CS3223 FINALS CHEATSHEET

L4: Query Evaluation: Sorting & Selection: Used to produce a sorted table of results, bulk loading a B+tree index, implement other algebra operators such as projection & join.

External Merge Sort: *Suppose file size of N pages and B buffer pages.*

Pass 0: Creation of sorted runs
- Read in and sort B pages at a time
- Number of sorted runs created = $\lceil N/B \rceil$
- Size of each sorted run = B pages (except possible the last run)

Pass i, $i \geq 1$: Merging of sorted runs
- Use B-1 Buffer pages for input & one buffer page for output
- Perform (B-1) way merge

N_0= number of sorted runs created in pass 0 = $\lceil N/B \rceil$
Total number of passes = $\lceil \log_{B-1}(N_0) \rceil + 1$
Total number of I/O = $2N(\lceil \log_{B-1}(N_0) \rceil + 1)$ //each pass reads & writes N pages

Optimization with Blocked I/O: *R/W in units of buffer blocks of b pages.*
- Allocate one block (b pages) for output
- Remainder space can ammodate $\left\lfloor \frac{B-b}{b} \right\rfloor$ blocks for input
- Can merge at most $\left\lfloor \frac{B-b}{b} \right\rfloor$ sorted runs in each merge pass

N_0= number of initial sorted runs = $\lceil N/B \rceil$
F = number of runs that can be merged at each merge pass = $\left\lfloor \frac{B}{b} \right\rfloor - 1$
Number of passes = $\lceil \log_F N_0 \rceil + 1$

Sorting using B+ trees: When table to be sorted has a B+ tree index on sorting attribute → Format 1: sequentially scan leaf pages of B+ tree; Format 2/3: Sequentially scan leaf pages and for each page visited, retrieve data records using RIDs. An index is a **clustered index** if the order of its data entries is 'close to' the order of the data records. An **index using format 1 is a clustered index**. There is at most one clustered index for each relation.

Access Path: refers to a way of accessing data records/entries. **Table scan**: scan all data pages, **Index scan**: scan index pages, **Index intersection**: combine results from multiple index scans (e.g. intersect, union).
Selectivity of access path = number of index & data pages retrieved to access data records. Most selective path retrieves least pages. Index I is a **covering index** for query Q if all attributes references in Q are part of the key of I
[Q can be evaluated using I without any RID lookup → **index-only plan**].

CNF Predicates: A **term** is of the form $R.A\ op\ c$ or $R.A_i\ op\ R.A_j$. A **conjunct** consists of one or more terms connected by ∨. A conjunct that contains ∨ is said to be **disjunctive**. A **conjunctive normal form (CNF)** predicate consists of one or more conjuncts connected by ∧.

B+ Tree matching predicates: B+ tree index $I = (K_1, \ldots, K_n)$. Non-disjunctive CNF predicate p. I matches p if p is of the form : $(\boldsymbol{K_1 = c_1}) \wedge \ldots \wedge (\boldsymbol{K_{i-1} = c_{i-1}}) \wedge (\boldsymbol{K_i\ op_i\ c_i}), \boldsymbol{i \in [1, n]}$,where $(K_1, \ldots, K_i)$ is a prefix of the key of I, and there is atmost one non-equality comparison operator which must be on the last attribute of the prefix K_i. **Hash Index matching predicates:** I matches p if p is of the form : $(\boldsymbol{K_1 = c_1}) \wedge (\boldsymbol{K_2 = c_2}) \wedge \ldots \wedge (\boldsymbol{K_n = c_n})$

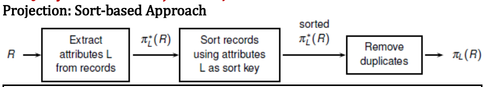
Primary Conjuncts: subset of conjuncts in p that I matches. **Covered Conjuncts:** subset of conjuncts in p covered by I. Each attribute in covered conjuncts appears in the key of I. $primary\ conjunct \subseteq convered\ conjuncts$.

Cost of B+tree Index evaluation of p: Let p' = primary conjuncts, pc = covered conjuncts: Navigate internal nodes to locate first leaf page
$Cost_{internal} = \lceil \log_F \left(\left\lceil \frac{||R||}{b_d} \right\rceil \right) \rceil$.Scan leaf pages to access all qualifying entry
$Cost_{leaf} = \left\lceil \frac{||\sigma'_p(R)||}{b_d} \right\rceil$. Retrieve qualified data records via RID lookup
$Cost_{RID} = 0\ or\ \left|\left|\sigma_{pc}(R)\right|\right|$, cost is zero if I is a covering or format-1 index.
Cost of RID lookups could be reduced by first sorting the RIDs: $\left\lceil \frac{||\sigma_{pc}(R)||}{b_d} \right\rceil \leq$
$Cost_{rid} \leq \min\left\{\left|\left|\sigma_{pc}(R)\right|\right|, |R|\right\}$

Cost of hash index evaluation of p:
Format-1 Index: at least $\left\lceil \frac{||\sigma'_p(R)||}{b_d} \right\rceil$. **Format-2 Index**: at least $\left\lceil \frac{||\sigma'_p(R)||}{b_d} \right\rceil$ + cost to retrieve data records = 0 if I is covering index, otherwise $\left|\left|\sigma'_p(R)\right|\right|$.

L5: Query Evaluation: Projection & Join

Projection: Sort-based Approach



Step 1:	Cost to scan record = $\lvert R \rvert$ Cost to output temporary result = $\lvert \pi_L^*(R) \rvert$
Step 2:	Cost to sort records = $2\lvert \pi_L^*(R) \rvert(\log_m(N_0) + 1)$ N_0 = number of initial sorted runs, $m = B - 1$ = merge factor
Step 3:	Cost to scan records = $\lvert \pi_L^*(R) \rvert$

Hash-Based Approach: Build a main-memory hashtable to detect & remove duplicates. Phase 1 **Partitioning phase**: partitions R into $R_1, R_2, \ldots, R_{B-1}$.
- Hash on $\pi_L(t)$ for each tuple $t \in R$.
- $R = R_1 \cup R_2 \cup \ldots \cup R_{B-1}$
- $\pi_L^*(R_i) \cap \pi_L^*(R_j) = \emptyset$ for each pair R_i & $R_j, i \neq j$

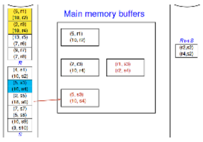
Phase 2 **Duplicate elimination phase**: eliminates duplicates from each $\pi_L^*(R_i)$
$\pi_L(R)$ = duplicate free union of $\pi_L(R_1), \pi_L(R_2) \ldots, \pi_L(R_{B-1})$

Partitioning Phase: Use **one buffer for input** & **(B-1) buffers for output**. Read R one page at a time into input buffer. For each tuple t in input tuple: project out unwanted attributes from t to form t'. Apply hash function h on t' to distribute t' into 1 output buffer. Flush the output buffer to disk whenever buffer is full. **Duplicate Elimination Phase:** For each partition R_i, initialize an in-memory hashtable. Read $\pi_L^*(R_i)$ one page at a time, for each tuple t read, hash t into bucket B_j with hash function $h'(h' \neq h)$. Insert t into B_j if $t \notin B_j$. Write out tuples in hashtable to results. **Partition Overflow:** Partition overflow problem: Hash table for $\pi_L^*(R_i)$ is larger than available memory buffers. **Solution**: recursively apply hash-based partitioning to the overflowed partition.
Hash-Based Approach: Analysis: Approach is effective if B is large relative to |R|. Assuming that h distributes tuples in R uniformly, Each R_i has $\frac{|\pi_L^*(R)|}{B-1}$ pages. Size of hash table for each $R_i = \frac{|\pi_L^*(R)|}{B-1} \times f$. Fudge factor is a small value that increases the number of partitions. To avoid partition overflow, $B > \frac{|\pi_L^*(R)|}{B-1} \times f$ or approximately $B > \sqrt{f \times |\pi_L^*(R)|}$.
Assume there's no partition overflow,
Cost of partitioning phase: $|R| + |\pi_L^*(R)|$ *Read |R|, output projected R**
Cost of duplicate elimination phase: $|\pi_L^*(R)|$ *Read projected R**
Total cost = $|R| + 2|\pi_L^*(R)|$

Sort-Based vs Hash-Based: Sort-based output is sorted. Its good if there are many duplicates or if distribution of hashed values are non-uniform. If $B > \sqrt{|\pi_L^*(R)|}$,
- Number of initial sorted runs $N_0 = \left\lceil \left(\frac{|R|}{B}\right) \right\rceil \approx \sqrt{|\pi_L^*(R)|}$
- Number of merging passes = $\log_{(B-1)} N_0 \approx 1$
- Sort-based approach requires 2 passes for sorting
- Both hash-based & sort-based methods have same IO cost.

Join Algorithms: BNLJ, INLJ, SMJ, HJ

Tuple Based Nest Loop Join:
IO Cost: $|R| + ||R|| \times |S|$
Page Based Nested Loop Join:
IO Cost: $|R| + |R| \times |S|$
Block Nested Loop Join
Buffer space allocation: 1 page for S, 1 page for output, remaining for R
IO Cost: $|R| + \left(\left\lceil \frac{|R|}{B-2} \right\rceil \times |S| \right)$



Index Nested Loop Join
Precondition: there is an index on the join attribute(s) of inner relation
Idea: foreach r in R: use r to probe S' index to find matching tuples
IO Cost = $|R| + ||R|| \times J$, $J = idx\ internal\ nodes + idx\ leaf\ nodes$

Sort Merge Join
Idea: sort both relations based on join attributes & merge them
A sorted relation R consists of partitions of R_i of records where $r, r' \in R_i$ iff r and r' have the same values for the join attribute(s)
Search for matching partitions by advancing ptr that is pointing to "smaller" tuple. Need to remember position of first tuple in matching S-partition to enable rewinding of S-pointer.
IO Cost = Cost to sort R + Cost to sort S + Merging Cost
Cost to sort R = $2|R|(\log_m(N_R) + 1)$, **Cost to sort S** = $2|S|(\log_m(N_S) + 1)$
If each S partition scanned atmost once during merge:
Merging Cost = $|R| + |S|$
Worst case occurs when each tuple of R requires scanning entire S:
Merging Cost = $|R| + ||R|| \times |S|$

Sort Merge Join Optimization
Create sorted runs of R; merge sorted runs of R partially. Create sorted runs of S; merge sorted runs of S partially. Merge remaining sorted runs of R & S and join them at the same time.
If $B > \sqrt{2|S|}$: Number of initial sorted runs of S < $\sqrt{|S|/2}$
Total number of initial sorted runs of R&S < $\sqrt{2|S|}$
One pass sufficient to merge & join initial sorted runs R&S
IO Cost = $3(|R| + |S|)$

Grace Hash Join, $R \bowtie_{R.A=S.B} S$: Consists of three phases:
1. Partition R into $R_1, \cdots, R_k$
2. Partition S into $S_1, \cdots, S_k$
3. Probing phase: probes each R_i with S_i
  - Read R_i to build a hash table
  - Read S_i to probe hash table

R is called the **build relation** & S is called the **probe relation.** Choose smaller relation to be build relation.

Partitioning (building phase):	Probing (matching) phase
init a hash table T with k buckets for each tuple r ∈ R insert r into bucket h(r.A) of T write each bucket Ri of T to disk init a hash table T with k buckets for each tuple s ∈ S insert s into bucket h(s.B) of T write each bucket Si of T to disk	for i = 1 to k init a hash table T for each tuple r in partition Ri insert r to bucket h'(r.A) of T for each tuple s in partition Si for each tuple r in bucket h'(s.B) of T: if r and s matches output (r ,s)

Grace Hash Join: Analysis
To minimize size of each partition of Ri, Set k = B−1 given B buffer pages
Assuming uniform hashing distribution,
- size of each partition Ri is $ceil\left(\frac{|R|}{B-1}\right)$
- size of hash table for Ri is $\frac{f \times |R|}{B-1}$
- During probing phase, $B > \frac{f \times |R|}{B-1} + 2$ (with one input buffer for Si & one output buffer)
- Approximately, $B > \sqrt{f \times |R|}$

Partition overflow problem: Hash table for Ri does not fit in memory
Solution: recursively apply partitioning to overflow partitions
I/O cost = Cost of partitioning phases + Cost of probing phase
I/O cost = 3(|R|+|S|) if there's no partition overflow problem

General Join Conditions
Multiple equality-join conditions: Example: (R.A = S.A) and (R.B = S.B)
- **Index Nested Loop Join**: use index on all or some of join attributes
- **Sort-Merge Join**: need to sort on combination of attributes
- Other algorithms essentially unchanged

Inequality-join conditions: Example: (R.A < S.A)
- **Index Nested Loop Join**: requires a B+-tree index
- **Sort-Merge Join & Hash-based Joins**: not applicable
- Other algorithms essentially unchanged

L6: Query Evaluation & Optimization

Query Evaluation Approaches: Materialized Evaluation: op is evaluated only when each of its operand has been completely evaluated or materialized. Intermediate results materialized to disk. **Pipelined Evaluation:** Output producted by a op is passed directly to its parent op. Exec of ops interleaved.

Pipelined Evaluation: A operator O is a blocking op if O may not be able to produce any output until it has received all the input tuples from its child op(s). E.g. of blocking ops: ext merge sort, sort-merge join, grace hash join.

Pipelined Evaluation: Iterator Interface: Top-down, demand-driven approach
1. **open**: initializes state of iterator: allocates resources for operation, initializes operator's arguments (e.g., selection conditions)
2. **getNext**: generates next output tuple, Returns null if all output tuples have been generated
3. **close**: deallocates state information

Relational Algebra Equivalence: (same rules for binary ops apply to ⋈,∪ etc)
Commutativity of binary ops: $R \times S \equiv S \times R$,
Associativity of binary ops: $(R \times S) \times T \equiv R \times (S \times T)$
Idempotence of unary ops: $\pi_{L'}(\pi_L(R)) = \pi_{L'}(R)$ if $L' \subseteq L \subseteq attributes(R)$

$$\sigma_{P1}\left(\sigma_{p2}(R)\right) \equiv \sigma_{p_1 \wedge p_2}(R)$$

Commutating selection w/ projection: $\pi_L\left(\sigma_p(R)\right) \equiv \pi_L\left(\sigma_p\left(\pi_{L \cup attr(p)}(R)\right)\right)$
Commutating selection w/ binary ops: $\sigma_p(R \times S) \equiv \sigma_p(R) \times S$ if $if\ attr(p) \subseteq attr(R)$.
Commutating projection w/ binary ops: let $L = L_R \cup L_S$
$\pi_L(R \times S) \equiv \pi_{L_R}(R) \times \pi_{L_S}(S)$

System R Optimizer: Uses enhanced dynamic programming approach that considers sort order of query plan's output. Maintains optPlan(Si, oi) instead of optPlan(Si). oi captures sort order of output produced by query plan wry Si. oi either null if output is unordered or a seq of attributes. optPlan(Si, oi)=cheapest query plan for relations Si with output ordered by oi if oi!=null

Cost Estimation of a Plan: uniformity, independence, inclusion assumptions
Join Selectivity: Consider *R JOIN S ON R.A=S.B*
Inclusion Assumption: If $\left|\left|\pi_A(R)\right|\right| \leq \left|\left|\pi_B(S)\right|\right|$, then $\pi_A(R) \subseteq \pi_B(S)$

$$rf(R.A = S.B) \approx \frac{1}{\max\{||\pi_A(R)||, ||\pi_B(S)||\}}$$

E.g. Rf(R.dept = S.dept) ≈ $\frac{1}{\max\{\ldots\}} = \frac{1}{4}$. $||Q|| \approx ||R|| \times ||S|| \times 1/4$

Equiwidth histograms: Each bucket has (almost) equal number of values
Equidepth histograms: Each bucket has (almost) equal number of tuples
Sub-ranges of adjacent buckets might overlap

Equiwidth Histogram

Bucket No	Value Range	No. of Tuples
1	[0,2]	8
2	[3,5]	4
3	[6,8]	15
4	[9,11]	3
5	[12,14]	15

Equidepth Histogram

Bucket No	Value Range	No. of Tuples
1	[0,3]	9
2	[4,6]	9
3	[6,8]	9
4	[9,13]	9
5	[14,14]	9

Query Q1: $\sigma_{A=6}(R), \left|\left|Q_1\right|\right| = 8$ **Query Q2:** $\sigma_{A\in[7,12]}(R), \left|\left|Q_1\right|\right| = 12$
Without Histogram: ||Q1|| ≈ 45/15;
Equiwidth Histogram: ||Q1|| ≈ 15/3
Equidepth Histogram: ||Q1|| ≈ (1/3 × 9)+(1/3 × 9)
Without Histogram: ||Q2|| ≈ 45/15 * 6
Equiwidth Histogram: ||Q2|| ≈ (2/3 × 15)+3+(1/3 × 15)
Equidepth Histogram: ||Q2|| ≈ (2/3 × 9)+(4/5 × 9)
MCV = Most Common Values: Separately keep track of the frequencies of the top-k most common values and exclude MCV from histogram's buckets

Query Q1 : $\sigma_{A=6}(R)$, ||Q1|| = 8
Query Q2 : $\sigma_{A\in[7,12]}$, ||Q2||= 12

MCV (k=2)

Value	No. of Tuples
6	8
14	9

Equidepth Histogram (with 3 buckets)

Bucket No	Value Range	No. of Tuples
1	[0,3]	9
2	[4,8]	10
3	[9,14]	9

Equidepth histogram: ||Q1|| ≈ (1/3 × 9)+(1/3 × 9)
Equidepth histogram with MCV: ||Q1|| ≈ 8
Equidepth histogram: ||Q2|| ≈ (2/3×9)+(4/5×9)
Equidepth histogram with MCV: ||Q2|| ≈ (2/(5−1)×10)+(4/(6−1)×9)

L7: Transaction Management

$\boldsymbol{T_j}$ **reads** $\boldsymbol{O}$ **from** $\boldsymbol{T_i}$ in a schedule S if the last write action on O before $R_j(O)$ in S is $W_i(O)$. $\boldsymbol{T_j}$ **reads from** $\boldsymbol{T_i}$ if T_j has read some object from T_i.
$\boldsymbol{T_i}$ **performs the final write on** $\boldsymbol{O}$ in a schedule S if the last write action on O in S is $W_i(O)$. An interleaved Xact execution schedule is **correct** if it is equivalent to some serial schedule over the same set of Xacts. Two actions on the same object **conflict** if: 1. atleast one of them is a write action, and 2. the actions are from different Xacts.

S and S' are **view equivalent** ($S \equiv_v S'$) if they satisfy all the conditions:
- If T_i reads A from T_j in S, then T_i must also read A from T_j in S'
- For each data object A, the Xact (if any) that performs the final write on A in S must also perform the final write on A in S'

S is a **view serializable schedule** (**VSS**) if S is view equivalent to some serial schedule over the same set of Xacts.

1. Dirty read problem (due to WR conflicts)
- T2 reads an object that has been modified by T1 and T1 has not yet committed → T2 could see an inconsistent DB state!

2. Unrepeatable read problem (due to RW conflicts)
- T2 updates an object that T1 has previously read and T2 commits while T1 still in prog. → T1 could get different value if it reads the object again

3. Lost update problem (due to WW conflicts)
- T2 overwrites the value of an object that has been modified by T1 while T1 is still in progress → T1's update is lost!

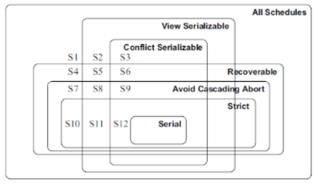
S and S' (over same set of Xacts) are said to be **conflict equivalent** ($S \equiv_c S'$) if they order every pair of conflicting actions of two committed Xacts in the same way. A schedule is a **conflict serializable schedule (CSS)** if it is conflict equivalent to a serial schedule over the same set of Xacts.
Conflict serializability graph for S is a directed graph G=(V,E) s.t.:
- V contains a node for each committed Xact in S
- E contains (T_i, T_j) if an action in T_i precedes and conflicts with one of T_j 's actions

T1: S is conflict serializable iff its conflict serializability graph is acyclic
T3: If S is view serializable and S has no blind writes, then S is also conflict serializable (Write on object O by T_i is called a **blind write** if T_i did not read O prior to the write)

For correctness, if T_i has read from T_j, then T_i must abort if T_j aborts. Recursive aborting process is known as **cascading abort**.

S is said to be a **recoverable schedule** if for every Xact T that commits in S, T must commit after T' if T reads from T'. (If T1 RF T2, T1 must commit after T2). While recoverable schedules guarantee that commited Xacts will not be aborted, cascading aborts of active Xacts are possible. To avoid cascading aborts, DBMS must permit reads only from commited Xacts. A schedule S is a **cascadeless schedule** if whenever T_i reads from T_j in S, $Commit_j$ must precede this read action.

Recovery using Before-Images: Efficient approach to undo actions of aborted Xacts is to restore before-images for writes. To enable use of before-images for recovery, use strict schedules.
A schedule S is a **strict schedule** if for every $W_i(O)$ in S, O is not read or written by another Xact until T_i either aborts or commits. Performance Tradeoff: Recovery (using bef-img) is more efficient, concurrent execs become more restrictive.



L8: Concurrency Control
Shared(S) locks for R(O)
Exclusive(X) locks for W(O)

Lock Requested	Lock Held		
	-	S	X
S	√	√	×
X	√	×	×

To R(O), request for S/X lock. To Update O, request for X lock. Lock request is granted on O if requesting lock mode is compatible with lock mode of existing locks on O. If T's lock request is not granted on O, T becomes blocked and added to O's request queue. When lock is released on O, lock manager checks request in the queue. When a Xact commits/aborts, all its locks are released & T is removed from any request queue it's in.

Two Phase Locking (2PL) Protocol:
- To read O, T must hold a S-lock or X-lock on O
- To write O, a Xact must hold a X-lock on O
- Once T releases a lock, T can't request any more locks

Growing phase: before releasing 1st lock
Shrinking phase: after releasing 1st lock

Strict Two Phase Locking (Strict 2PL) Protocol:
- To read O, T must hold a S-lock or X-lock on O
- To write O, a Xact must hold a X-lock on O
- T must hold on to locks until Xact commits or aborts

Theorem 1: 2PL Schedules are conflict serializable
Theorem 2: Strict 2PL schedules are strict & conflict serializable
Deadlocks: cycle of Xacts waiting for locks to be released by each other.
Dealing with deadlocks: deadlock prevention & detection.

Waits-for-graph (WFG): Nodes represent active Xacts. Add an edge $T_i \to T_j$ if T_i is waiting for T_j to release a lock. Lock manager will: add an edge when it queues a lock req, update edge when it grants a lock req. If WFG has a cycle, deadlock is detected. Break a deadlock by aborting a Xact in the cycle. Alternative to WFG: timeout mechanism.

Deadlock Prevention: Assume older xacts (smaller timestamp) have higher priority than younger Xacts. Suppose Ti requests for a lock that conflicts with a lock held by Tj.

Wait-die policy:
- lower-priority Xacts never wait for higher priority Xacts;
- non-preemptive: only a Xact requesting for a lock can get aborted
- a younger Xact may get repeatedly aborted
- a Xact that has all the locks it needs is never aborted

Wound-wait policy:
- higher-priority Xacts never wait for lower-priority Xacts.
- preemptive

Prevention Policy	T_i has higher p.	T_i has lower p.
Wait-Die	Ti waits for Tj	Ti aborts
Wound-Wait	Tj aborts	Ti waits for Tj

To avoid starvation, a restarted Xact must use its original timestamp!

Lock Conversion: Increase concurrency by allowing lock conversions
$UG_i(A)$: T_i upgrades its S-lock on object A to X-lock
$DG_i(A)$: T_i downgrades its X-lock on object A to S-lock

Lock Upgrade: Upgrade request is blocked if another Xact is holding a shared lock on A. Upgrade request is allowed if T_i has not released any lock.
Lock Downgrade: Downgrade request is allowed if: 1. T_i has not modified A, and 2. T_i has not released any lock.

Performance of Locking
- Resolve Xact conflicts using blocking & aborting
- Blocking causes delays in other waiting Xacts
- Aborting and restarting Xact wastes work done by Xact
- To increase system throughput: reduce locking granularity, reduce time a lock is held, reduce hotspot (frequently accessed and modified object)

Phantom read problem: A Xact re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently committed Xact. Phantom problem can be prevented by **predicate locking** (e.g. (balance>1000)). In practice, phantom problem is prevented via **index locking.**

Isolation Level	Dirty Read	Unrepeated Read	Phantom Read
READ UNCOMMITED	Possible	Possible	Possible
READ COMMITED	Not possible	Possible	Possible
REPEATABLE READ	Not possible	Not possible	Possible
SERIALIZABLE	Not possible	Not possible	Not possible

SQL's ***SET TRANSACTION ISOLATION LEVEL*** *command*

Degree	Isolation Level	Write locks	Read Locks	Predicate Locking
0	Read Uncommited	long d.	none	none
1	Read Commited	long d.	short d.	none
2	Repeatable Read	long d.	long d.	none
3	Serializable	long d.	long d.	yes

Short duration lock: lock acquired for an operation could be released after the end of operation before Xact commits/aborts; **Long duration lock**: lock acquired for an operation is held until Xact commits/aborts

Locking Granularity: Lock database → relation → page → tuple.
Size of data items being locked: highest(coarsest) granularity = database, lowest (finest) granularity = tuple. If Xact T holds a lock mode M on a data granule D, then T implicitly also holds lock mode M on granules finer than D.

Before acquiring S-lock/X-lock on a data granule G, need to acquire I-locks on granules coarser than G in a top-down manner.

Lock compatability matrix

Lock Requested	Lock Held: -	I	S	X
I	√	√	×	×
S	√	×	√	×
X	√	×	×	×

intention shared (IS): intent to set S-locks at finer granularity; **intention exclusive (IX):** intent to set X-locks at finer granularity

Lock compatability matrix

Lock Requested	Lock Held: -	IS	IX	S	X
IS	√	√	√	√	×
IX	√	√	√	×	×
S	√	√	×	√	×
X	√	×	×	×	×

Multi-granular locking protocol:
- Locks are acquired in top-down order
- To obtain S or IS lock on a node, must already hold IS or IX lock on its parent node
- To obtain X or IX lock on a node, must already hold IX lock on its parent node
- Locks are released in bottom-up order

L9: Multiversion Concurrency Control
$W_i(o)$ creates a new version of object O. $R_i(O)$ reads an appropriate version of O. Read-only Xacts are not blocked by update Xacts. Update Xacts are not blocked by read-only Xacts. Read-only Xacts are never aborted. *Notation:* $W_i(x)$ *creates a new version of x denoted by* x_i. If there are multiple versions of an object x, a read action on x could return any version. An interleaved execution could correspond to different multiversion schedules depending on the MVCC protocol.

Two schedules, S and S', over the same set of transactions, are defined to be **multiversion view equivalent** ($S \equiv_{mv} S'$) if they have the same set of **read-from relationships.** $R_i(x_j)$ occurs in S iff $R_i(x_j)$ occurs in S'.

A multiversion schedule S is called a **monoversion schedule** if each read action in S returns the most recently created object version. A monoversion schedule is defined to be a **serial monoversion schedule** if it is also a serial schedule. E.g (mono): $R_1(x_0), W_1(x_1), R_2(x_1), W_2(y_2), R_1(y_2), W_1(z_1)$

A multiversion schedule S is defined to be **multiversion view serializable schedule (MVSS)** if there exists a serial monoversion schedule (over the same set of Xacts) that is multiversion view equivalent to S.
Theorem 1: A view serializable schedule (VSS) is also a multiversion view serializable schedule (MVSS). *but a MVSS does not imply VSS.*

Snapshot Isolation (SI): Each Xact T sees a snapshot of DB that consists of updates by Xacts that committed before T starts. $W_i(O)$ creates a version of O denoted O_i. O_i is more recent version compared to O_j if $commit(T_i) > commit(T_j)$. $R_i(O)$ reads either its own update (if $W_i(O)$ precedes $R_i(O)$) or the latest version of O that is created by a Xact that commited before T_i started. **Concurrent Update Property:** If multiple concurrent Xacts updated the same object, only one of Xacts is allowed to commit. If not, the schedule may not be serializable.

First Committer Wins (FCW) Rule: Before commiting T, the system checks if there exists a committed concurrent Xact T' that has updated some object that T has also updated. If T' exists, then T aborts. Otherwise, T commits.
First Updater Wins (FUW) Rule: Whenever a Xact T needs to update an object O, T requests for a X-lock on O.
If the X-lock is not held by any concurrent Xact, then
- T is granted the X-lock on O
- If O has been updated by any concurrent Xact, then T aborts
- Otherwise, T proceeds with its execution

Otherwise, if the X-lock is being held by some concurrent Xact T', then T wait untils T' aborts or commits.
- If T' aborts, then
  - Assume that T is granted the X-lock on O
  - If O has been updated by any concurrent Xact, T aborts
  - Otherwise, T proceeds with its execution
- If T commits, then T is aborted

Garbage Collection: A version O_i of object O may be deleted if there exists a newer version O_j (i.e., $commit(T_i) < commit(T_j)$)
such that for every active Xact T_k that started after the commit of T_i (i.e., $commit(T_i) < start(T_k)$), we have $commit(T_j) < start(T_k)$

W1(x1), C1, W2(x2), C2, **W4(x4),** R3(y0), C4, W5(x5), C5, R6(z0)

Active Transactions: T_3 & T_6. Versions that can be deleted: x1, x4

Snapshot Isolation Tradeoffs: Performance of SI often similar to Read Committed. Unlike Read Committed, SI does not suffer from lost update or unrepeatable read anomalies. SI is vulnerable to some non-serializable executions. Snapshot isolation does not guarantee serializability.

Write Skew Anamoly

T1	T2
R1(x0)	
	R2(x0)
R1(y0)	
	R2(y0)
W1(x1)	
Commit1	
	W2(y2)
	Commit2

Read-Only Transaction Anomaly

T1	T2	T3
R1(y0)		
	R2(x0)	
W1(y1)		
Commit1		
	R2(y0)	
	W2(x2)	
		R3(x0)
		R3(y1)
		Commit3
	Commit2	

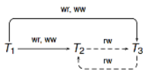
The above SI schedules are not MVSS. Draw DSG

Serializable Snapshot Isolation (SSI) Protocol
Stronger protocol that guarantees serializable SI schedules.
- Keep track of rw dependencies among concurrent Xacts
- Detect formation of T_j involving two rw dependencies ($T_i \to T_j \to T_k$)
- Once detected, abort one of T_i, T_j or T_k.
- May result in unnecessary rollbacks due to false positives of SI anomalies.
- **ww dependency** from T1 to T2: T1 write a version of x, and T2 later writes the immediate successor version of x.
- **wr dependency** from T1 to T2: T1 writes a version of x, and T2 reads this version of x
- **rw dependency** from T1 to T2: T1 reads a version of x, T2 later creates the immediate successor version of x

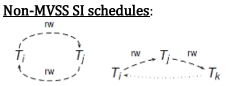
x_j is the **immediate successor** of x_i if (1) T_i commits before T_j, and (2) no transaction that commits between T_i's and T_j's commits produces a version of x.

Dependency Serialization Graph (DSG): Consider a schedule S consisting of a set of committed transactions {T1, · · · ,Tk}. DSG(S) is an edge-labelled directed graph (V,E). V represents tx and E represent xact dependencies. Edge types: - -> concurrent, → non-concurrent.

W1(x), W1(y), C1,
R2(x), W2(y), C2,
W3(x), R3(y), C3



Non-MVSS SI schedules:



If S is a SI schedule that is not MVSS, then 1. There is atleast one cycle in DSG(S), and 2. For each cycle in DSG(S), there exists three tx T_i, T_j, T_k
- T_i & T_k are possibly the same tx
- T_i & T_j are concurrent with an edge $T_i \to^{rw} T_j$, and
- T_j & T_k are concurrent with an edge $T_j \to^{rw} T_k$

L10: Crash Recovery: **Recovery Manager** guarantees atomicity and durability. **Undo**: Remove effects of aborted Xacts to preserve atomicity. **Redo**: Re-installing effects of commited Xact for durability. **Commit(T)** – install T's updated pages into database. **Abort(T)** – restore all data that T updated to their prior values. **Restart** – recover db to consistent state from system failure: 1. Abort all active Xacts at the time of system failure 2. Installs updates of all commited Xacts that were not installed in the DB before failure

Steal Policy: Allows dirty pages updated by T to be replaced from buffer pool before T commits (allow dirty pages to be written to disk before T commits)
Force Policy: requires all dirty pages updated by T to be written to disk when T commits

	Force	No-force
Steal	undo & no redo	undo & redo
No-steal	no undo & no redo	no undo & redo

Log-based Database Recovery
Log (aka trail/journal): history of actions executed by DBMS
- Contains a log record for each write, commit, & abort
- Stored in stable storage
- Each log record has a unique identifier called Log Sequence Number (LSN) (earlier log record has smaller LSN).

ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) Recovery Algorithm
- Designed to work with a steal, no-force approach. Assumes strict 2PL.
- Log file
- Transaction table (TT)
  - One entry for each **active Xact**
    - **XactID**: Transaction identifier, **Xact status** (C or U), **lastLSN**: LSN of the most recent log record for this Xact
- Dirty page table (DPT)
  - One entry for each **dirty page** in buffer pool
    - **pageID**: pageID of dirty page, **recLSN**: LSN of the earliest log record for an update that caused the page to be dirty
- Information in log records
  - **LSN, type, XactID, pageID, prevLSN, undoNextLSN**
  - prevLSN: LSN of the previous log record for the same Xact
- Update log record
  - XactID of page updated, before- & after-image of update

Implementing Abort: Undo all updates by Xact to DB pages
- **Write-ahead logging (WAL)** protocol: Do not flush an uncommited update to the DB until log record containing its before-image has been flushed to the log
- To enforce WAL: each db page contains LSN of the most recent log record (**pageLSN**) that describes an update to this page
- Before flushing a **database page** P to disk, ensure that all the log records up to the log record corresponding to **P's pageLSN** have been flushed to disk
- To undo all updates of Xact: **TT** maintains one entry for each active Xact, Each TT entry stores the LSN of most recent log record for Xact (**lastLSN**); use **lastLSN** to retrieve most recent log record, the other records are retrieved via **prevLSN**
- **Logging Changes During Undo**: Changes made to DB while undoing a Xact are also logged to ensure that an action is not repeated in the event of repeated undos

Implementing Commit: Ensure all updates of Xact must be in stable storage (database or log) before Xact is committed
- **Force-at-commit protocol:** Do not commit a Xact until the after-images of all its updated records are in stable storage
- To enforce force-at-commit protocol: Write a commit log record for Xact & Flush all the log records for Xact to disk
- Xact is considered to have committed if its commit log record has been written to stable storage

Implementing Restart: Recovery from system crashes consists of 3 phases: **Analysis phase**: identifies dirtied buffer pool pages & active Xacts at time of crash. **Redo phase**: redo actions to restore database state to what it was at the time of crash. **Undo phase**: undo actions of Xacts that did not commit
Repeating History During Redo: During restart following a crash, first restore system to the state before crash, and then undo the actions of Xacts that are active at the time of crash

Updating Xact Table (transID, lastLSN, status)
- When the first log record is created for Xact T, create a new entry for T with status = U
- When a new log record r is created for Xact T, update lastLSN for T's entry to be r's LSN
- If Xact T commits, update status for T's entry to be C
- When an end log record is generated for Xact T, remove T's entry

Updating Dirty Page Table (pageID, recLSN)
- When a page P in bufpool is updated & DPT has no entry for P, create new entry for P: **recLSN** = LSN of log record corresponding to update
- When a dirtied page P in bufpool is flushed to disk, remove entry for P

Types of Log Records
Update log record: After updating a page P, create an update log record r
- Update pageLSN of P = LSN of r
  - **pageID**: identifier of page being updated
  - **offset**: byte offset within page indicating start of updated portion
  - **length**: number of bytes for updated portion of data page
  - **before-image**: value of the changed bytes before update
  - **after-image**: value of the changed bytes after update

Compensation log record (CLR): When the update described by an update log record (ULR) is undone, create a compensation log record
  - **pageID**
  - **undoNextLSN** = LSN of next log record to be undone (i.e., prevLSN in ULR)
  - action taken to undo update

Commit log record: When Xact is to be committed, create commit log record r
- All log records (up to and incl. r) are force-written to stable storage
- Xact is considered committed once r has been written to stable storage

Abort log record: When a Xact is to be aborted, create an abort log record
- Undo is initiated for this Xact

End log record: Once the additional follow-up processing initiated by a aborted/committed Xact has completed, create an end log record
** Update log records & CLRs are classified as* ***redoable log records***

Analysis Phase
1. Determines the point in the log to start the Redo phase
2. Determines the superset of buffer pages that were dirty at time of crash
3. Identifies active Xacts at the time of crash
At the end of Analysis phase:
- Xact table = list of all active Xacts (with status = U) at time of crash
- dirty page table = superset of dirty pages at time of crash

Redo Phase
- **RedoLSN** = smallest recLSN among all dirty pages in DPT
- Let r be the log record with LSN = RedoLSN
- Scan the log in forward direction starting from r

If (r is an update log record) or (r is a CLR) then
- fetch page P that is associated with r
- If (P's pageLSN < r's LSN) then
  - Reapply logged action in r to P
  - update P's pageLSN = r's LSN

At the end of Redo Phase,
- Create end log records for Xacts with status=C in Xact Table & remove their entries from TT → System is restored to state at time of crash

Undo Phase: abort active Xacts at time of crash (loser Xacts)
- Abort loser Xacts by undoing their actions in reverse order
- Initialize L = set of lastLSNs (with status = U) from TT

Repeat until L becomes empty:
- delete the largest lastLSN from L
- let r be the log record corresponding to this lastLSN
- if r is an update log record for Xact T on page P then
  - create a CLR r_2 for T: r_2's undoNextLSN = r's prevLSN
  - update T's entry in TT: lastLSN = r_2's LSN
  - undo the logged action on page P
  - update P's pageLSN = r_2's LSN
  - `Update-L-and-TT`(r's prevLSN)
- else if r is a CLR for Xact T then
  - `Update-L-and-TT`(r's undoNextLSN)
- else if r is an abort log record for Xact T then `Update-L-and-TT`(r's prevLSN)

`Update-L-and-TT(lsn)`
- if lsn is not null then add lsn to L
- else create an end log record for T & remove T's entry from TT

Checkpointing: Perform checkpoint operations periodically to speed up restart recovery**.** Checkpointing synchronizes state of log with database state
Simple Checkpointing
1. Stop accepting any new update, commit, & abort operations
2. Wait till all active update, commit, & abort operations have finished
3. Flush all dirty pages in buffer
4. Write a checkpoint log record containing Xact table
5. Resume accepting new update, commit, & abort operations

During restart recovery, Analysis Phase begins with the latest **checkpoint log record** (**CPLR**)
- Init **Xact table** with CPLR's Xact table & **dirty page table** to be empty

Fuzzy Checkpointing in ARIES
1. Let DPT' be the dirty page table & TT' be the Xact table
2. Write a begin_checkpoint log record
3. Write a **end_checkpoint log record** containing **DPT' & TT'**
4. Write a special **master record** containing the LSN of the begin_checkpoint log record to a known place on stable storage

During restart recovery, Analysis Phase starts with the **begin_checkpoint log record (BCPLR**) identified by the master record
- Let ECPLR denote the **end_checkpoint log record (ECPLR)** corresponding to BCPLR
- Assume that there are no log records between BCPLR & ECPLR
- Init **Xact table** with ECPLR's Xact table & **dirty page table** with ECPLR's dirty page table

Redo Phase: Optimization: Exploit information in DPT to avoid retrieving P
Optimization condition: (P is not in DPT) or (P's recLSN in DPT > r's LSN)
If optimization condition holds, then the update of r has already been applied to P → r can be ignored & no need to fetch P!
- If (r is redoable) and (optimization condition does not hold) then
  - fetch page P that is associated with r
  - If (P's pageLSN < r's LSN) then
    - Reapply logged action in r to P & update P's pageLSN = r's LSN
  - Else
    - // recLSN ≤ r's LSN ≤ P's pageLSN;
    - Update P's entry in DPT: recLSN = P's pageLSN + 1