Lecture 28: Recovery

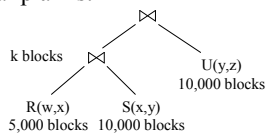
Friday, December 1, 2000

Outline

- Finish example 7.38
- Logging (8.1)
- Undo loging (8.2)

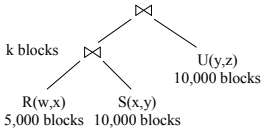
Example 7.38

- Logical plan is:



- Main memory M = 101 buffers

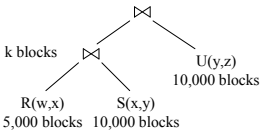
Example 7.38



Naïve evaluation:

- 2 partitioned hash-joins
- Cost 3B(R) + 3B(S) + 4k + 3B(U) = 75000 + 4k

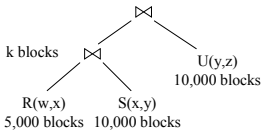
Example 7.38



Smarter:

- Step 1: hash R on x into 100 buckets, each of 50 blocks; to disk
- Step 2: hash S on x into 100 buckets; to disk
- Step 3: read each Ri in memory (50 buffer) join with Si (1 buffer); hash result on y into 50 buckets (50 buffers) -- here we *pipeline*
- Cost so far: 3B(R) + 3B(S)

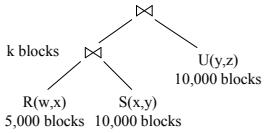
Example 7.38



Continuing:

- How large are the 50 buckets on y ? Answer: k/50.
- If k <= 50 then keep all 50 buckets in Step 3 in memory, then:
- Step 4: read U from disk, hash on y and join with memory
- Total cost: 3B(R) + 3B(S) + B(U) = 55,000

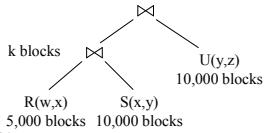
Example 7.38



Continuing:

- If 50 < k <= 5000 then send the 50 buckets in Step 3 to disk
  - Each bucket has size k/50 <= 100
- Step 4: partition U into 50 buckets
- Step 5: read each partition and join in memory
- Total cost: 3B(R) + 3B(S) + 2k + 3B(U) = 75,000 + 2k

Example 7.38



Continuing:

- If k > 5000 then materialize instead of pipeline
- 2 partitioned hash-joins
- Cost 3B(R) + 3B(S) + 4k + 3B(U) = 75000 + 4k

Example 7.38

Summary:

- If k <= 50, cost = 55,000
- If 50 < k <=5000, cost = 75,000 + 2k
- If k > 5000, cost = 75,000 + 4k

Types of Failures

- Wrong data entry
  - Prevent by having constraints in the database
  - Fix with data cleaning
- Disk crashes
  - Prevent by using redundancy (RAID, archive)
  - Fix by using archives
- Fire, theft, bankruptcy…
  - Buy insurance, change profession…
- System failures: *most frequent* (e.g. power)
  - Use recovery

System Failures

- Each transaction has *internal state*
- When system crashes, internal state is lost
  - Don't know which parts executed and which didn't
- Remedy: use a **log**
  - A file that records every single action of the transaction

Transactions

- In ad-hoc SQL
  - each command = 1 transaction
- In embedded SQL
  - Transaction starts = first SQL command issued
  - Transaction ends =
    - COMMIT
    - ROLLBACK (=abort)

Transactions

- Assumption: the database is composed of *elements*
  - Usually 1 element = 1 block
  - Can be smaller (=1 record) or larger (=1 relation)
- Assumption: each transaction reads/writes some elements

Correctness Principle

- There exists a notion of correctness for the database
  - Explicit constraints (e.g. foreign keys)
  - Implicit conditions (e.g. sum of sales = sum of invoices)
- *Correctness principle*: if a transaction starts in a correct database state, it ends in a correct database state
- Consequence: we only need to guarantee that transactions are *atomic*, and the database will be correct forever

Primitive Operations of Transactions

- INPUT(X): read element X to memory buffer
- READ(X,t): copy element X to transaction local variable t
- WRITE(X,t): copy transaction local variable t to element X
- OUTPUT(X): write element X to disk

Example

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

Action	T	Mem A	Mem B	Disk A	Disk B
REAT(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

The Log

- An append-only file containing log records
- Note: multiple transactions run concurrently, log records are interleaved
- After a system crash, use log to:
  - Redo some transaction that didn't commit
  - Undo other transactions that didn't commit

Undo Logging

Log records

- <START T> = transaction T has begun
- <COMMIT T> = T has committed
- <ABORT T>= T has aborted
- <T,X,v>= T has updated element X, and its *old* value was v

Undo-Logging Rules

U1: If T modifies X, then <T,X,v> must be written to disk before X is written to disk

U2: If T commits, then <COMMIT T> must be written to disk only after all changes by T are written to disk

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
REAT(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,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	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>

Recovery with Undo Log

After system's crash, run recovery manager

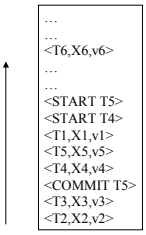
- Idea 1. Decide for each transaction T whether it is completed or not
  - <START T>….<COMMIT T>…. = yes
  - <START T>….<ABORT T>……. = yes
  - <START T>…… = no
- Idea 2. Undo all modifications by incompleted transactions

Recovery with Undo Log

Recovery manager:

- Read log from the end; cases:
  - <COMMIT T>: mark T as completed
  - <ABORT T>: mark T as completed
  - <T,X,v>: if T is not completed
    then write X=v to disk
    else ignore
  - <START T>: ignore

Recovery with Undo Log



Recovery with Undo Log

- Note: all undo commands are *idempotent*
  - If we perform them a second time, no harm is done
  - E.g. if there is a system crash during recovery, simply restart recovery from scratch

Recovery with Undo Log

When do we stop reading the log ?

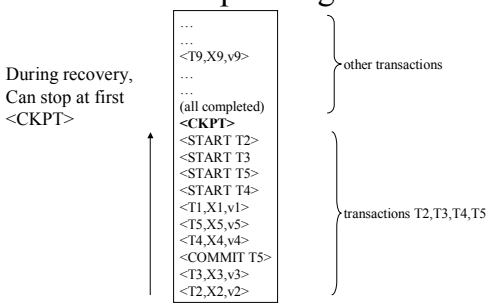
- We cannot stop until we reach the beginning of the log file
- This is impractical
- Better idea: use checkpointing

Checkpointing

Checkpoint the database periodically

- Stop accepting new transactions
- Wait until all curent transactions complete
- Flush log to disk
- Write a <CKPT> log record, flush
- Resume transactions

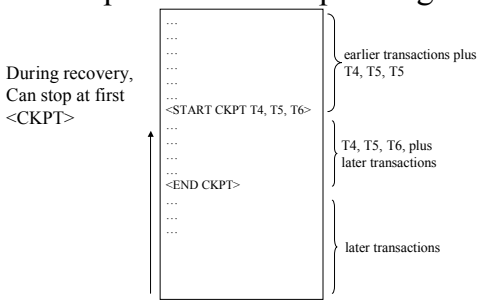
Undo Recovery with Checkpointing



Nonquiescent Checkpointing

- Problem with checkpointing: database freezes during checkpoint
- Would like to checkpoint while database is operational
- =nonquiescent checkpointing

Nonquiescent Checkpointing

- Write a <START CKPT(T1,…,Tk)> where T1,…,Tk are all active transactions
- Continue normal operation
- When all of T1,…,Tk have completed, write <END CKPT>

Undo Recovery with Nonquiescent Checkpointing