|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **External Merge Sort**. File size of N pages. Use B num of buffer pages.  Pass 0: Create sorted runs: - read in and sort B pages at a time. - Num of sorted runs = . Size of each sorted run = B pages (except for last run)  Pass i, i ≥ 1: Merging of sorted runs: - Use B-1 buffer pages for input & 1 buffer page for output. - Performs (B-1) way merge  Analysis: N0 = num of sorted runs created in pass 0 = . Num of passes = (+1 for pass 0)  Total num of I/O = (each pass reads N pages & writes N pages) | | | | | | | | |
| **Optimization w Blocked I/O** : Given B buffer pages: Read and write in units of buffer blocks of b pages. If b = 1, then same as default  - Allocate 1 block (b pages) for output. - Remainder blocks for input. F = num of runs that can be merged per pass =  Analysis: N0 = num of initial sorted runs = . Num of passes = | | | | | | | | |
| **Sorting using B+-trees** : When table to be sorted has a B+-tree index on sorting attributes  1) Format 1: Sequentially scan leaf pages of B+-tree. 2) Format 2 or 3: Sequentially scan leaf pages of B+-tree + retrieve data records using RIDs | | | | | | | | |
| **Access Path**: 1) Table scan = scan all data pages. 2) Index scan = scan index pages. 3) Index intersection = combine results from multiple index scans | | | | | | | | |
| **B+-tree: Include Columns** : E.g. B+-tree index on Student name: *CREATE INDEX stu\_name\_index ON Student (name) INCLUDE (major, year)*  An index I is a **covering index** for a query Q if all attrs referenced in Q are part of the key or include column(s) of I: - Q can be evaluated using I w/o any RID lookup - Such an evaluation plan is known as index-only plan | | | | | | | | |
| Conjunctive Normal Form (CNF) Predicates: - **term** = *R.A op c* or *R.Ai op R.Aj* . - **conjunct** = 1 or more terms connected by  A conjunct that contains is disjunctive (or contains a disjunction). - **CNF predicate** = 1 or more conjuncts connected by | | | | | | | | |
| **B+-tree** **index** I = (K1, K2, …, Kn). Non-disjunctive CNF predicate p  I **matches** p if p is of the form: (0 or more equality predicate)  where (K1, …, Ki) is a prefix of the key of I and there is at most 1 non-equality comparison operator which must be on the last attr of the prefix (i.e. Ki) | | | | | | | | |
| **Hash index** I = (K1, K2, …, Kn). Non-disjunctive CNF predicate p. I **matches** p if p is of the form: (K1 = c1) (K2 = c2) … (Kn = cn) | | | | | | | | |
| **Primary Conjuncts** : The subset of conjuncts in selection predicate p that an index I matches. Primary conjuncts: (age ≥ 18) (age ≤ 20)  E.g. B+-tree index I = (age, weight, height). Predicate p = (age ≥ 18) (age ≤ 20) (weight = 65) (level = 3). (BETWEEN counted as 1 op?). | | | | | | | | |
| **Covered Conjuncts** : all the attrs in conjunct appear in the key or include column(s) of I. Primary conjuncts Covered conjuncts | | | | | | | | |
| r |  |  | bd | bi | F | | h | B |
| RA expr | Num of tuples in output of r | Num of pages in output of r | Num of data records that can fit on a page | Num of data entries that can fit on a page | avg fanout of B+-tree index (num of pointers to child nodes) | | height of B+-tree index  h = if format-2 index on table R | Num of available buffer pages |
| **Cost of B+-tree index evaluation of p**. Let p' = primary conjuncts of p, pc = covered conjuncts of p  1. Navigate nodes to get 1st leaf page =  2. Scan leaf pages to get RID = 3. RID lookups =  Cost of RID lookups could be reduced by first sorting RIDs: | | | | | | | | |
| **Cost of hash index evaluation of p**  Let p' = primary conjuncts of p  For format-1 index: cost to retrieve data records = at least | | | | | | For format-2 index: - cost to retrieve data entries: at least  - cost to retrieve data records = | | |

|  |  |  |
| --- | --- | --- |
| **Projection** | = project cols given by list L from relation R. = same as but preserves duplicates | |
| **Sort-based Approach**. Cost analysis: 1) Cost to scan records and extract attributes = |R|. Cost to output temporary result = ||  2) Cost to sort records = 2||(logm(N0) + 1), where N0 = num of initial sorted runs, m = merge factor. 3) Cost to scan records = ||  **Optimized Sort-based Approach**: R -> 1) + 2a) Create sorted runs w attrs L -> 2b) + 3) Merge sorted runs & remove duplicates ->  If B > : Num of initial sorted runs N0 = . Num of merging passes = logB-1(N0) ≈ 1  Cost = pass 0 + pass 1 = [ |R| + ] + which is the same I/O for hash based | | |
| **Hash based approach**. Consider . Build a main-memory hash table T to detect & remove duplicates. Cost = |R| if T fits in main memory  If T cannot fit in main memory and no partition overflow, Total cost = |R| + 2 (B > size of partition + 1)  1) Partitioning phase (Cost = |R| + ): partitions R into R1, R2, …, RB-1 (no overlap btw partitions). 1 buffer for input, (B-1) buffer for output  - For each tuple t in input buffer: = t'. hash t' into 1 output buffer. Flush output buffer to disk whenever buffer is full  2) Eliminates duplicates from each (Cost = ): For each Ri: - initialize in-memory hash table. - for each tuple t Ri, - Hash t into bucket Bj w hash function h' (h' ≠ h). -- Insert t into Bj if t Bj. -- Output tuples in hash table  **Partition overflow problem**: Hash table for is larger than available memory buffers. Soln: Recursively apply partitioning. f ≈ 1.2  Each Ri has pages. Size of hash table for each Ri = , where f = fudge factor. To avoid partition overflow, B > ≈ B > | | |
| **Using Indexes**: If index is ordered (e.g. B+-tree) whose search key includes wanted attrs as a prefix: - scan data entries in order. - remove duplicates | | |
| **Joins** | Given a join R S. R = outer relation (smaller) and S = inner relation (larger) | |
| **Tuple-based Nested Loop Join**. I/O Cost Analysis: |R| + \* |S| | | **Page-based Nested Loop Join**. I/O Cost: |R| + |R| \* |S| |
| **Block Nested Loop Join.** 1 page S, 1 page output, B-2 pages for R. I/O Cost: |R| + . For each B-2 page of R (for each page S, check) | | |
| **Index Nested Loop Join**. Let R.Ai = S.Bj be join condition. Assume Uniform dist: each R-tuple joins w num of S-tuples  - For format-1 B+-tree index on S, I/O Cost = |R| + \* J, where J = = search index's internal nodes + search index's leaf nodes | | |
| **Sort-Merge Join.** Cost to sort R = 2|R|(), where NR = num of initial sorted runs of R, m = merge factor. Cost sort S = 2|R|()  If each S partition is scanned at most once during merging, merging cost = |R| + |S|  Worst case when each tuple of R requires scanning entire S, merging cost = |R| + \* |S| | | |
| **Optimized Sort-Merge Join**: Combine merge phase of sorting & merge phase of join. - Merge sorted runs until B > N(R, i) + N(S, j) ≈ for some i & j, then do merge and join at same time, where N(R,i) = total num of sorted runs of R at end of pass i of sorting R  If n pass until can merge and join at same time. - I/O Cost = [ 2 \* n \* (|R| + |S|) ] + (|R| + |S|) = pass 1 + … + pass n + 1 = (2n + 1)(|R| + |S|) | | |
| **Hash Join**. For RS. Partition R and S into k partitions using some hash fn h: - R = R1 … Rk, t Ri iff h(t.A) = i. - S = S1 … Sk, t Si iff h(t.B) = i.  Joins corresponding pair of partitions R S = (R1 S1) (R2 S2) … (Rk Sk). | | |
| **Grace Hash Join**. R = build (smaller) relation. S = probe (outer) relation. Set k = B-1 given B buffer. 1) Partition R into R1, …, Rk. 2) Partition S into S1, …, Sk.  3) Probing phase: probes each Ri with Si: - Read Ri to build hash table. - Read Si to probe hash table  Assume uniform hash dist: - size of each partition Ri is |R|/(B-1). - size of hash table for Ri is f \* |R| / (B-1), where f = fudge factor  - During probing phase, B > f \* |R| / (B-1) + 2 (1 for Si + 1 output). - Approximately, B > . Can assume B > [size of each partition of min(R,S) + 2].  I/O Cost = cost of partitioning phases + cost of probing phase = [ 2 \* (|R| + |S|) ] + [ |R| + |S| ] = 3 \* (|R| + |S|) (if no partition overflow) | | |
| **Multiple equality-join**: (R.A = S.A) (R.B = S.B). Algos: 1) INLJ: use index on all or some of join attrs. 2) SMJ: sort on combi of attrs. 3) Other algos same  **Inequality-join conditions**: (R.A < S.A). Algos: 1) INLJ: requires B+-tree index. 2) SMJ & Hash-based Joins: N/A. 3) Other algos same | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Set Operations**. Sorting approach for R S: Sort R using all attrs. Sort S using all attrs. Merge sorted operands to combine and discard duplicates  Hashing for R S: Same as GHJ. Algos for R - S are similar to those for R S | | | | | |
| **Aggregation**. Can use sorting or hashing approach. Avoid table scan: If there's a covering index for query  **Group-by aggregation**: partition by grp attr and run aggregation or use index | | | | | |
| 1) **Materialized evaluation**: - Operator is evaluated only when each of its operands has been completely evaluated or materialized  - Intermediate results are materialized to disk. - Bottom up approach. Can combine pipelined evaluation w **partial materialization**  2) **Pipelined evaluation**: - Output produced by a operator passed directly to its parent operator. - Execution of operators is interleaved. - Top down mtd.  - Operator O = **blocking operator** if O cannot start until it has received all the input tuples from its child operators (e.g. external merge sort, SMJ, GHJ)  - 3 fns: 1) **open**: initialize state of iterator; resource, args (selection conditions). 2) **getNext**: generate next output tuple. Return null when done. 3) **close** | | | | | |
| **Query plans** | Join plan notation: outer relation is left child, inner relation is right child. For hash join: probe relation is left child, build relation is right | | | | |
| Commutativity of binary ops | | 1.1) | 1.2) | | |
| Associativity of binary ops | | 2.1) | 2.2) | | |
| Idempotence of unary op | | 3.1) | | 3.2) | |
| Commutating selection w projection | | 4.1) | | | |
| Commutating selection w binary ops | | 5.1)  (if attr(p) attr(R)) | 5.2)  (if attrs(p) attrs(R)) | | 5.3) |
| Commutating projection w binary ops | | 6.1) | 6.2) .  (if attr(p)attr(R) LR and attr(p)attr(S) LS) | | 6.3) |
| **Query Optimization** | Types of query plan trees: 1) Linear: if at least 1 operand for each join operation is a base relation; otherwise plan is bushy  2) Left-deep: if every right join operand is a base relation. 3) Right-deep: if every left join operand is a base relation | | | | |
| **Query plan enumeration**. Dynamic programming formulation. Input: A SPJ query q on relations R1, R2, …, Rn. Output: An optimal query plan for q. O(3n)  Get best access plan for all single relations. Calculate optimal plan involving 2, 3 and so on relations by finding best way to join each smaller relation. | | | | | |
| **System R Optimizer**. - enumerates only left deep query plans. - avoids cross-product query plans. - considers early & . - Uses enhanced dp approach  - use optPlan(Si, oi) where oi captures the sort order of output produced by query plan w.r.t Si. - oi = NULL if output is unordered or a seq of attrs. | | | | | |
| **Cost Estimation**. Assumptions: - Uniform dist. - Diff attrs are indep. - Inclusion: For R S, if , then | | | | | |
| **Size estimation**. Consider query q = , where p = t1 t2 … tn, and e = R1 R2 … Rm, . Each term ti is a filter  Reduction/selectivity factor of a term ti = rf(ti) = . By independence assumption,  Consider relation R(A,…) w = 45 and = 15. Using uniformity assumption, rf(A = c) ≈ 1/. So estimated to be 3 | | | | | |
| **Join selectivity** factor = selectivity factor for join predicates. rf(R.A = S.B) = . Assume .  By inclusion assumption, every R-tuple joins w some S-tuple. By uniformity assumption, S-tuples for each S.B value.  Thus, each R-tuple joins w S-tuples. So . rf(R.A = S.B) ≈ 1/max{, } | | | | | |
| Estimation using **Histograms**. Main idea: - partition attribute's domain into buckets. Assume value dist within each bucket is uniform  1) Equiwidth: ea bucket has almost same num of values. 2) Equidepth: ea bucket has almost same num of tuples. Sub-ranges of adj buckets can overlap  Improved Histogram Estimation w MCV (most common values). Track freq of top-k MCV and exclude MCV from histogram's buckets | | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Transaction** (Xact; BEGIN TRANSACTION, ..., COMMIT). Each Xact must end w either commit or abort action. An active Xact = Xact that is still in progress | | | | |
| **Schedule** = list of actions from a set of Xacts, where order of actions within each Xact is preserved. **Serial schedule** = actions of Xacts are not interleaved  - Tj reads O from Ti if last write action on O before Rj(O) in S is Wi(O). - Tj reads from Ti if Tj read some obj from Ti. - Ti performs final write on O  An interleaved Xact execution schedule is **correct** if it is "equivalent" to some serial schedule over the same set of Xacts  2 schedules S and S' (over the same set of Xacts) are **view equivalent** (S S') if: 1) If Ti reads A from Tj in S, then Ti must also read A from Tj in S'  2) For each data obj A, the Xact (if any) that performs the final write on A in S must also perform the final write on A in S'  **View Serializable Schedule** **(VSS)** = a schedule S that is some serial schedule over the same set of Xacts | | | | |
| **VSG(S)** to capture read-from and final-write relations among transactions in S. Node = Xact, Edges = precedence relations among Xacts.  Edge (Tj, Ti): - If Ti reads from Tj. - If both Ti & Tj updates the same obj O & Ti performs final write on O. - If Tj reads obj O from initial DB & Ti update obj O  If VSG cyclic, S not VSS. If VSG acyclic, then S is VSS iff a serializable schedule produced from a topological ordering of VSG that is S | | | | |
| **Conflicting Actions** | 2 actions on the same object conflict if 1) at least 1 of them is a write action, and 2) the actions are from diff Xacts | | | |
| **Anomalies w Interleaved Xact Executions**  1) Dirty read (due to WR conflict): - T2 reads an obj that has been modified by T1 and T1 has not yet committed. - T2 could see an inconsistent DB state  2) Unrepeatable read (RW): - T2 updates an obj that T1 has read and T2 commits while T1 still in progress. - T1 could get a diff value if it reads obj again  3) Lost update (WW): - T2 overwrites value of an obj that has been modified by T1 while T1 still in progress. - T1's update is lost  4) Phantom read (concurrent Xact): A Xact re-executes a query on a predicate and get a diff result due to a recently committed Xact. Similar to 2)  Phantom problem can be prevented by predicate locking. In practice, phantom problem is prevented via index locking | | | | |
| **Conflict Serializable Schedules (CSS)** if it is to a serial schedule over the same set of Xacts  **Conflict equivalent** (S S') if they order every pair of conflicting actions of 2 commited Xacts in the same way  **Conflict serializability graph CSG(S)**: Node = commited Xact, E contains (Ti, Tj) if an action in Ti precedes and conflict w one of Tj's actions  **Thrm 1**: A schedule is CSS iff its CSG is acyclic. **Thrm 2**: *.* For convenience, use serializable to mean conflict serializable | | | | |
| **Blind write** = A write on obj O by Ti where Ti did not read O prior to the write. | | | **Thrm 3:** If S is VSS and S has no blind writes (WW), then S is also CSS | |
| **Cascading Aborts** | | For correctness, if Ti has read from Tj, then Ti must abort if Tj aborts. Recursive aborting process = cascading abort | | |
| **Recoverable Schedules** = for every Xact T that commits in schedule S, T must commit after T' if T reads from T'  - Guarantee committed Xacts won't be aborted, but cascading aborts of active Xacts still possible (incur performance penalty) | | | | |
| **Cascadeless schedule** = if Ti reads from Tj in schedule S, Commitj must precede this read action. | | | | **Thrm 4 + 5**: Strict cascadeless recoverable |
| **Before-Images**. To undo actions of aborted Xacts is to restore before-images for writes. E.g. W1(A), W2(A), Abort2  However, before-image recovery doesn't always work. E.g. W1(A), W2(A), Abort1. Will undo effect of W2 as well | | | | |
| **Strict schedule** = for every Wi(O) in schedule S, O is not read or written by another Xact until Ti either aborts or commits  Enables use of before-images for recovery. Tradeoff: - recovery more efficient. - concurrent executions more restrictive | | | | |
| Serial schedule must be strict & conflict serializable. Cascadeless if all read operations are non-dirty. Strict if all read and write operations are non-dirty | | | | |

|  |
| --- |
| **Concurrency Control**. For each input action (R, W, C, A) to Xact scheduler, either output action to schedule, postpone (block Xact) or reject (abort Xact) |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Lock Requested | Lock Held | | | | | | - | IS | IX | S | X | | IS | ✓ | ✓ | ✓ | ✓ |  | | IX | ✓ | ✓ | ✓ |  |  | | S | ✓ | ✓ |  | ✓ |  | | X | ✓ |  |  |  |  |   1) **Lock-Based Concurrency Control**. Locking modes: - Shared (S) locks for reads. - Exclusive (X) locks for read/writes.  Let Si(O) = Xact Ti is requesting S-lock on obj O. Xi(O) similarly defined. Ui(O) = Xact Ti releases lock on obj O  1) A lock request is granted on O if requesting lock mode is compatible w lock modes of existing locks on O  2) If T's lock request is not granted on O, T is **blocked**: execution is suspended & T added to O's request queue  3) When a lock is released on O, lock manager checks the request of the 1st Xact T in the request queue for O. If can be granted, T acquires its lock on O and resumes execution after its removal from the queue  4) 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**. 1) To read O, T need S or X on O. 2) To write O, T need X on O3) Once T releases a lock, T can't request any more locks  2PL can be split into 2 phases: 1) Growing phase: before releasing 1st lock. 2) Shrinking phase: after releasing 1st lock. **Thrm 1**: 2PL schedules are CSS |
| **Strict 2PL Protocol**: 1) and 2) are the same. 3) A Xact must hold on to locks until Xact commits or aborts. **Thrm 2**: Strict 2PL schedules are strict & CSS |
| **Deadlocks** = Cycle of Xacts waiting for locks to be released by each other. To deal w deadlocks: either detect deadlock or prevent deadlock  **Deadlock Detection: Waits-for graph (WFG):** Nodes = active Xacts. Edge Ti Tj if Ti waiting for Tj to release lock. Remove edge when lock request granted  Deadlock is detected if WFG has a cycle. Breaks a deadlock by aborting a Xact in cycle. Alternative to WFG: timeout mechanism   |  |  |  | | --- | --- | --- | | Prevention Policy | Ti has higher priority | Ti has lower priority | | Wait-die | Ti waits for Tj | Ti aborts | | Wound-wait | Tj aborts | Ti waits for Tj |   **Deadlock Prevention**: Older Xacts (smaller timestamp) have higher priority than younger Xacts.  Suppose Ti requests for a lock that conflicts w a lock held by Tj  1) 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  2) Wound-wait: higher priority Xacts never wait for lower priority. - preemptive. To avoid starvation, a restarted Xact must use its original timestamp |
| **Lock conversion**. Increase concurrency by allowing lock conversions. Interleaved executions become possible w lock upgrading  UGi(A) = Ti upgrades its S on obj A to X. - Blocked if another Xact is holding a S lock on A. - Allowed if Ti has not released any lock  DGi(A): Ti downgrades its X on obj A to S. - Allowed if Ti has not modified A & Ti has not released any lock |
| Default = READ COMMITTED. BEGIN TRANSACTION; SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | ... | SERIALIZABLE }; ... COMMIT;   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | Degree | Isolation level | Dirty Read | Unrepeatable Read | Phantom Read | Write Locks | Read Locks | Predicate Locking | | 0 | READ UNCOMMITTED | possible | possible | possible | long duration | none | none | | 1 | READ COMMITTED | not possible | possible | possible | long duration | short | none | | 2 | REPEATABLE READ | not possible | not possible | possible | long duration | long | none | | 3 | SERIALIZABLE | not possible | not possible | not possible | long duration | long | yes |   **Short duration**: can release lock acquired after end of operation before Xact commits/aborts. **Long duration**: lock acquired held until Xact commits/aborts |
| **Locking Granularity** = size of data items being locked. Highest (coarsest) granularity = DB -> relation -> page -> tuple = Lowest (finest) granularity  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  **Multigranularity Locking**: Use a new intention lock (I-lock) mode. Before acquiring S or X on a data granule G, need to acquire I-locks on granules coarser than G in a top-down manner. Locks are released in bottom-up order  - Intention Shared (IS): intent to set S-locks at finer granularity. - Intention exclustive(IX): intent to set X-locks at finer granularity |

|  |
| --- |
| **Multiversion Concurrency Control (MVCC)**. Maintain multiple versions of each obj. Wi(x) creates a new version of x denoted by xi. Initial version = x0  Pros: - Read-only Xacts not blocked by update Xacts. - Update Xacts not blocked by read-only Xacts. - Read-only Xacts never aborted  **Multiversion View Equivalent** (S S') if S and S' have the same set of read-from relationships. i.e. Ri(xj) occurs in S iff Ri(xj) occurs in S'  **Monoversion schedule** = a multiversion schedule S where each read action in S returns the most recently created obj version  **Serial monoversion schedule** = a monoversion schedule where it is also a serial schedule (i.e. no interleaving)  **Multiversion view serializable schedule (MVSS)** = a multiversion schedule S where a serial monoversion schedule that is to S  **Thrm 1**: CSS -> VSS -> MVSS. Not MVSS -> Not CSS  Test for MVSS: use **MVSG(S)** to capture read-from relations among transactions in S (similar to VSG). Cyclic = not MVSS. Acyclic = MVSS  Edge (Tj, Ti) MVSG(S) if: - Ti reads from Tj. - Tj reads obj O from Tk & Ti update obj O, then either (Ti, Tk) MVSG(S) or (Tj, Ti) MVSG(S) |
| **MVCC Protocol: Snapshot Isolation (SI)**. Each Xact T sees a snapshot of DB that consists of updates by Xacts that committed before T starts  Each Xact T has 2 timestamps: start(T), commit(T). 2 Xacts T and T' are **concurrent** if they overlap, i.e. [start(T), commit(T)] [start(T'), commit(T')] ≠  Oi is a **newer** version compared to Oj if commit(Ti) > commit(Tj)  Ri(O) reads either its own update (if Wi(O) precedes Ri(O)) or the latest version of O created by a Xact that committed before Ti started,  - **Concurrent Update Property**: If multiple concurrent Xacts updated same obj, only 1 of the Xacts can commit. If not, schedule may not be serializable.  **First Commiter Wins (FCW)**: Before commiting T, if a committed concurrent T' that updated some obj that T also updated -> T aborts. Else, T commits  **First Updater Wins (FUW)**: Whenever a Xact T needs to update an obj O, T requests for a X-lock on O.  If X-lock not held by any concurrent Xact, then - T is granted X-lock on O. - If O has been updated by any concurrent Xact, then T aborts. - Else T proceeds.  Else, if X-lock is held by some concurrent Xact T', then T waits until T' aborts or commits.  If T' aborts, then: - assume T is granted X-lock on O. - If O has been updated by any concurrent Xact, then T aborts. - Else T proceeds with its execution.  If T' commits, then T is aborted |
| **Garbage Collection**. A version Oi of obj O may be deleted if there exists a newer version Oj (i.e. commit (Ti) < commit(Tj)) s.t. for every active Xact Tk that started after the commit of Ti (i.e. commit(Ti) < start(Tk)), we have commit(Tj) < start(Tk) |
| SI has similar performance to READ COMMITTED. Unlike READ COMMITTED, SI don't suffer from lost update or unrepeatable read anomalies. But SI is vulnerable to some non-serializable executions: 1) **Write Skew Anomaly**. 2) **Read-Only Transaction Anomaly**. SI don't guarantee serializability |
| Write Skew Anomaly: R1(x0), R2(x0), R1(y0), R2(y0), W1(x1), C1, W2(y2), C2. (LHS). DSG = T1 - - rw - -> T2 - - rw - -> T1  Read-only Anomaly: R1(y0), R2(x0), W1(y1), Commit1, R2(y0), W2(x2), R3(x0), R3(y1), Commit3, Commit2. (RHS). DSG = T3 - - rw - -> T2 - - rw - -> T1 –wr–> T3  Both are SI schedule but not MVSS |
| **Serializable Snapshot Isolation (SSI)**. Guarantees serializable SI schedules. Track rw dependencies among concurrent Xacts  If there exists a Tj involved in 2 rw dependencies, abort 1 of Ti, Tj, or Tk.  May result in unnecessary rollbacks due to false positives of SI anomalies. **SSI schedule** = schedule S that is SI and MVSS |
| **Transactional Dependencies**. - ww dependency from T1 to T2: T1 writes a version of obj x. T2 later writes the IS version of x  - wr from T1 to T2: T1 writes a version of obj x. T2 reads this version of x. - rw from T1 to T2: T1 reads a version of obj x. T2 later creates the IS version of x  xj = immediate successor (IS) of xi if 1) Ti commits before Tj, and 2) no Xact that commits btw Ti's and Tj's commits produces a version of x |
| **Dependency Serialization Graph (DSG).** E = transactional dependencies: Ti Tj or Ti Tj or Ti Tj. Edge - -> concurrent Xacts. –> non-concurrent Xacts. |
| **Non-MVSS SI schedules. Thrm 2**: If S is a SI schedule that is not MVSS, then 1) There is at least 1 cycle in DSG(S), and  2) For each cycle in DSG(S), there exists 3 transactions, Ti, Tj and Tk s.t. - Ti & Tk are possibly the same transaction |

|  |
| --- |
| **Crash Recovery.** - Undo: remove effects of aborted Xact to preserve atomicity. - Redo: re-installing effects of committed Xact for durability  Failure types: 1) Xact failure: Xact aborts. 2) System crash: loss of volatile memory contents. 3) Media failures: data lost/corrupted on non-volatile storage |
| **Recovery Manager**. Commit(T): install T's updated pages into DB. Abort(T): restore all data that T updated to their prior values.  Restart: recover DB to a consistent state from system failure. Abort all active Xacts at time of system failure. Installs updates of all committed Xacts that were not installed in the DB before the failure. Desirable properties: Little overhead. Recover quickly from a failure |
| |  |  |  | | --- | --- | --- | |  | Force | No-force | | Steal | undo & no redo | undo & redo | | No-steal | no undo & no redo | no undo & redo |   **Steal policy**: allows dirty pages updated by T to be replaced (written to disk) from buffer pool before T commits.  **Force policy**: requires all dirty pages updated by T to be written to disk when T commits. |
| Log is stored as a sequential file of records in stable storage (multiple copies on non-volatile storage devices). Earlier log records = smaller LSN  1) Update log record (ULR): Update **pageLSN** of P = LSN of r. Extra fields: - offset = byte offset within page indicating beginning of updated portion. - length = num of bytes for updated portion of data page. - before-image/after-image = value of changed bytes before/after update.  2) Compensation log record (CLR): When update described by an ULR is undone, create a CLR. Stable storage = DB or log  3) Commit log record: All log records (up to and including r) are force-written to stable storage. 4) Abort log record: Undo is initiated for this Xact.  ULR & CLRs are classified as redoable log records. When an end log record is generated for Xact T, remove T's entry in TT |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **LSN** (log seq num) | **type** | **XactID** | **pageID** | **prevLSN** | **undoNextLSN** | | id for log record | type of record |  | what page is modified | LSN of prev log record for same Xact | LSN of next log record to be undone = prevLSN of ULR | |  | update | t3 | p1 | null | - | |  | clr | t4 | p5 | 120 | 90 | |  | commit | t2 | - | 130 | - | |  | abort | t2 | - | 140 | - | |  | end | t1 | - | 150 | - | |  | begincp |  |  |  | - | |  | endcp |  |  |  | - | |
| |  |  | | --- | --- | | Dirty Page Table (DPT) | | | **pageID** | **recLSN** | | page ID of dirty page | LSN of earliest log record |  |  |  |  | | --- | --- | --- | | Transaction Table (TT) | | | | **XactID** | **lastLSN** | **status** | |  | LSN of most recent log record | C or U | |
| ARIES (Algo for Recovery and Isolation Exploiting Semantics). Designed to work w a steal, no-force approach. Assumes strict 2PL for concurrency control |
| **Write-ahead logging (WAL) protocol**: don't flush an uncommitted update to DB until the log record w its before-image has been flushed to the log  - Each DB page contains the LSN of the most recent log record = **pageLSN**, that describes an update to this page  - Before flushing a DB page P to disk, ensure all log records up to the log record corresponding to P's **pageLSN** have been flushed to disk |
| **Force-at-commit protocol**: don't commit a Xact until the after-images of all its updated records are in stable storage. - Write a commit log record for Xact.  - Flush all log records for Xact to disk. Xact is considered to have committed if its commit log record has been written to stable storage |
| **Simple Checkpointing**: 1) Stop accepting any new update, commit & abort operations. 2) Wait till all active update, commit & abort ops 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 ops  During restart recovery, Analysis Phase begins w the latest checkpoint log record (CPLR): - initialize TT w CPLR's Xact table. - Initialize DPT 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 (BCPLR)  3) Write a end\_checkpoint log record (ECPLR) containing DPT' & TT'. 4) Write a master record containing the LSN of theBCPLR to stable storage  During restart recovery, Analysis Phase starts w BCPLR identified by the master record: - For simplicity, Assume no log records btw BCPLR & ECPLR  - Initialize TT w ECPLR's Xact table. - Initialize DPT w ECPLR's dirty page table |
| **Implementing Restart**. 1) Analysis phase: identifies dirtied buffer pool pages & active Xacts at time of crash  2) Redo phase: redo actions to restore DB state to what it was at time of crash. 3) Undo phase: undo actions of Xacts that didn't commit |
| **Analysis phase**. 1) Initializes DPT and TT to be empty. 2) Scan the log in forward direction to process each log record r (for Xact T):  2a) If r is an end log record: { Remove T from TT }.  2b) Else { Add an entry in TT for T if T not in TT. Update **lastLSN** of entry = r's LSN. Update status of entry to C if r is a commit log record }  2c) If (r is a redoable log record for page P) & (P is not in DPT): { Create an entry for P in DPT w pageID of entry = P's pageID and **recLSN** of entry = r's LSN }  At end of analysis phase: TT = list of all active Xacts (w status = U) at time of crash. DPT = superset of dirty pages at time of crash |
| **Redo phase**. 1) **RedoLSN** = smallest **recLSN** among all dirty pages in DPT. Let r be log record w LSN = **RedoLSN**. 2) Scan log in forward dirn starting from r.  2a) If (r is ULR or CLR): { Fetch page P associated w r. If (r's LSN > P's **pageLSN**) then { Reapply logged action in r to P. Updates P's **pageLSN** = r's LSN } }  At end of Redo Phase, create end log records for Xacts w status = C in TT & 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  1) Initialize L = set of **lastLSNs** (w status = U) from TT. 2) Repeat until L becomes empty: {  2a) Delete largest **lastLSN** from L. Let r = log record corresponding to this **lastLSN**.  2b) If r is an ULR for T on page P, then { - create a CLR r2 for T: r2's **undoNextLSN** = r's **prevLSN**. - update T's entry in TT: **lastLSN** = r2's LSN. - Create a DPT  entry for P (with recLSN = r2's LSN) if P not in DPT. - undo the logged action on page P. - update P's **pageLSN** = r2's LSN. - *Update-L-and-TT* (r's **prevLSN**) }  2c) Else if r is a CLR for Xact T, then { *Update-L-and-TT*(r's **undoNextLSN**) }  2d) Else if r is an abort log record for Xact T, then { *Update-L-and-TT*(r's **prevLSN**) } }  def *Update-L-and-TT*(lsn): { if lsn is not null: { add lsn to L }. Else { create an end log record for T & remove T's entry from TT } } |
| **ARIES Analysis Phase w Fuzzy Checkpoint**. 1) Retrieve theBCPLR identified by the master record. 2) Retrieve theECPLR corresponding to BCPLR  3) Initialize DPT & TT using ECPLR’s contents. 4) Scan the log in forward dirn (starting from ECPLR) to process each log record r (for Xact T):  4a) If r is an end log record then remove Tfrom TT  4b) Else { Add an entry in TT for Tif Tnot in TT. Update **lastLSN** of entry to r’s LSN. Update status of entry to C if ris a commit log record }  4c) If (ris a redoable log record for page P) & (P not in DPT), then { Create an entry for P in DPT with pageID = P’s pageID and **recLSN** of entry = r’s LSN } |
| **ARIES Redo Phase w Fuzzy Checkpoint**. 1) **RedoLSN** = smallest recLSN among all dirty pages in DPT. Let rbe the log record with LSN = **RedoLSN** .  2) Scan the log in forward dirn starting from r. 2a)If (ris a redoable log record) and (condition C is false) then { 2ai) fetch page Passociated with r  2aii) If (P’s **pageLSN** < r’s LSN) then { Reapply logged action in rto P. Update P’s **pageLSN** = r’s LSN }  2aiii) Else { Update P’s entry in DPT: **recLSN** = P’s **pageLSN** + 1 } } # **recLSN** ≤ r's LSN ≤ P's **pageLSN**.  3) At the end of Redo Phase, Create end log records for Xacts with status = C in TT & remove their entries from TT.  Condition C: (Pis not in DPT) or (P’s **recLSN** in DPT > r’s LSN). If Condition C = TRUE: update of r already applied to P. r can be ignored |