Recovery System

Outline

- Failure Classification
- Stable Storage Implementation
- Recovery and Atomicity
- Log-Based Recovery
  - Logical undo
- Remote Backup Systems

Recovery System

High availability

- **Idea**: the DBMS should be *always* usable...
- ... but DBMSs are subject to failures:
  - Disk crash, power lost, software error, fire in machine room, sabotage, etc.
- Need to prevent information lost!
  - To fulfill this, synchronized backup copies are necessary
  - In case of failure, keep working on a backup copy

- **Recovery system:** it restores DB to a consistent state previous to the failure. **Two** parts:
  1. *During normal processing*: it collects relevant info, used in case of failure recovery.
  2. *After a failure*: it restores the DB to a state that ensures *consistency*, *atomicity* and *durability*.

Main or backup system

Types of Failure

Logical inconsistencies

- **Transaction failure**: due to an error that can be...
  - *Logical error*: transaction doesn't complete due to internal conditions
  - *System error*: transaction is terminated by DBMS due to a condition such as a *deadlock* (can be re-executed)
- **System crash**: due to hardware or software failure
  - Fail-stop assumption: in a crash, non-volatile storage contents are not corrupted

    Fair assumption: DBMS have many integrity checks to prevent disk-data corruption
- **Disk failure**: Some blocks lose their content due to a disk (e.g., head crash) or data-transfer failure

Log-Based Recovery

- **Goal**: perform **all** or **no** DB modification issued by a given T_i.
- How to ensure this if there is a **failure** when several transactions are being processed?
  1. Each transaction stores info about upcoming operations before changing the DB
  2. After a failure, the recovery system will **undo** or **redo** the different operations

Ensure a **consistent state**

- To store DB modification operations, we can use a **log**:
  - Sequence of **log records**, saved on **stable** storage before **write**(X)
  - Once written in the log, the DB can be modified.
    - DB modification ≡ update on disk buffer / on disk itself
  - Log records are appended to the log file
    - Note the overhead imposed by logging!
    - Note too that the log might become unreasonably large

Alternatives: *Shadow-copies*, *Shadow-paging*, …

Log-Based Recovery

- Step 1: We maintain a **log** to be prepared for possible failures
- Types of **log records**:
  - $<T_i, X_j, V_o, V_n>$, when DB is *updated*:
    - T_i : *Id.* of the transaction that issued the write operation
    - X_j : *Data-item id.* (**block_id + offset**) of the data item written
    - V_o : *Old value* of the data item prior to the write
    - V_n : New value to be written
  - $<T_i$ **start**$>$, when transaction T_i starts
  - $<T_i$ **commit**$>$, after T_i commits
    - The output of the block containing the *commit log record* is the **single atomic action** that denotes a commit
  - $<T_i$ **abort**$>$, after T_i aborts

Log	DB updates
$<T_0$ **start**$>$	
$<T_0$, A, 1000, 950$>$	
	$A = 950$
$<T_0$, B, 2000, 2050$>$	
	$B = 2050$
$<T_0$ **commit**$>$	
$<T_1$ **start**$>$	
$<T_1$, C, 700, 600$>$	
	$C = 600$
$<T_1$ **commit**$>$	

Log-Based Recovery

- Step 2: in case of failure, recover a consistent state of the DB

- Note that after a failure, our picture is:
  - We might have a log file with a set of operations of different transactions
    - Transactions can be unfinished (no commit/abort log record)
  - We have a version of the DB in disk which might be not up-to-date
    - Due to, e.g., operations in buffer that weren't sent to disk
    - DB in disk doesn't reflect all modifications of the log
    - DB state might be even inconsistent!

Log	DB updates
<T_0 **start**>	
<T_0, A, 1000, 950>	
	A = 950
<T_0, B, 2000, 2050>	
	B = 2050
<T_0 **commit**>	
<T_1 **start**>	
<T_1, C, 700, 600>	
	C = 600
<T_1 **commit**>	

Log-Based Recovery

- Step 2: in case of failure, recover a consistent state of the DB
- Each transaction T_i can be:
  - **redone**: data items updated by T_i are set to their *new* values
    - Performed in a forward passing from the **first** log record
    - Usually, interleaved for all transactions
  - **undone**: data items updated by T_i are restored to their *old* values
    - Performed in a backward passing from the **last** log record
    - Recorded in the **log**, too:
      - <T_i , X , V> , **redo-only** log record when item X is restored to its old value V.
      - <T_i **abort**>, when undo is completed

Log	DB updates
<T_0 **start**>	
<T_0, A, 1000, 950>	
	A = 950
<T_0, B, 2000, 2050>	
	B = 2050
<T_0 **commit**>	
<T_1 **start**>	
<T_1, C, 700, 600>	
	C = 600
<T_1 **commit**>	

Order is important!!

Log-Based Recovery

- Step 2: in case of failure, recover a consistent state of the DB
- Each transaction T_i is:
  - **redone** if the log contains **both**:
    - <T_i **start**> record
    - <T_i **commit**> or <T_i **abort**> record
  - **undone** if the log contains:
    - <T_i **start**> record
    - but there is **no** <T_i **commit**> **nor** <T_i **abort**> record

Log	DB updates
<T_0 **start**>	
<T_0, A, 1000, 950>	
	A = 950
<T_0, B, 2000, 2050>	
	B = 2050

Log-Based Recovery

- Do we really want to redo/undo *all* the transactions in the *entire* log?
  - This is time-consuming!!

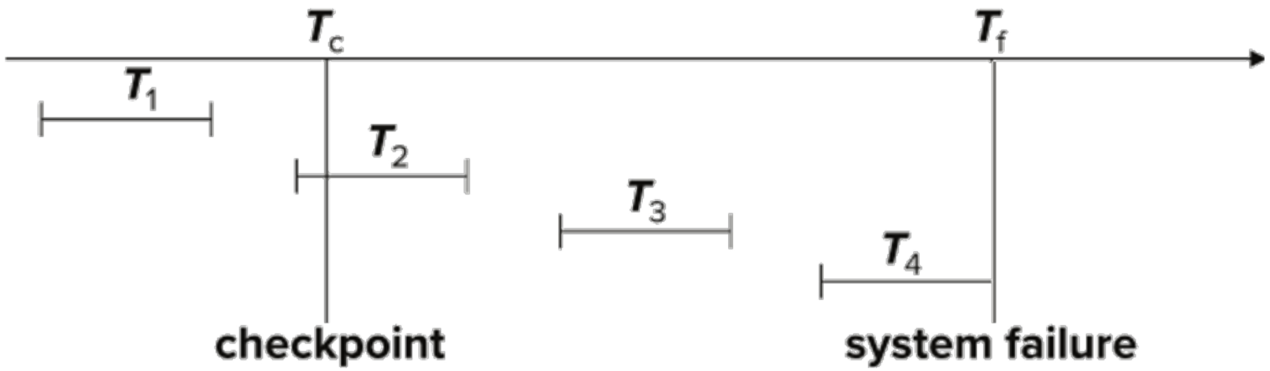
- We use **checkpoints**. Periodically:
  1. Save onto stable storage any log record in main memory.
  2. Output to disk all modified DB buffer blocks.
  3. Save a log record <**checkpoint** *L*>, where *L* is a list of all transactions active at the time of the checkpoint.

  - All updates are stopped while doing a checkpoint

Reformulate Step 2

Log-Based Recovery

- Step 2: in case of failure, recover a consistent state of the DB
- Find the most recent < **checkpoint** *L* >
  - Consider only transactions in *L* *or* with a <T_i **start**> record **after** the checkpoint.
- Each transaction T_i is:
  - **redone** if the log contains
    - <T_i **commit**> or <T_i **abort**> record
  - **undo**(T_i): otherwise



Transactions finished (committed/aborted) before the checkpoint are not considered. They can be erased.

Log-Based Recovery

Recovery algorithm: two phases

1. Redo phase

1. Find last <**checkpoint** *L*> record, and set *undo-list* = *L*.
2. Scan forward from that checkpoint:
    1. If a record $<T_i, X_j, V_1, V_2>$ is found, **redo** it (write V_2 to X_j)
       Regular update
    2. If a record $<T_i, X_j, V>$ is found, **redo** it (write V to X_j)
       Redone-only operation
    3. If a log record <T_i **start**> is found, **add** T_i to *undo-list*
    4. If a log record <T_i **commit**> *or* <T_i **abort**> is found, **remove** T_i from *undo-list*

- At the end, *undo-list* contains all incomplete transactions

2. Undo phase

Recovery Algorithm

Recovery algorithm: two phases

1. **Redo phase**
2. **Undo phase:** scan log **backwards** from end *until* *undo-list* is empty:
   - When it finds a **log record** for a T_i which is in the *undo-list* :
     1. If it is a $<T_i, X_j, V_1, V_2>$ log record:
        1. Write a **redone-only** log record $<T_i , X_j, V_1>$
        2. Assign value V_1 to X_j (**undo** operation)
     2. If it is a $<T_i$ **start**$>$ log record:
        1. Write a log record $<T_i$ **abort**$>$
        2. Remove T_i from *undo-list*

After undo phase, normal processing can resume

Recovery Algorithm

- E.g.,

Log

<T_0 **start**>
<T_0, B, 2000, 2050>
<T_1 **start**>
<checkpoint $\{T_0, T_1\}$>
<T_1, C, 700, 600>
<T_1 **commit**>
<T_2 **start**>
<T_2, A, 500, 400>
<T_0, B, 2000>
<T_0 **abort**>

Crash!!

Most recent checkpoint

Redo pass

L=$\{T_0, T_1\}$
redo C=600; L=$\{T_0, T_1\}$
L=$\{T_0\}$
L=$\{T_0, T_2\}$
redo A=400; L=$\{T_0, T_2\}$
redo B=2000; L=$\{T_0, T_2\}$
L=$\{T_2\}$

Undo pass

L=$\{\}$
undo A=500; L=$\{T_2\}$
----; L=$\{T_2\}$
----; L=$\{T_2\}$

Exercise: recovery algorithm

- Apply the recovery algorithm:

Log
<T_0 **start**>
<T_0, B, 2000, 2050>
<T_1 **start**>
<checkpoint {T_0, T_1}>
<T_1, C, 700, 600>
<T_1 **commit**>
<T_2 **start**>
<T_2, A, 500, 400>
<T_0, C, 600, 550>

Log-Based Recovery: Logical Undo

- Some operations cannot be undone by restoring old values, as usual
  Other transactions may have updated the value in the meantime!

- We use **logical undo**: Undo with **compensating operations**
  - Log records contain the undo operation to be executed (**logical undo logging**)
  - *Redo* is always logged physically

- **Operation logging** works as:
  1. Write log record <T_i, O_j, **operation-begin**> when O_j starts.
  2. Normal physical records are used for the modifications.
  3. Write log record <T_i, O_j, **operation-end**, U> when O_j finishes.
     - U contains the info. required to perform a **logical undo**

Log
<T_0 **start**>
<T_0, B, 2000, 2050>
<T_0, O_1, operation-begin>
<T_0, C, 700, 600>
<T_0, O_1, operation-end, (C,+100)>
<T_1 **start**>
<T_1, O_2, operation-begin>
<T_1, C, 600, 400>
<T_1, O_2, operation-end, (C,+200)>
// T_0 is instructed to abort
<T_0, C, 400, 500>
<T_0, O_1, operation-abort>
<T_0, B, 2000>
<T_0 **abort**>
<T_1 **commit**>

Log-Based Recovery: Logical Undo

- E.g.,

Log

<T_0 **start**>
<T_0, B, 2000, 2050>
<T_0 **commit**>
<T_1 **start**>
<T_1, B, 2050, 2100>
<T_1, O_4, operation-begin>
<checkpoint {T_1}>
<T_1, C, 700, 400>
<T_1, O_4, operation-end, (C,+300)>
<T_2 **start**>
<T_2, O_5, operation-begin>
<T_2, C, 400, 300>

Crash!!

Most recent checkpoint

Redo pass

L={T_1}
redo C=400; L={T_1}
----; L={T_1}
L={T_1, T_2}
----; L={T_1, T_2}
redo C=300; L={T_1, T_2}

L={}
undo B=2050; L={T_1}
----; L={T_1}
----; L={T_1}
----; L={T_1}
undo C=700; L={T_1}
L={T_1}
----; L={T_1, T_2}
undo A=400; L={T_1, T_2}

Undo pass

Exercise: Logical Undo

- How does it look the log after recovery?

Log
<T_0 **start**>
<T_0, O_6, operation-begin>
<T_0, B, 2000, 2050>
<T_0, O_6, operation-end, (B,-50)>
<T_1 **start**>
<checkpoint {T_0, T_1}>
<T_1, C, 700, 600>
<T_2 **start**>
<T_1 **commit**>
<T_0, C, 600, 550>
<T_2, O_7, operation-begin>
<T_2, B, 2050, 1900>
Crash!!

Types of Failure

- **Transaction failure**: due to an error that can be...
  - *Logical error*: transaction doesn't complete due to internal conditions
  - *System error*: transaction is terminated by DBMS due to a condition such as a *deadlock* (can be re-executed)

Physical inconsistencies

- **System crash**: due to hardware or software failure
  - Fail-stop assumption: in a crash, non-volatile storage contents are not corrupted

    Fair assumption: DBMS have many integrity checks to prevent disk-data corruption

- **Disk failure**: Some blocks lose their content due to a disk (e.g., head crash) or data-transfer failure

Protection against Data-Transfer Failure

- **Stable storage**: A utopian form of storage that survives *all* failures
  - Approx.: maintain 2+ copies of each block on separate disks
  - Control updates to protect against failure during data transfer, which can result in inconsistent copies
    - *Partial failure*: destination block has incorrect information
      Problem in the middle of the transference
    - *Total failure*: destination block was never updated
      Problem at the beginning

- Protection against data-transfer failure using multiple copies of blocks
  1. Write the information onto the **first** physical block.
  2. Write the same information onto the **second** physical block.

  Operation completes only after 2nd write successfully finishes

Protection against Data-Transfer Failure

- To **recover from a partial failure**:

1. First find inconsistent blocks:
   - Do not compare the two copies of every disk block.
     Expensive!
   - Check only blocks with writes in-progress
     - Store in-progress disk writes on non-volatile storage
2. With each inconsistent block:
   1. If a copy has an error (e.g., *bad checksum*), overwrite it with the other copy
   2. If both have no error (but are different), we have two options:
      - Overwrite the second block by the first block (*successful write*)
      - Overwrite the first block by the second block (*no change at all*)

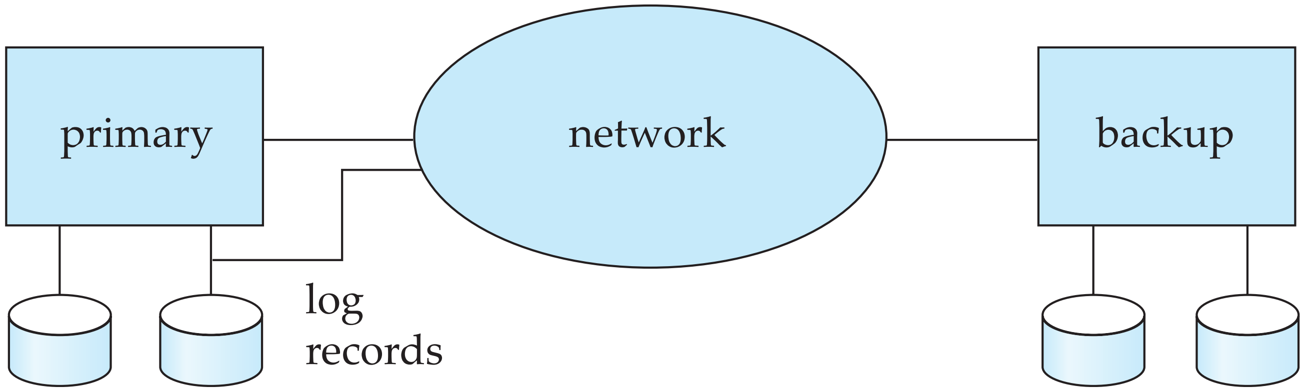
Protection against Disk Crash

- To prevent loss of data, use **dumps**
  Similar to checkpointing
  - Periodically, copy (dump) the entire DB to stable storage in backup system
    - No transaction may be active during the dump
  - Procedure:
    1. Output all buffer log records onto disk
    2. Output all buffer blocks onto disk
    3. Copy the contents of the DB to stable storage
    4. Output a record <**dump**> to log

- To recover from disk failure:
  1. Restore DB from most recent **dump**
  2. Read the **log** and *redo* all transactions **committed** after the dump

Remote Backup Systems

- Remote backup systems to provide *high availability*
  - Transaction processing goes on even when the primary site fails



  - Both subsystems are physically separated
  - Log records are sent to remote system too
- If primary system fails, for remote system to take control, it needs to get up-to-date (recover)
  - From data copied and log records

Remote Backup Systems

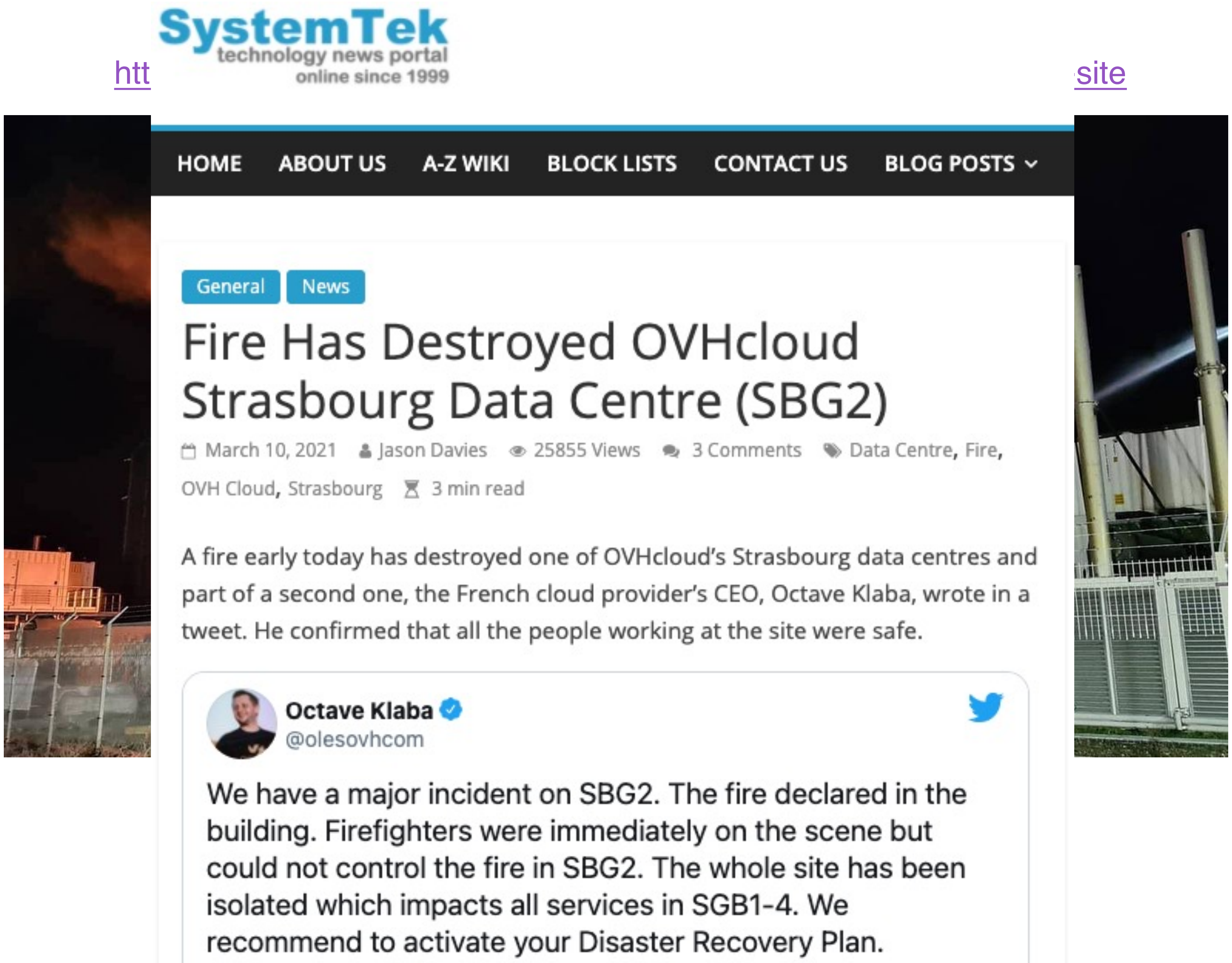
- To reduce **time to recover**, backup site periodically *redoes log records*.
  - Sets a new *checkpoint* and deletes earlier parts of the log

- To ensure **persistence**, transaction commit is only effective when the log records reach the backup.
  - Note the delay of commit
    Trade-off time-processing vs. persistence

- In practice, different approaches:
  - **One-safe**: transaction committed as its log record is written at primary
  - **Two-very-safe**: transaction committed as its log record is written at both sites
    - Reduces availability! (transactions cannot commit if either site fails)
  - **Two-safe**: it behaves as *two-very-safe* if both sites are active. If backup is down, as *one-safe*

Remote Backup Systems

htt site



SystemTek
technology news portal
online since 1999

HOME ABOUT US A-Z WIKI BLOCK LISTS CONTACT US BLOG POSTS

General News

Fire Has Destroyed OVHcloud Strasbourg Data Centre (SBG2)

March 10, 2021 Jason Davies 25855 Views 3 Comments Data Centre, Fire, OVH Cloud, Strasbourg 3 min read

A fire early today has destroyed one of OVHcloud's Strasbourg data centres and part of a second one, the French cloud provider's CEO, Octave Klaba, wrote in a tweet. He confirmed that all the people working at the site were safe.

Octave Klaba @olesovhcom

We have a major incident on SBG2. The fire declared in the building. Firefighters were immediately on the scene but could not control the fire in SBG2. The whole site has been isolated which impacts all services in SGB1-4. We recommend to activate your Disaster Recovery Plan.

Recovery System