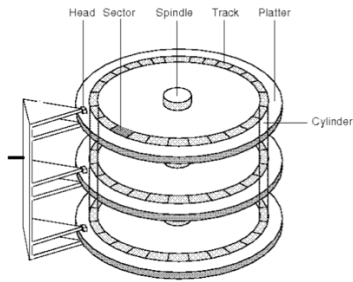


## Storage

### Block

- data is stored and retrieved in units (logical) called blocks
- The unit of access (or the unit of transfer between disk and memory) is always a block even if a single bit is affected

### Component of Disk



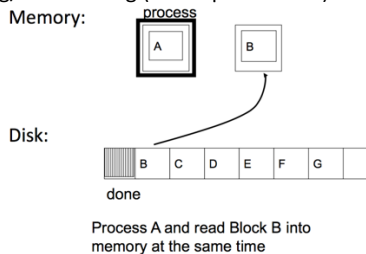
\*remember there are two surfaces per platter

- A block consists one or more sectors (physical)

Access time: **seek time + rotational delay + transfer time**

- Seek time: time for finding the target cylinder
- Rotational delay: time for  $\frac{1}{2}$  revolution (rotate to the target sector)
- Transfer time = block size / transfer rate
- If the block going to read is the next block: sequential IO

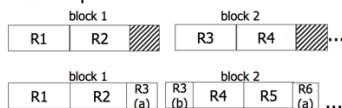
### Double buffering/ Prefetching (Disk Optimization)



## File Storage

### Storing records in blocks

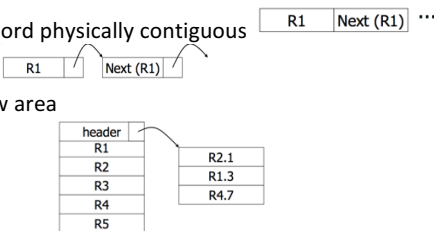
- Fixed length vs Variable length
- Spanned vs Un-spanned



### Block factor

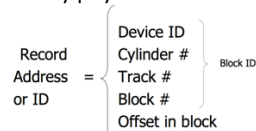
#### Storing a relation

1. Next record physically contiguous
2. Linked
3. Overflow area

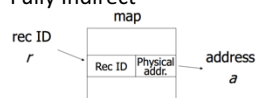


### Record address

- Purely physical



- Fully Indirect



### Block access

- Reading or writing: a block access costs 1-time unit
- Processing in RAM is free

### Cache

- Keep a cache of recently accessed blocks in RAM
- Goal: request for block can be satisfied from disk cache instead of the actual disk

## Indexing

### Why index helps query processing?

- Number of index blocks is usually small compared with the number of data blocks
- Usually, index blocks are small enough to fit in main memory. Therefore, searching for keys only require memory operation)
- Point to the relevant data blocks directly

### Understand index (index is sorted)

Dense	Sparse
One entry one record	One entry one block (search key is 1 <sup>st</sup> record)
-Efficient for queries that specify search key value -Know <b>existence</b> of a data record without access data records	- <b>Less space</b> consulted -The records must be <b>sorted</b>
Clustered	Unclustered
Entry order = record order <i>One for each relation</i>	Entry order $\neq$ record order <i>One or more</i>
Good for <b>range</b> query	Good for updates
Primary	Secondary
Entry = primary key Only one for each relation	Entry $\neq$ primary key One or more
$\approx$ clustered index (determine the placement of records in the data file)	$\approx$ unclustered index (does not determine the placement of records in the data file)

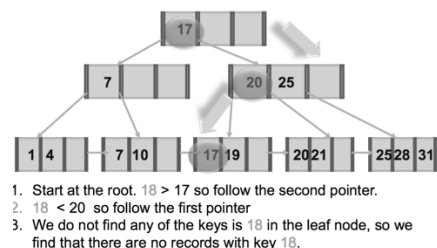
### Multi-level index:

- the 2<sup>nd</sup> level must be sparse
- The 1<sup>st</sup> level of secondary index must be dense

### B+ tree

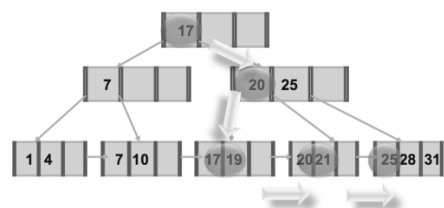
- 1<sup>st</sup> level is dense index

### Look up (look up 18)



1. Start at the root. 18 > 17 so follow the second pointer.
2. 18 < 20 so follow the first pointer
3. We do not find any of the keys is 18 in the leaf node, so we find that there are no records with key 18.

### Range query (R.K >=17 AND R.K <=26)



### Properties

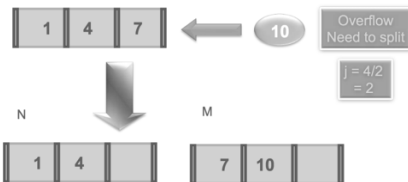
1. Let v be a key in a non-leaf node, and p be the pointer to the right of v, v always equal to the smallest key value in the (right-side) sub-tree pointed by p
2. Minimum key rule

	Non-leaf node	Leaf node	Root
# pointers	$\lceil (n+1)/2 \rceil$	$\lfloor (n+1)/2 \rfloor$	$1(1 \text{ record})/2$
# keys	$\lceil (n+1)/2 \rceil - 1$	$\lfloor (n+1)/2 \rfloor$	1

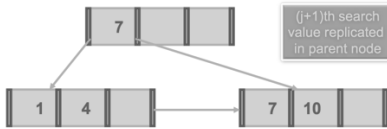
### Build B+ tree

- Insertion of a node

- When there is overflow, the first  $j = \lceil (n+1)/2 \rceil$  keys remain in old node, put the rest in new node



- Update sequential pointer and parent node



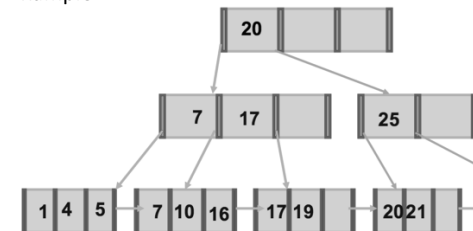
- Non-leaf node split: the first  $j = \lceil n/2 \rceil$  keys remain in old node, put the rest in new node
- Update the pointers and higher level node

#### Bulk insertion

- Initialization: sort all data entries & empty root
- Insert pointer to first leaf in a new root node
- If root node is full, split the root and create new root page.
- Lower IO during build, advantages for concurrency control

#### Deletion of a node

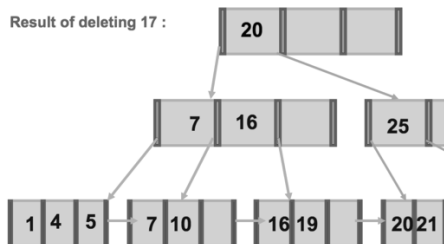
- Check the number of keys and pointers the node from which a deletion occurred
- If the number is less than the requirement



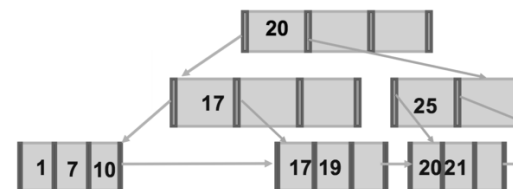
Case 1

Adjust parent : 17 → 16

Result of deleting 17 :



case 2

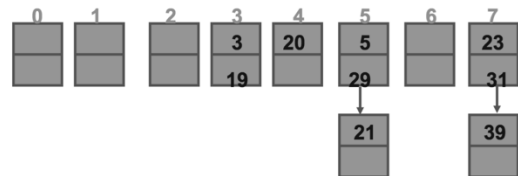


#### Maximum order(状况) and analysis of B+ tree

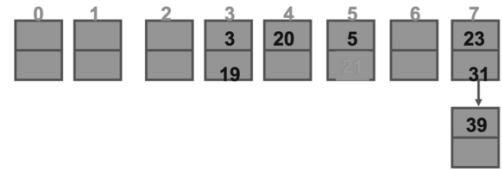
- Max number of pointers pointing to records = # of keys in leaf nodes
- Disk-based tree I/O:  $h + 1$  (access data record)
  - Minus 1 if root is cached in memory
- Height (h):  $\log_f N$ , N: # records, f: fanout max # keys  
 $h = \log_p(N/k) + 1$  p: max # pointers; k: max # keys

#### Hash index

##### Static hash

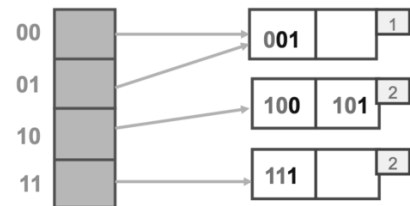


- The keys in the same bucket are NOT sorted
- Deletion (delete 29)

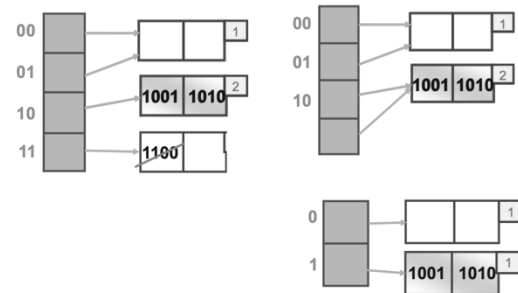


- Does not support range query

#### Extensible hash



- Buckets can share a block
- Deletion



#### Compare Hashing and B+ tree

- Hashing is only good for query with a given search key
- B+ tree is also helpful for range query

What attributes to build index in practice? What kind of index do you build?

#### Query Processing

##### Motivation

- Good algorithms can greatly improve performance

##### Basic idea of query processing

- Measure of cost: # of disk I/O
- M = # of RAM buffers
- Size of buffer = size of block
- $B(R)$  = # of blocks in R
- $T(R)$  = # of tuples in R
- $V(R, a)$  = # of distinct values of attribute 'a' in R

#### Scan (selection $\sigma$ ): read the entire content of relation R

SELECT \* FROM R WHERE a = v;

Relation R	Index	Cost
Clustered	X	$R(B)$
Unclustered	X	$T(R)$
$V(R, a)=100$	Clustering	$R(B)/100$
$V(R, a)=10$	Unclustering	$T(B)/10$
$V(R, a)=20,000$	V	1

#### Duplicate Elimination

- One memory buffer holds on block of R's tuples; M-1 buffers hold single copies of every tuple seen so far  
**Cost:  $B(R)$**
- Sort-based duplication elimination:
  - At the last phase of merge-sort write only one copy to the output

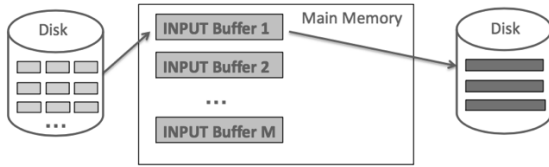
**Cost:  $3B(R)$**

## Sort

- If sorted on A and B+ tree index on A -> a scan of the index gives desired sorted order
- If R is small to fit in RAM, retrieve it using table scan and sort using RAM sorting algorithm
- If R is too large to fit in, use multi-way merge-sort algorithm (e.g. merge sort)

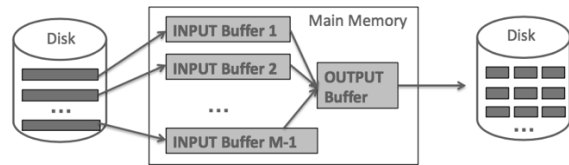
## Merge sort

Produce sorted sub-lists



## Merge the sorted sub-lists

1 input-buffer holds 1 block from a sorted sub-list



Memory requirement:  $B(R) / M < M-1$  (or  $B(R) < M^2$ )

**Cost:  $3B(R)$**

## Natural Join

Algorithm	Memory requirement	Cost	Methodology
One pass join	$M \geq \min(B(R), B(S))$	$B(R) + B(S)$	Read all tuples of S(smaller) using M-1 buffer Read each block of R into Mth buffer and JOIN
Simple block-based nested loop	1 for S, 1 for R, 1 for joint tuple $M = 3$	$B(S) + B(S) * B(R)$	[S is outer if S is smaller] For each block $b_s$ in S do For each block $b_r$ in R do Output joint results between tuples in $b_s$ & $b_r$
Block-based nested loop	—	$B(S) + B(R) \lceil B(S)/M-1 \rceil$	Use M-1 blocks hold block of outer S Use 1 block as an input buffer for scanning R And 1 output buffer for joint tuples
Sort merge join	$M^2 \geq B(R) \text{ \& } M^2 \geq B(S)$ All tuples with common y-value must fit in M buffer	$5(B(R) + B(S))$	Sort R and S using merge sort $(3+1) * (B(R) + B(S))$ Merge the sorted R and S $(B(R) + B(S))$
Refined sort-merge join	$M^2 \geq B(R) + B(S)$ All tuples with common y-value must fit in M buffer	$3(B(R) + B(S))$	Create sorted sub-lists of size M for both R and S Using M-1 input buffers to hold 1 block of each sub-list Do {find the smallest value 'y' among input buffers Identify all tuples have 'y' Output joint tuples of those with common 'y' } until one of the sub-list exhausted
Hash join	$M^2 \geq \min(B(R), B(S))$ If a partition is too large for memory -> new hash	$3(B(R) + B(S))$	Partition the tuples of R and S into list of buckets For each pair of buckets in bucketR and bucketS Join by simple natural join
Index-based join	—	$B(R) + T(R) * (B(S)/V(S, y))$ Or $T(S)/V(S, y)$	For each block b in relation R For each tuple t in b Find all tuples in S matching with t by index Join these tuples
Sorted index join	—	$B(R) + B(S)$	Purpose R and S have index on the joint attribute Perform merge join

\*Block-based nested loop join is the same with simple natural join, if the outer relation is small enough to fit in M-1 buffers; it is the same as simple block-based NL, if M equals 2

## Query Optimization

### Join

Communicative:  $R \bowtie S = S \bowtie R$

Associative:  $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$

### Selection

Communicative

$$\sigma_{\text{cond1}}(\sigma_{\text{cond2}}(R)) = \sigma_{\text{cond2}}(\sigma_{\text{cond1}}(R))$$

Associative

$$\sigma_{\text{cond}}(R \cup S) = \sigma_{\text{cond}}(R) \cup \sigma_{\text{cond}}(S)$$

Splitting law

$$\sigma_{\text{cond1 AND cond2}}(R) = \sigma_{\text{cond1}}(\sigma_{\text{cond2}}(R))$$

$$\sigma_{\text{cond1 OR cond2}}(R) = \sigma_{\text{cond1}}(R) \cup \sigma_{\text{cond2}}(R)$$

Pushing law

$$\sigma_{\text{cond}}(R \times S) = \sigma_{\text{cond}}(R) \times S$$

$$\sigma_{\text{cond}}(R \times S) = R \bowtie_{\text{cond}} S$$

### Projection

Pushing law

$$\pi_{\text{attr}}(R \times S) = \pi_{\text{attr}}(\pi_{\text{attr}}(R \times S))$$

Rules for Improving logical plan

1. Push selection down  
Move selection down the tree
2. Restrictive select  
Position the lead node relations with the most restrictive SELECT operations so they are executed first
3. Replace product with join  
Combine a PRODUCT operation with a subsequent SELECT operation into a join, if the condition represents a join condition
4. Move projection down  
Move list of projection attributes down the tree as far as possible by creating new PROJECT operations as needed

Selection	
Operation	Size of operation
$\sigma_{A=C}(R)$	$T(S) = T(R)/V(R, A)$
$\sigma_{A < C}(R)$	$T(S) = T(R)/3$
$\sigma_{A \neq C}(R)$	$T(S) = T(R) \frac{V(R,A)-1}{V(R,A)}$ (or $T(s) = T(R)$ )
$\sigma_{c1 \text{ OR } c2}(R)$	$T(S) = T(R)(1 - (1 - \frac{m1}{T(R)})(1 - \frac{m2}{T(R)}))$
Join	
Operation	Size of operation
$R \bowtie S$ (on single attribute Y)	$T(R \bowtie S) = \frac{T(R)T(S)}{\max(V(R, Y), V(S, Y))}$
$R \bowtie S$ (on multiple attributes Y)	$T(R)T(S)$ divided by max of each common attribute

#### Selecting a Join method

- One pass:  
buffer manager has enough buffers
- Nested-loop:  
no enough buffer
- Sort-merge:  
One or both relations are already sorted on join attributes  
Or there are two or more joins on the same attributes so that the result will be sorted & used directly in the second sort-join
- Index-based:  
Index on join attribute of probe relation  
Or the build relation is small
- Hash-join: Multi-pass join is necessary  
Or no opportunity to use already sorted relations or index

#### Materialization

- store the result of each operation on disk. If it is needed by another operation, that operation will read the result from disk

#### Pipelining

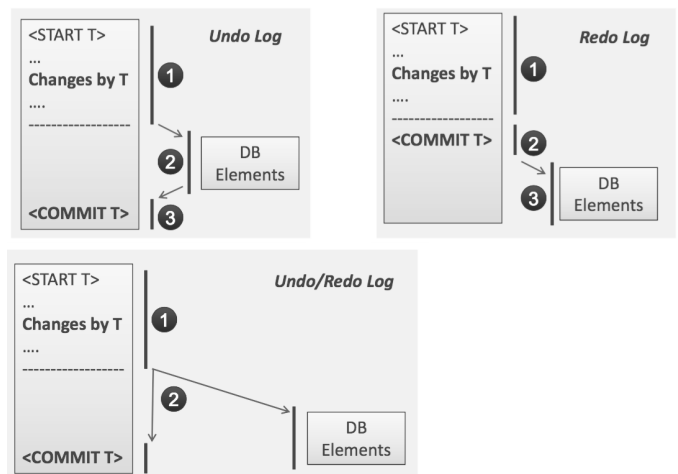
- the result produced by one operation are passed directly to the operation that uses it. Several operational must share main memory at any time

#### Failure Recovery

- Transaction
  - o the processes that query and modify the database
  - o it executes a number of steps in sequence
  - o several of these steps may modify the database
- Properties [ACID]
  - o **Atomicity**  
A transaction is either performed entirely or not performed at all
  - o **Correctness**  
If T starts with DE in consistent state, then T executes in isolation => T leaves consistent state
  - o **Isolation**  
A transaction should appear as if it is executed isolation from other transactions
  - o **Durability**  
Changes applied to the database by a committed transaction must persist in the database

Buffer to computational unit	Disk to buffer
READ(X, t)	INPUT(X)
WRITE(X, t)	OUTPUT(X)

- Transaction manager
  - o READ (X, t), WRITE (X, t)
- Buffer manager
  - o INPUT (X), OUTPUT (X)



#### Concurrency Control

##### Serial schedule

(T1 完了才接着是 T2; 不论 T1,T2 的顺序, 结果都应该是一样的)

- It contains all the actions of one transaction, then **all** the actions of another transaction and so on.
- Final state depends on the order of transactions

##### Serializable schedule:

- A schedule is serializable if it is equivalent to a serial schedule
- Not serial but the effect is the same

##### Conflict-Serializability

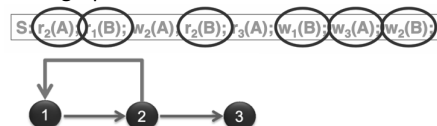
- A schedule is conflict serializable if it can be transformed into a serial schedule by a series of swapping of adjacent non-conflict actions

##### Conflict

- Adjacent actions in a schedule cannot swap
- Two actions by the same transaction  
 $ri(X); wi(Y)$
- Two writes by  $T_i, T_j$  to same element  
 $wi(X); wj(X)$
- Read/write by  $T_i, T_j$  to same element  
 $ri(X); wj(X);$   
 $wi(X); rj(X);$

##### Precedence graph test

If the graph has no circle -> conflict-serializability



##### Concurrency control by locks

- Scheduler ensures serializability by locks and timestamps
- A transaction can only read or write an element if it previously was grant a lock on that element and has not release the lock yet
- If a transaction locks an element, it must later unlock it
- No two transactions may have locked the same element

$l_i(A)$  = transaction  $T_i$  acquires lock for element A  
 $u_i(A)$  = transaction  $T_i$  releases lock for element A

##### 2PL locking (guarantee serializability)

- 1<sup>st</sup> phase: locks are obtained, no lock is released
- 2<sup>nd</sup> phase: locks are released, no more lock is obtained  
⇒ In every transaction, all lock actions (no matter which type it is) precede all unlock actions => **2PL rule**
- A transaction that obeys 2PL condition is called a two-phase-locked transaction (**2PL transaction**)
- BUT, deadlock is possible

##### Different lock modes

- Types
  - o Shared lock (for READ)
  - o Exclusive lock (for WRITE)
  - o Update lock (shared -> exclusive)

$sl_i(A)$  = transaction  $T_i$  requests a **shared lock** for element  $A$   
 $xl_i(A)$  = transaction  $T_i$  requests an **exclusive lock** for element  $A$

- For each element
  - o One exclusive lock
  - o Multiple shared locks
- If an element  $X$  is locked by exclusive lock by  $T_1$ , no other transactions may lock  $X$ , neither by exclusive lock nor shared
- If an element  $X$  is locked by shared lock by  $T_1$ , no other transactions may lock  $X$  by exclusive lock, but they can lock  $X$  by shared lock
- Upgrading
  - o First to take a shared lock
  - o Later when  $T$  was ready to write the new value, upgrade the lock to exclusive
  - o Problem

T1	T2
$sl_1(A)$	
	$sl_2(A)$
$xl_1(A)$ DENIED	
	$xl_2(A)$ DENIED

- Update lock
  - o An update lock  $ul_i(X)$  gives  $T$  only the privilege to read  $X$ , not to write  $X$
  - o Only the update lock can be upgraded to a write lock later; a read lock cannot be updated
  - o An update lock can be granted on  $X$  even if a shared lock on  $X$  exists
  - o If an element  $X$  is locked by update lock by  $T_1$ , no other transactions may lock  $X$  by any type of locks

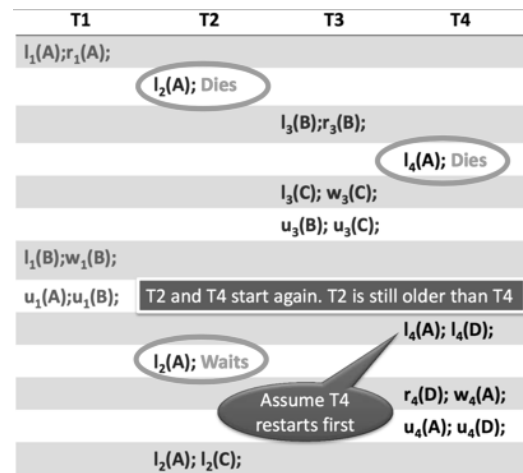
## Deadlock

- Each set of transactions is waiting for resource (e.g. lock) currently held by another transaction in the set => no one can make progress
- Timeout
  - o At least one of the transactions need to be aborted and restarted (roll back)
  - o Put a time limit on how long a transaction may be active, and if a transaction exceeds this time, roll it back
- Waits-for-graph
  - o A transaction that currently holds or waits for any lock is a node
  - o An edge from node  $T$  to  $U$ , if there is some element  $A$ , such that  $T$  desires the element  $A$  held by  $U$
  - o Deadlock  $\Leftarrow$  cycle
  - o Prevention
    - Refuse to allow an action that creates a cycle
    - Roll back any transaction that may cause cycle
- Timestamps-based

### Wait-Die

$T$  is waiting for a lock that is held by  $U$   
 If  $TS(T) < TS(U)$  //  $T$  is older than  $U$   
      $T$  waits  
 Else  
      $T$  dies;  $T$  is rolled back

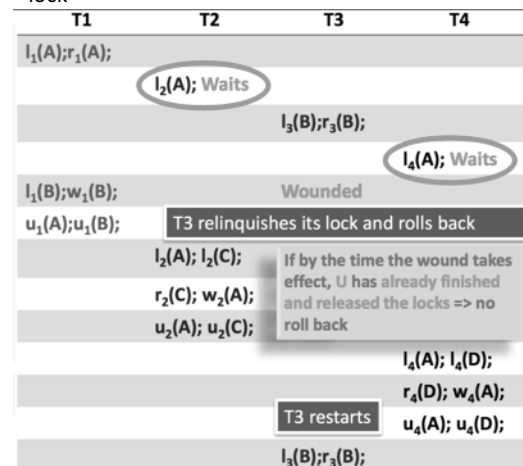
Rollback transactions tend to do little work as they are still in lock-gathering stage



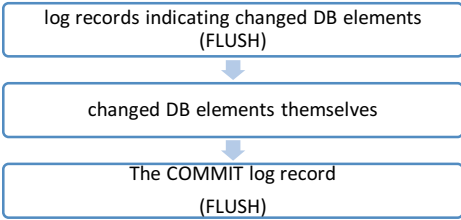
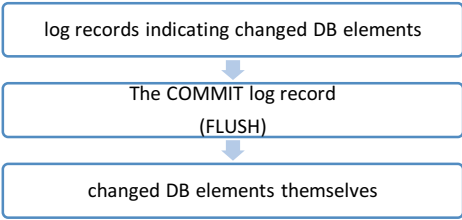
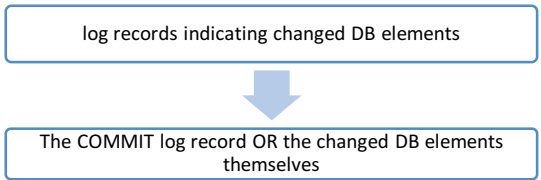
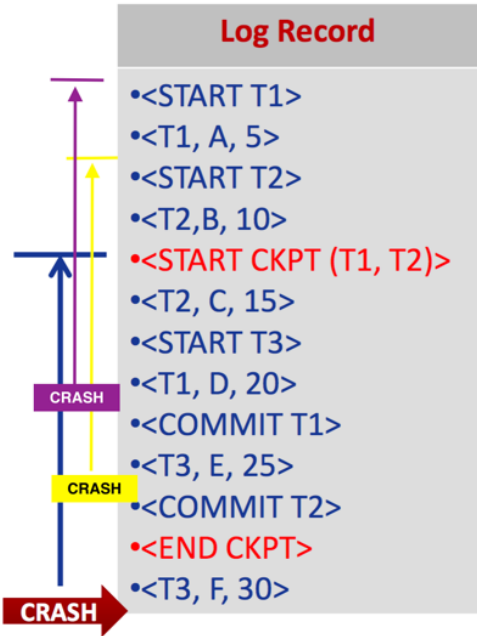
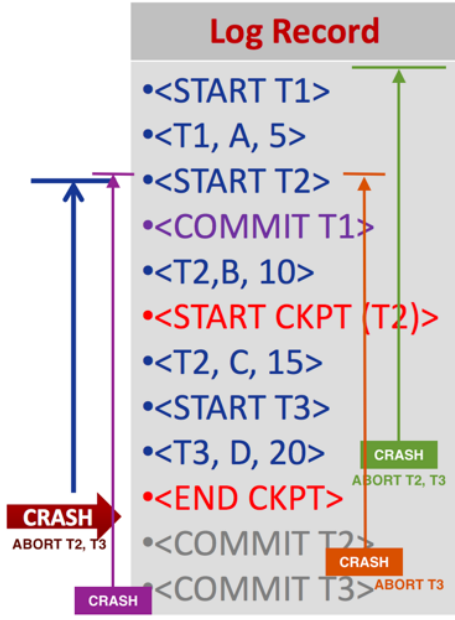
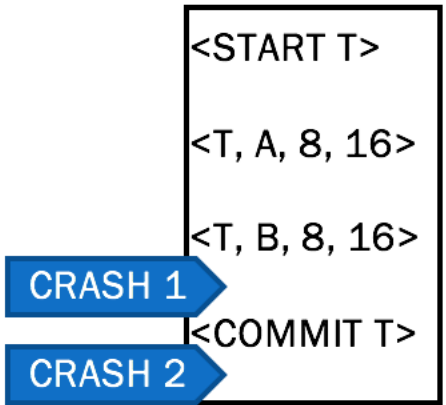
### Wounded-wait

$T$  is waiting for a lock that is held by  $U$   
 If  $TS(T) < TS(U)$  //  $T$  is older than  $U$   
      $T$  wounds  $U$ ;  $U$  rolled back  
 Else  
      $T$  waits

Rollback transactions have done more work by acquiring lock



- Compare of wait for graph and Timestamp
  - o Wait for graph minimizes the number of times we must abort a transaction due to dead lock, but the graph can be very large
  - o TS-based schemes may roll back even when there is no deadlock, but it is more time efficiency

	UNDO	REDO	UNDO/REDO
Definition	Undo all modifications by <b>incomplete</b> transactions (<START T>.....)	Redo all committed transactions before crash, <b>ignore incomplete T</b>	
<T, X, v>	<T, X, <b>old</b> >	<T, X, <b>new</b> >	<T, X, <b>old, new</b> >
Rules	If T modifies X, then <T, X, v> must be written before OUTPUT<X> <COMMIT T> must be written <b>after</b> <OUTPUT X>	If T modifies X, then both <T, X, v> and <COMMIT T> must be written to disk <b>before</b> OUTPUT(X)	The update log record<T, X, v, w> must be written <b>before</b> T modifies X <i>OUTPUT can be either before or after COMMIT</i>
Order			
Recovery Manager	Read from end	Read from beginning	Redo all committed T (top-down) Undo all un-committed T (bottom-top)
Checkpoint			 <p>CRASH 1: UNDO CRASH 2: REDO</p>
Problem	Requires that data be written immediately after a transaction finishes → increase number of disk I/O	Requires to keep all modified blocks in buffers until the transaction commits and the log records have been flushed → increase average number of buffers needed	Solution: Increase the flexibility at the expense (cost) of maintaining more information on the log

