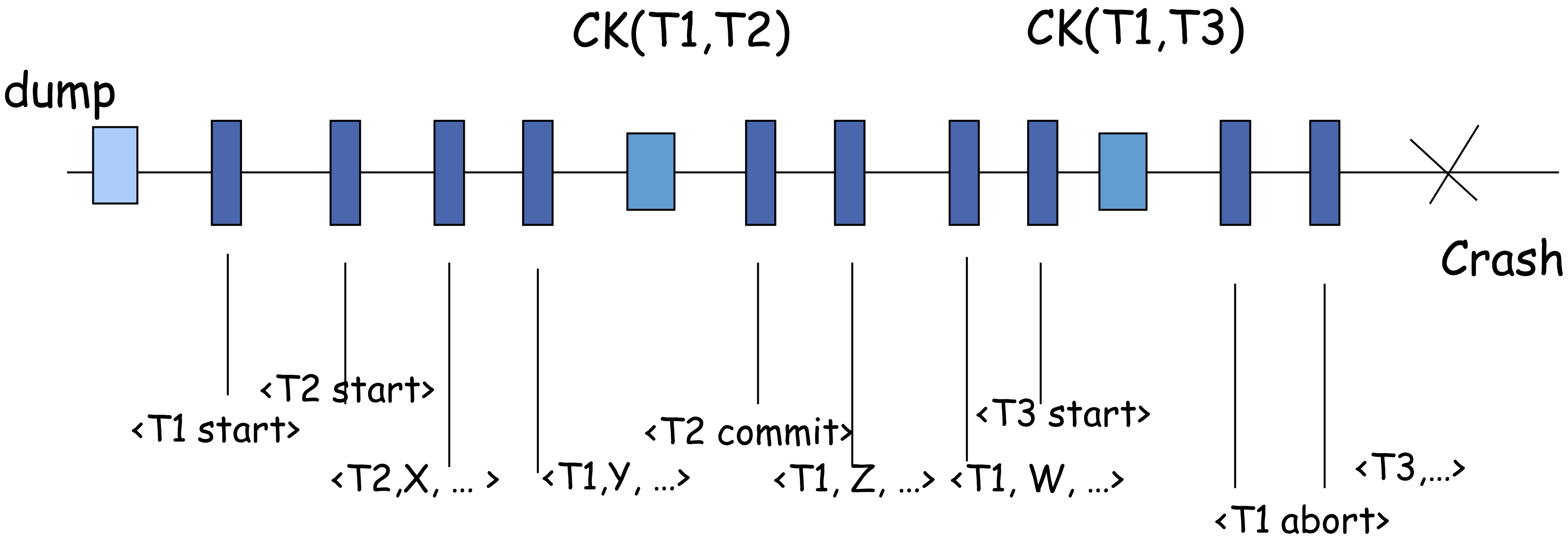
CK(Ti, …, Tk) record in the Log, where Ti, …, Tk are the transactions active at the checkpoint



Checkpointing

To Recover from system failure:
- consult the Log
- redo all transactions in the checkpoint or started after the checkpoint that committed;
- undo all transaction in the checkpoint or started after the checkpoint that are not committed

To recover from disk failure:
- restore database from most recent dump
- apply the Log Recovery



Recovery algorithm

<T_0 **start**>
<T_0, A, 0, 10>
<T_0 **commit**>
<T_1 **start**>
<T_1, B, 0, 10>
<T_2 **start**>
<T_2, C, 0, 10>
<T_2, C, 10, 20>
<checkpoint {T_1, T_2}>
<T_3 **start**>
<T_3, A, 10, 20>
<T_3, D, 0, 10>
<T_3 **commit**>
crash

Redo={} Undo={}

Scan the Log backword until checkpoint:
<T commit> -> add T to Redo
<T start> if T is not in Redo, add T to Undo

For each T acticve at the checkpoint,
if T is not in Redo, add T to Undo

Redo={T3} Undo={T1, T2}

Redo={T3} Undo={}

Redo={} Undo={}

Log file

Log file in a stable storage

Moreover:

before the commit of a transaction,

save the block of the Log file contaning the records of the transaction on disk

before updating the database,

save the block of the Log file containing the update record on disk

It is always possible to execute Undo and it is always possible to execute Redo of a transaction if needed

Atomic actions

Advantages of atomic actions:

a designer can reason about system design as

1) no failure happened in the middle of a atomic action

2) separate atomic actions access to consistent data
(property called “serializability”, concurrency control).