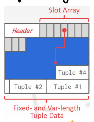
Page: Fixed-size block of data.
Hardware page: 4KB OS Page: 4KB Database Page 512B-16KB.
offset = Page Number * PageSize.

Header → Page Size, cksm, DBMS version, Transaction visibility, compression information.
DATA

Tuple Storage: Slotted pages



Header keeps track of
1. The number of used slots
2. The offset of the starting location of the last slot used

Each Tuple is assigned a unique identifier page_id + offset / loc.

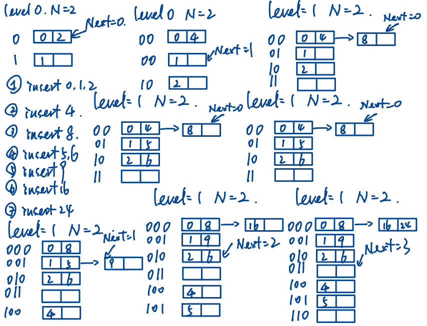
DATABASE WORKLOADS
On-Line Transaction Processing: fast operations that only read / update a small amount of data each time
On-Line Analytical Processing: Complex queries that read a lot of data to compute aggregates
Hybrid Transaction + Analytical Processing: OLTP + OLAP together on the same database instance

DATA STORAGE MODELS
N-ARY STORAGE MODEL (NSM)
The DBMS stores all attributes for a single tuple contiguously in a page
Ideal for OLTP workloads where queries tend to operate only on individual entity and insert-heavy workloads.
Advantages: · Fast inserts, updates, deletes
· Good for queries that need the entire tuple
Disadvantages: Not good for scanning portions of the table and / or a subset of the attributes.

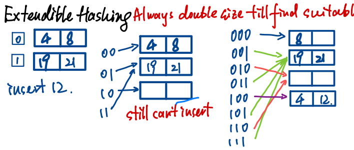
DECOMPOSITION STORAGE MODEL (DSM)
The DBMS stores the values of a single attribute for all tuples
Ideal for OLAP workloads where read-only queries perform large scans over a subset of the table's attribute.
TUPLE IDENTIFICATION: ¹Fixed Length ²Embedded Tuple ID.
Advantages: Better query processing and data compression. Reduces the amount of wasted I/O because the DBMS reads data it needs.
Disadvantages: Slow for point queries, inserts, updates and deletes because of tuple splitting / stitching

HASHING: Linear Hashing solving collision: 1. tombstone 2. Movement.
NON UNIQUE KEY 1. Separate Linked List 2. Redundant Keys ✓
ROUND ROBIN HASHING. A→1 B→1→2. C→2→3 D→|→2→3(C→4)
CUCKOO HASHING. On insert, check every table and pick anyone with free slot. If no table has free slot, evict one item and rehash it.
Table 1: | A | B | C → evict A → rehash A → evict B → rehash B
Table 2: | A |

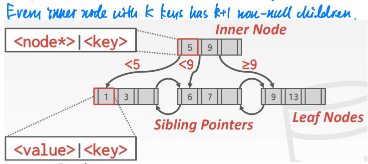
CHAINED HASHING LINEAR HASHING.



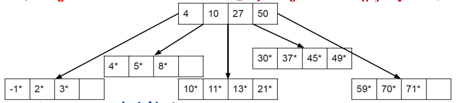
Extendible Hashing: Always double size till find suitable.



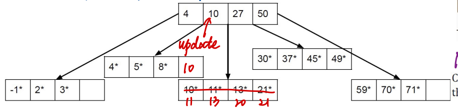
Trees.
A B+ tree is an M-way search tree with:
Perfectly balanced (leaf node at same depth)
Every node other than the root is at least half-full ($M/2-1 \le$ #keys $\le M-1$).
Every inner node with k keys has k+1 non-null children.



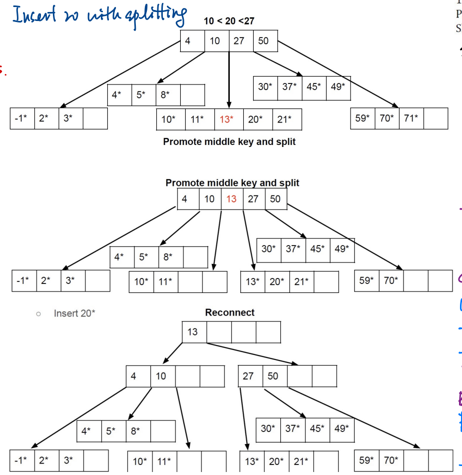
Insert: Find correct leaf node L.
Put data entry into L in sorted order.
If L has enough space, done!
Otherwise ① optionally, try to redistribute a key to sibling first
② 1. Split entries evenly, 2. Insert index entry pointing to L2 into parent
To split inner node, split entries evenly, but push up middle key
Delete: Start at root, find leaf L where entry belongs.
Remove the entry. If L is half-full, done!
If L has only M/2-1 entries
→ Try to redistribute, borrowing from sibling (adjacent node with same parent)
→ Redistribution fails, merge L and sibling
* If merge occurred, must delete entry (pointing to L or sibling) from parent of L.



Insert 20 with redistribution.

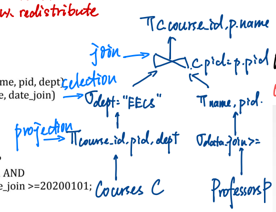


Insert 20 with splitting



* 记得更新 parent node 的 pointer 值
* 不同 parent 的 node 不可以 redistribute

Query Plan Trees
Schemas:
Courses(course_id, course_name, pid, dept)
Professors(professor_id, name, date_join)
Query:
SELECT C.course_id, P.name
FROM Courses C, Professors P
WHERE C.pid = P.professor_id AND C.dept = "EECS" AND P.date_join >= 20200101;

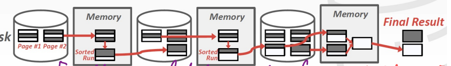


Sorting Algorithms.
If data fits in memory, we can use standard sorting algorithms (Quicksort)
If data does not fit in memory, aware of reading / writing speed.
External Merge Sort.
Divide and conquer algorithm that splits data into separate runs, sorts them individually, and then combines them into longer runs.
Sorted Run
A run is an ordered list of key/value pairs.
Key: The attributes to compare to compute the sort order
value: Tuple (early materialization) Record ID (late materialization)

2-way-merge sort

Pass #0
→ Read pages of the table into memory, one at a time
→ Sort the page into a run and write it back to disk

Pass #1,2,3,...
→ Recursively merge pairs of runs into runs twice as long
→ Uses three buffer pages (2 for input pages, 1 for output)



For N pages of data, Number of passes $1+\lceil \log_2 N \rceil$
Total I/O cost $2N \cdot$ (number of passes)

GENERAL EXTERNAL MERGE SORT.
Pass #0
→ Assume the DBMS has a finite number of B buffer pool pages to hold input and output data.
→ Read B pages of the table into memory at a time. Sort them and write back to disk.
→ Produce [N / B] sorted runs of size B

Pass #1,2,3,...
→ Merge B-1 runs (i.e., K-way merge)

Number of Passes $1+\lceil \log_{B-1} \lceil N/B \rceil \rceil$ Total I/O cost $2N \cdot$ (# of passes)

Consider sorting two datasets on a modern desktop machine with small memory. Assume that we have a 484 GB dataset and a 376 GB dataset.

For this question:
1 GB = 1024 MB, 1 MB = 1024 KB, 1 KB = 1024 B
Page size: 16 KB
Show your work for **BOTH** datasets to receive full credit.

1. Suppose RAM = 1MB, calculate passes and IO cost
$B = \frac{1MB}{16KB} = 64$ (pages).
For 484 GB database: $N = 484 \times 2^{20} KB / 2^4 KB = 484 \times 2^{16}$.
passes $= 1 + \lceil \log_{63} \frac{484 \times 2^{16}}{64} \rceil = 5$
IO $= 2N \cdot$ # passes $= 2 \times 5 \times 484 \times 2^{16}$ pages $= 2 \times 5 \times 484 \times 2^{16} \times 16KB = 4840GB$

Using B+ Tree for sorting
→ clustered: always better, → non-clustered: always worse
Hashing is an alternative to sorting

Phase 1: Partition
① Divide tuples into buckets on hash key ② Write them out to disk when full
Use a hash function h1 to split tuples into partitions on disk.
→ A partition is one or more pages that contain the set of keys with same hash value
→ Partitions are "spilled" to disk via output buffers. B buffers, B-1 partitions, 1 input
Phase 2: Rehash:
Build an in-memory hash table for each partition and compute the aggregation
For each partition on the disk: → Read it into memory and build an in-memory hash table on a second hash function h2.
→ Go through each bucket of this hash table to bring together matching tuples

Joining
M pages in table R, m tuples in R
N pages in table S, n tuples in S

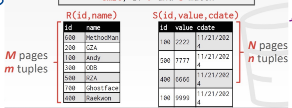
Nested Loop JOIN

```
foreach tuple r ∈ R:      Outer
  foreach tuple s ∈ S:    Inner
    emit, if r and s match
```

cost: $M + (m \cdot N)$

BLOCK NESTED LOOP JOIN

```
foreach B - 2 blocks bR ∈ R:
  foreach block bS ∈ S:
    foreach tuple r ∈ B - 2 blocks:
      foreach tuple s ∈ bS:
        emit, if r and s match
```



cost: $M + (\lceil \frac{M}{B-2} \rceil \cdot N)$
* for normal case, B=3
cost $= M + M \cdot N$
* if $B-2 \ge M$
cost $= M + N$ 2 read, 0 write

INDEX NESTED LOOP JOIN.

```
foreach tuple r ∈ R:
  foreach tuple s ∈ Index(ri = sj):
    emit, if r and s match
```

Assume the cost of each index probe is constant C
Cost: $M + m \cdot C$
* Pick the smaller table as the outer table
* Buffer as much of the outer table as possible.

SORT MERGE JOIN
Phase 1. Sort: ① Sort both tables on the join keys. ② Use the merge sort algorithm
Phase 2. Merge: ① Walk through the two sorted tables with cursors and emit matching tuples
② May need to backtrack depending on join type

```
sort R,S on join keys
cursorR ← Rsorted, cursorS ← Ssorted
while cursorR and cursorS:
  if cursorR > cursorS:
    increment cursorS
  if cursorR < cursorS:
    increment cursorR
  elif cursorR and cursorS match:
    emit
    increment cursorS
```

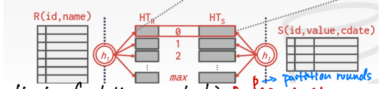
Backtrack case: dups in R, but S moves forward
Sort cost (R) $2M \cdot (1 + \lceil \log_{B-1} \lceil \frac{M}{B} \rceil \rceil)$
Sort cost (S) $2N \cdot (1 + \lceil \log_{B-1} \lceil \frac{N}{B} \rceil \rceil)$ } read & write
Merge cost: $(M+N)$. Total: Cost + Sort + Merge
Worst case for merging phase: Merge cost: $M \cdot N$ (worst) read
Join attribute of all the tuples in both relations contains the same value

Hash Join
Phase 1 Build: Scan the outer relation and populate a hash table using the hash function h1 on join attributes
Phase 2 Probe: Scan the inner relation and use h1 on each tuple to jump to a location in the hash table and find a matching tuple
Approach 1: Full Tuple
✓ Avoid having to retrieve the outer relation's tuple contents on a match
✗ Takes up more space
Approach 2: Tuple Identifier
→ Could be base table or intermediate output from query plan
✓ Ideal for column stores. Ideal for join selectivity is low

Grace Hash Join.
Build Phase: Hash both tables on join attribute into partitions
Probe Phase: Compare tuples in corresponding partitions.

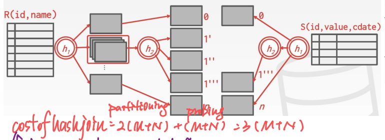
Perform nested loop join on each pair of matching buckets in the same level between R and S.

```
foreach tuple r ∈ bucketR,0:
  foreach tuple s ∈ bucketS,0:
    emit, if match(r, s)
```



How big of a table can we hash: $(B-1) \cdot (B-2) = N$.
For a table of N pages, we roughly need sqrt(N) buffers.
If hash distributes unevenly with fudge factor f > 1,
we need roughly sqrt(f N) buffers
2(M+N) Read & Write, M+N Read

Recursive Partitioning
RECURSIVE PARTITIONING



cost of hash join: 3(M+N) → 2(M+N) + (M+N)
Assume we have enough buffers.

Algorithm	IO Cost	Example
Simple Nested Loop Join	$M + (m \cdot N)$	1.3 hours
Block Nested Loop Join	$M + (M \cdot N)$	50 seconds
Block Nested Loop Join using B Buffers	$M + (\lceil M / (B-2) \rceil \cdot N)$	0.15 seconds
Index Nested Loop Join	$M + (m \cdot C)$	Variable
Sort-Merge Join	$M + N$ + (sort cost)	0.75 seconds
Hash Join	$3(M + N)$	0.45 seconds

Processing Models
Iterator (Volcano / Pipeline) Model
Each query plan operator implements a Next() function.
→ On each invocation, the operator returns either a single tuple or a null marker if there are no more tuples.
→ The operator implements a loop that calls Next() on its children to retrieve their tuples and then process them. * Allows for tuple pipelining

Materialization MODEL
Each operator processes its input all at once and then emits its output all at once.
→ The operator "materializes" its output as a single result.
→ The DBMS can push down hints (e.g., LIMIT) to avoid scanning too many tuples.
Better for OLTP because queries only access a small number of tuples at a time
→ Lower execution / coordination overhead
→ Fewer function calls

Vectorization Model.
Like the Iterator Model where each operator implements a Next() function, but...
Each operator emits a **batch** of tuples instead of a single tuple.
→ The operator's internal loop processes multiple tuples at a time.
→ The size of the batch can vary based on hardware or query properties.
Ideal for OLAP queries because it greatly reduces the number of invocations per operator
Allows for operators to more easily use vectorized (SIMD) instructions to process batches of tuples

Plan processing direction.
Approach #1: Top-to-Bottom
→ Start with the root and "pull" data up from its children.
→ Tuples are always passed with function calls.
Approach #2: Bottom-to-Top
→ Start with leaf nodes and push data to their parents.
→ Allows for tighter control of caches/registers in pipelines.

Access Methods for each page
SEQUENTIAL SCAN retrieve from the buffer pool, iterate over the tuple and check whether to include
Optimizations: Zone maps: Pre-computed aggregates for the attribute values in a page.
DBMS checks the zone map first to decide whether it wants to access the page.
Late materialization: DSM DBMS can delay stitching together tuples until the upper part of QP.

INDEX SCAN DBMS picks an index to find the tuples the query needs
If there are multiple indexes that the DBMS can use for a query:
→ Compute sets of Record IDs using each matching index.
→ Combine these sets based on the query's predicates (union vs. intersect).
→ Retrieve the records and apply any remaining predicates.
Set intersection can be done with bitmaps, hashmaps and Bloom filters

Expression Evaluation
DBMS represents a WHERE as an expression tree.
slow solution: evaluate the expression directly.
(JIT compilation)

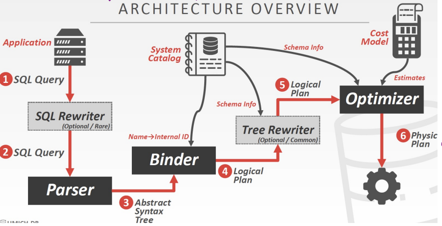
Modification Queries.
Operators that modify the database (INSERT, UPDATE, DELETE) are responsible for checking constraints and updating indexes.
UPDATE/DELETE:
→ Must keep track of previously seen tuples.

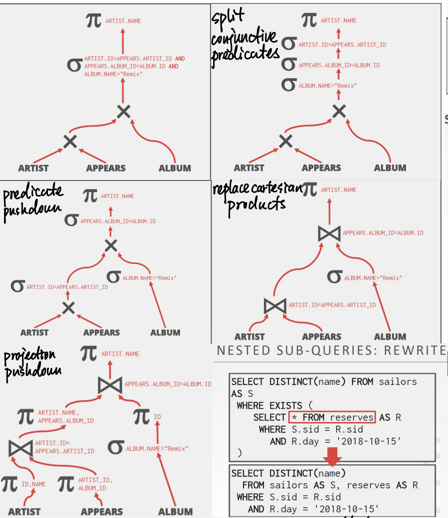
Query Optimization

Heuristic/Rules
-> Rewrite the query to remove inefficient operations
-> Need to examine catalog, but not data.

Cost-based search
-> Use a model to estimate the cost of executing a plan
-> Evaluate multiple equivalent plans for a query and pick the lowest cost

ARCHITECTURE OVERVIEW





NESTED SUB-QUERIES: REWRITE

```
SELECT DISTINCT(name) FROM sailors
AS S
 WHERE EXISTS (
   SELECT * FROM reserves AS R
    WHERE S.sid = R.sid
      AND R.day = '2018-10-15'
 )
```

```
SELECT DISTINCT(name)
  FROM sailors AS S, reserves AS R
 WHERE S.sid = R.sid
   AND R.day = '2018-10-15'
```

```
SELECT MAX(rating) FROM sailors
```

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = ###

 GROUP BY S.sid
HAVING COUNT(*) > 1
```

Outer Block

Impossible / unnecessary predicates $1=0$ / $1=1$

Join Elimination
From A AS A1 Join A AS A2 ON A1.id = A2.id
=> Select * From A

Merging Predicates
Where val Between 1 AND 100
OR val Between 50 AND 150

Choice #1: Physical Costs
→ Predict CPU cycles, I/O time, cache misses, RAM consumption, pre-fetching, etc... (dominant)
→ Depends heavily on hardware.

Choice #2: Logical Costs
→ Estimate result sizes per operator.
→ Complexity of the operator algorithm implementation.
→ # sequential I/Os, # random I/Os, # arithmetics.

Cost-based Search and Statistics

For each relationship R, the DBMS maintains the following:
-> N_R: Number of tuples in R.
-> $V(A,R)$: Number of distinct values for attribute A.
Selection cardinality $SC(A,R)$: the avg # records with a value for an attribute A: $N_R / V(A,R)$ (Assume uniformity)
Logical costs: Estimate the result size of every operator in query plan
Use the estimated result of child as input into parent.
Result size estimation: hard to compute complex predicates

Selectivity: the fraction of tuple that qualify ≈ probability
$sel(P_1 \wedge P_2) = sel(P_1) \cdot sel(P_2)$ — assume $P_1 P_2$ independent
$sel(P_1 \vee P_2) = sel(P_1) + sel(P_2) - sel(P_1) \cdot sel(P_2)$

Result Size Estimation for JOINS

$\frac{N_R \cdot N_S}{\ }$
estSize = $\max(V(A,S), V(A,R))$

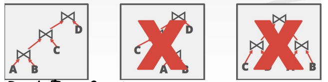
Selectivity assumptions:
Uniform Data: Distribution is the same
Independent Predicates: Predicates are independent
Inclusion Principle: The domain of join keys overlap such that each key in inner relation also exist in the outer

Equalwidth histogram: buckets of same width
Equaldepth histogram: occurrence of bucket same
Other approaches: sampling and sketches

Query Optimization
After performing rule-based rewriting, DBMS will enumerate different plans for the query and estimate costs
① single relation ② multiple relation ③ nested subqueries

Single relation -> sequential scan -> B-search -> index scan

Fundamental decision in System R: Only consider left-deep join trees.
→ Many modern DBMSs still make this assumption.

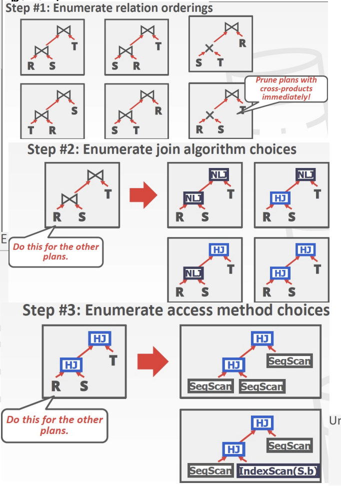


Brute Force Search

Step #1: Enumerate relation orderings

Step #2: Enumerate join algorithm choices

Step #3: Enumerate access method choices



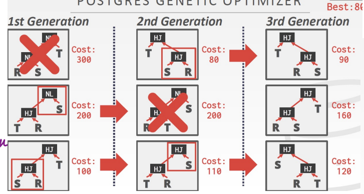
POSTGRES OPTIMIZER

Examines all types of join trees
→ Left-deep, Right-deep, bushy

Two optimizer implementations:
→ Traditional Dynamic Programming Approach
→ Genetic Query Optimizer (GEQO)

Postgres uses the traditional algorithm when # of tables in query is **less** than 12 and switches to GEQO when there are 12 or more.

POSTGRES GENETIC OPTIMIZER



Concurrency Control Theory

Transaction is the execution of a sequence of one or more operations (eg SQL queries) on a database to perform high-level function

Strawman System
Execute each txn one-by-one as they arrive at DBMS.
-> One and only one txn can be running at the same time

Before a txn starts, copy the entire database to a new file.
-> If txn completes successfully, overwrite the original one
-> If txn fails, discard the dirty copy

concurrent execution of indep transactions
-> Reduced waittime -> Better utilization

Formal Definition
Database: A fixed set of named data objects
Transaction: A sequence of read and write operations

Correctness Criteria: ACID
Atomicity: All actions in the txn happen, or none happen
Consistency: If txn is consistent, DB starts consistent -> result consistent
Isolation: Execution of one txn is isolated from other txn
Durability: If a txn commits, its effects persist.

Atomicity
Approach 1: logging Approach 2: Shadow Paging

Consistency
Database Consistency: The database accurately models the real world and follows integrity constraints. Transaction consistency: If the database is consistent before transaction starts, it will be consistent after.

Isolation: DBMS achieves concurrency by interleaving the actions
Pessimistic: Don't let problems arise. Optimistic: Deal with conflicts
Correctness: If schedule is equivalent to some serial execution

Serial Schedule

→ A schedule that does not interleave the actions of different transactions.

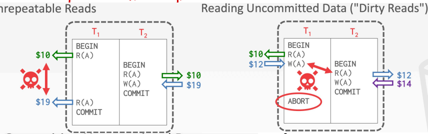
Equivalent Schedules

→ For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
→ Doesn't matter what the arithmetic operations are!

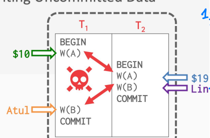
Serializable Schedule: A schedule that is equivalent to some serial execution of transactions. If each transaction preserves consistency, every serializable schedule preserves consistency.

Unrepeatable Reads

Reading Uncommitted Data ("Dirty Reads")



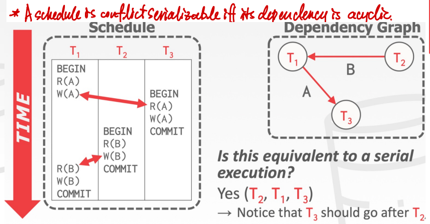
Overwriting Uncommitted Data



Conflict equivalent:
1. Involve same actions of the same transaction. Every pair of conflicting actions is ordered the same way.

Dependency Graphs: Edge from Ti to Tj iff
-> An operation Oi of Ti conflicts with an operation Oj and Tj
-> Oi appears earlier in the schedule than Oj

* A schedule is conflict serializable iff its dependency graph is acyclic.



Is this equivalent to a serial execution?
Yes (T_2, T_1, T_3)
→ Notice that T_3 should go after T_2, although it starts before it!

Durability
-> No torn updates -> use logging or shadow paging
-> No changes from failed transactions

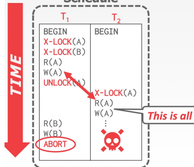
Locks

S-lock shared locks for reads
X-lock exclusive locks for writes

Phase 1 Growing
-> Each txn requests the locks that it needs from the lock manager
-> The lock manager grants/denies lock requests

Phase 2 Shrinking
-> The txn is allowed to release locks that it previously acquired
It cannot acquire new locks

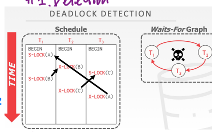
Cascading Aborts



This is a permissible schedule in 2PL, but the DBMS has to also abort T_2 when T_1 aborts.
→ Any information about T_1 cannot be "leaked" to the outside world.

Strong Strict 2PL: only release lock when commit/abort

Dead Lock
#1 Detection



Handling:
-> choose a victim, rollback to break the cycle
-> Victim txn will restart / abort

Victim selection
-> Age -> Process
-> # locked -> # txns.

#Prevention When a txn tries to acquire a lock that is held by another txn, the DBMS kills one of them to prevent DL
Older timestamps = High Priority

Wait-Die: -> If the requesting txn has higher priority than holding txn, the requesting txn waits for holding txn.
-> Otherwise requesting txn aborts.
Wound-Wait -> If requesting has higher priority, the holding txn aborts and releases lock -> Otherwise requesting txn waits.

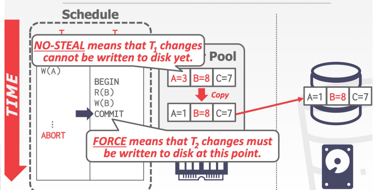
Crash recovery
Actions during normal txn processing to ensure DBMS can recover
Actions after failure to recover DB to a state of ACD

Storage Types
Volatile Storage DATA does not persist after failure (DRAM, SRAM)
Non-volatile storage DATA persists after failure (HDD, SDD)
Stable Storage A non-existent form of non-volatile storage that survives all possible failures scenarios.

Transaction Failures logical errors / Internal State Errors (deadlock)
System Failures Software Failure / Hardware Failure
Storage Media Failures Non-repairable hardware failure

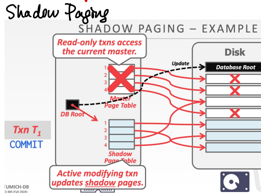
Steal: Whether DBMS allows an uncommitted txn to overwrite the most recent committed value of an object in non-volatile storage
Force: Whether DBMS requires that all updates made by a txn are reflected on non-volatile storage before txn can commit

NO-STEAL + FORCE



easiest to implement: No redo, No undo

Shadow Paging



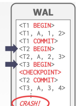
Master: only changes from committed txns.
Shadow: Temp DB with uncommitted changes
commits: shadow -> master
No steal / Force

Write ahead log (Steal + no-force)

-> Write a <BEGIN> record to the log
-> When txn finishes, write a <COMMIT>.
-> Make sure all log records are flushed before sending Ack

contains TXN ID, Object ID, Before Value, After Value.

Checkpoints



Any txn that committed before the checkpoint is ignored (T_1).
T_2 + T_3 did not commit before the last checkpoint.
→ Need to redo T_2 because it committed after checkpoint.
→ Need to undo T_3 because it did not commit before the crash.

ARIES MAIN IDEAS

Write-Ahead Logging:
→ Any change is recorded in log on non-volatile storage before the database change is written to disk.
→ Must use STEAL + NO-FORCE buffer pool policies.

Repeating History During Redo:
→ On restart, retrace actions and restore database to exact state before crash.

Logging Changes During Undo:
→ Record undo actions to log to ensure action is not repeated in the event of repeated failures.

Log Sequence Numbers

Name	Where	Definition
flushedLSN	Memory	Last LSN in log on disk
pageLSN	page$_x$	Newest update to page$_x$
recLSN	page$_x$	Oldest update to page$_x$ since it was last flushed
lastLSN	T_i	Latest record of txn T_i
MasterRecord	Disk	LSN of latest checkpoint

Each page contains a page LSN.
-> The LSN of most recent update to that page.
System keeps track of flushed LSN
-> The max LSN flushed.
Before page x can be written to disk, we must flush the log at least to
-> pageLSN ≤ flushed LSN
Update page LSN every time a txn modifies a record.
Update flushedLSN in memory everytime DBMS writes out WAL buffer to disk

TRANSACTION COMMIT

Write COMMIT record to log.

All log records up to txn's COMMIT record are flushed to disk.
→ Log flushes are sequential, synchronous writes to disk.
→ Many log records per log page.

When the commit succeeds, write a special TXN-END record to log.
→ This does not need to be flushed immediately.

TRANSACTION ABORT

Aborting a txn is a special case of the ARIES undo operation applied to only one txn.

We need to add another field to our log records:
→ prevLSN: The previous LSN for the txn.
→ This maintains a linked-list for each txn that makes it easy to walk through its records.

COMPENSATION LOG RECORDS

A **CLR** describes the actions taken to undo the actions of a previous update record.

It has all the fields of an update log record plus the undoNext pointer (the next-to-be-undone LSN).

CLRs are added to log records but the DBMS does not wait for them to be flushed before notifying the application that the txn aborted.

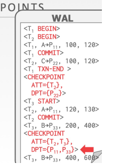
Abort Algorithm
-> First write an ABORT record to log for the txn.
-> then play back the txn's updates in reverse order.
For each update record: -> Write a CLR entry to the log.
-> Restore the old value.
-> At end, write a TXN-END log record.
* CLR never needs to be undone

Fuzzy CKPT

Pause modifying txns while the DBMS takes CKPT
-> prevents queries from acquiring write locks on data
-> Don't have to wait until all txns finish before taking CKPT

Record: Active Transaction table (ATT) Dirty Page Table (DPT)
ATT: -> txn ID -> status -> last LSN
Txn Status codes: R running C committing U candidate for undo
DPT keeps track of which pages in the buffer pool contain changes from transactions that haven't been flushed to disk.

SLIGHTLY BETTER CHECKPOINTS

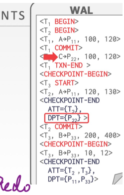


At the first checkpoint, assuming P_{11} was flushed, T_2 is still running and there is only one dirty page (P_{22}).

At the second checkpoint, assuming P_{22} was flushed, T_2 and T_3 are active and the dirty pages are (P_{11}, P_{33}).

Fuzzy Checkpoints
DBMS allows active txns to continue the run while the system writes log records for checkpoint

FUZZY CHECKPOINTS



The *LSN* of the CHECKPOINT-BEGIN record is written to the database's MasterRecord entry on disk when the checkpoint successfully completes.

Any txn that starts after the checkpoint is excluded from the ATT in the CHECKPOINT-END record.

Aries #Analysis #Redo #Redo
Analysis: Figure out which txns committed or failed since the checkpoint
Redo: Repeat all actions. Undo: Reverse effects of failed txn

ANALYSIS PHASE

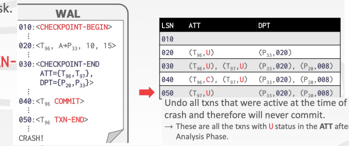
Scan log forward from last successful checkpoint.

If you find a TXN-END record, remove its corresponding txn from ATT.

All other records:
→ Add txn to ATT with status UNDO.
→ On commit, change txn status to COMMIT.

For UPDATE records:
→ If page P not in DPT, add P to DPT, set its recLSN=LSN.

ANALYSIS PHASE EXAMPLE



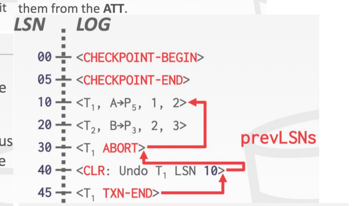
To redo an action:
→ Reapply logged action.
→ Set pageLSN to log record's LSN.
→ No additional logging, no forced flushes!

Undo all txns that were active at the time of crash and therefore will never commit.
→ These are all the txns with U status in the ATT after Analysis Phase.

Process them in reverse *LSN* order using the lastLSN to speed up traversal.

Write a CLR for every modification.

At the end of Redo Phase, write TXN-END log records for all txns with status C and remove them from the ATT.



FULL EXAMPLE