CS3223

AY22/23 SEM 2

github/jovyntls github/JothamWong

1. k* is an actual **data record** (with search key k)
2. k* is of the form **(k, RID)** - fixed length $(\mathtt{k},\ \bullet)$
3. k* is of the form **(k, RID-list)** - e.g. (k, {RID11, RID12})

Notation	Meaning
r	relational algebra expression
$\|\|r\|\|$	number of tuples in output of r
$\|r\|$	number of pages in output of r
b_d	number of data records that can fit on a page
b_i	number of data entries that can fit on a page
F	average fanout of B$^+$-tree index (i.e., number of pointers to child nodes)
h	height of B$^+$-tree index (i.e., number of levels of internal nodes)
	$h = \lceil \log_F(\lceil \frac{\|\|R\|\|}{b_i} \rceil) \rceil$ if format-2 index on table R
B	number of available buffer pages

04.1 Sorting and Selection

External Merge Sort

- **sorted run** $\rightarrow$ sorted data records written to a file on disk
- divide and conquer
  1. create temporary file R_i for each B pages of R sorted
  2. merge: use $B-1$ pages for input, 1 page for output
- total I/O = $2N(\lceil \log_{B-1}(N_0) \rceil + 1)$
  - $2N$ to create $\lceil N/B \rceil$ sorted runs of B pages each
  - merging sorted runs: $2N \times \lceil \log_{B-1} N_0 \rceil$

optimisation with blocked I/O

- sequential I/O - read/write in *buffer blocks* of b pages
- one block (b pages) for output, remaining blocks for input
- number of runs merged per pass, * $F = \lfloor \frac{B}{b} \rfloor - 1$
- number of passes = $\lceil \log_F(N_0) \rceil + 1$

Sorting with B$^+$-trees

- when *sort key is a prefix of the index key* of the B$^+$-tree
- sequentially scan leaf pages of B$^+$-tree
  - for Format-2/3, use RID to retrieve data records

04.2 SELECTION: $\sigma_p(R)$

- $\sigma_p(R)$: selects rows from relation R satisfying predicate p
- **access path**: a way of accessing data records/entries
  - **table scan** $\rightarrow$ scan all data pages
  - **index scan** $\rightarrow$ scan index pages
  - **index intersection** $\rightarrow$ combine results from index scans
- **selectivity** of an access path $\rightarrow$ number of index & data pages retrieved to access data records/entries
  - more selective = fewer pages retrieved
- index I is a **covering index** for query Q $\rightarrow$ if all attributes referenced in Q are part of the key of I
  - Q can be evaluated using I without any RID lookup (**index-only** plan)

Matching Predicates

- **term** $\rightarrow$ of form $R.A\ op\ c$ or $R.A_i\ op\ R.A_j$
- **conjunct** $\rightarrow$ one or more terms connected by $\vee$
- **CNF predicate** $\rightarrow$ comprises one or more conjuncts connected by $\wedge$

$$\overbrace{(\underbrace{\text{rating} \geq 8}_{\text{term/conjunct}} \vee \underbrace{\text{director = "Coen"}}_{\text{term/conjunct}})}^{\text{disjunctive conjunct}} \wedge \underbrace{(\text{year} > 2003)}_{\text{term/conjunct}} \wedge \underbrace{(\text{language} = \text{"English"})}_{\text{term/conjunct}}$$

B$^+$-tree matching predicates

- for index $I = (K_1, K_2, \ldots, K_n)$ and non-disjunctive CNF predicate p
  - *at most one* non-equality comparison operator which must be on the last attribute of the prefix (K_i)
- matching index: matching records are in contiguous pages
  - non-matching index: not contiguous $\Rightarrow$ less efficient

Hash index matching predicates

- for hash index $I = (K_1, K_2, \ldots, K_n)$ and non-disjunctive CNF predicate p, all attributes must be present and all ops must be equality.

Primary/Covered Conjuncts

- **primary conjuncts** $\rightarrow$ subset of conjuncts that I matches
  - e.g. p = (age $\geq$ 18) $\wedge$ (age $\leq$ 20) $\wedge$ (weight=65) for I = (age, weight, height)
- **covered conjuncts** $\rightarrow$ subset of conjuncts covered by I
  - each attribute in covered conjuncts appears in key of I
- primary conjuncts $\subseteq$ covered conjuncts

Cost of Evaluation

let p' = primary conjuncts of p, $\quad p_c$ = covered conjuncts of p

B$^+$-tree index evaluation of p

1. navigate internal nodes to find first leaf page

$$\text{cost}_{\text{internal}} = \begin{cases} \lceil \log_F(\lceil \frac{||R||}{b_d} \rceil) \rceil & \text{if I is a format-1 index} \\ \lceil \log_F(\lceil \frac{||R||}{b_i} \rceil) \rceil & \text{otherwise} \end{cases}$$

2. scan leaf pages to access all qualifying data entries

$$\text{cost}_{\text{leaf}} = \begin{cases} \lceil \frac{||\sigma_{p'}(R)||}{b_d} \rceil & \text{if I is a format-1 index} \\ \lceil \frac{||\sigma_{p'}(R)||}{b_i} \rceil & \text{otherwise} \end{cases}$$

3. retrieve qualified data records via RID lookups

$$\text{cost}_{\text{RID}} = \begin{cases} 0 & \text{if I is a covering format-1 index,} \\ ||\sigma_{p_c}(R)|| & \text{otherwise} \end{cases}$$

   - reduce cost with **clustered** data records (sort RIDs): $\lceil \frac{||\sigma_{p_c}(R)||}{b_d} \rceil \leq \text{cost}_{RID} \leq \min\{||\sigma_{p_c}(R)||, |R|\}$

hash index evaluation of p

- **format-1**: cost to retrieve data records $\geq \lceil \frac{||\sigma_{p'}(R)||}{b_d} \rceil$
- **format-2**: cost to retrieve data entries $\geq \lceil \frac{||\sigma_{p'}(R)||}{b_i} \rceil$

  cost to retrieve data records = $\begin{cases} 0 & \text{if I is a covering index,} \\ ||\sigma_{p'}(R)|| & \text{otherwise} \end{cases}$

05. PROJECTION $\pi_{A_1,\ldots,A_m}(R)$ AND JOIN

- $\pi_L(R)$ eliminates duplicates, $\pi^*_L(R)$ preserves duplicates

Sort-based approach

cost analysis

1. extract attributes: $|R|$ scan + $|\pi^*_L(R)|$ output temp result
2. sort records: $2|\pi^*_L(R)|(\log_m(N_0) + 1)$
3. remove duplicates: $|\pi^*_L(R)|$ to scan records

optimised sort-based approach

- if $B > \sqrt{|\pi^*_L(R)|}$, same I/O cost as hash-based approach
  - $N_0 = \lfloor \frac{|R|}{B} \rfloor \approx \sqrt{|\pi^*_L(R)|}$ initial sorted runs
  - $\log_{B-1}(N_0) \approx 1$ merge passes

Hash-based approach

1. **partitioning phase**: hash each tuple $t \in R$
   - $R = R_1 \cup R_2 \cup \cdots \cup R_{B-1}$
     - for each R_i & $R_j, i \neq j$, $\pi^*_L(R_i) \cap \pi^*_L(R_j) = \emptyset$
   - for each t: project attributes to form t', hash $h(t')$ to one output buffer, flush output buffer to disk when full
   - one buffer for input, $(B-1)$ buffers for output
2. **duplicate elimination** from each $\pi^*_L(R_i)$
   - for each R_i: initialise in-mem hash table, hash each $t \in R_i$ to bucket B_j with $h' \neq h$, insert if $t \notin B_j$
   - write tuples in hash table to results

- **I/O cost** (no partition overflow): $|R| + 2|\pi^*_L(R)|$
  - partitioning cost: $|R| + |\pi^*_L(R)|$
  - duplicate elimination cost: $|\pi^*_L(R)|$
- partition overflow: recursively apply partitioning
  - to avoid, $B >$ size of hash table for R_i = $\frac{|\pi^*_L(R)|}{B_1} \times f$
    - approximately $B > \sqrt{f \times |\pi^*_L(R)|}$

Projection using Indexes

- if index search key contains all wanted attributes *as a prefix*
  - **index scan** data entries in order & eliminate duplicates

05.2 JOIN $R \bowtie_\theta S$

R = outer relation (smaller relation); $\quad S$ = inner relation

! for **format-2** index, add cost of retrieving record

nested loop joins

- **tuple-based** nested loop join: $|R| + ||R|| \times |S|$
- **page-based** nested loop join: $|R| + |R| \times |S|$
- **block nested loop join**: $|R| + (\lceil \frac{|R|}{B-2} \rceil \times |S|), \quad |R| \leq |S|$
  - 1 page output, 1 page input, $(B-2)$ pages to read R
  - for each $(B-2)$ pages of R: for each P_S of S: check r,s
- **index nested loop join**:

$$|R| + ||R|| \times \left(\log_F(\lceil \frac{||S||}{b_d} \rceil) + \lceil \frac{||S||}{b_d ||\pi_{B_j}(S)||} \rceil \right)$$

  - joining $R(A, B) \bowtie_A S(A, C)$ with B+tree index on $S.A$
  - for each tuple $r \in R$, use r to probe S's index for match

sort-merge join

- sort R & S: $2|R|(\log_m(N_R) + 1) + 2|S|(\log_m(N_S) + 1)$
- merge cost: $|R| + |S|$ (worst case $|R| + ||R|| \times |S|$)

optimised sort-merge join

- merge sorted runs until $B > N(R, i) + N(S, j)$; then do merge and join at the same time
- I/O cost: $3 \times (|R| + |S|)$
  - if $B > \sqrt{2|S|}$, one pass to merge initial sorted runs
  - $2(|R| + |S|)$ for initial sorted runs, $|R| + |S|$ for merging

hash join

1. partition R and S into k partitions on join column
   - $\pi_A(R_i) \cap \pi_B(S_j) = \emptyset \quad \forall R_i, S_j, i \neq j$
   - $R = R_1 \cup R_2 \cup \cdots \cup R_k, \quad t \in R_i \iff h(t.A) = i$
   - $S = S_1 \cup S_2 \cup \cdots \cup S_k, \quad t \in S_i \iff h(t.B) = i$
2. join corresponding partitions: $R \bowtie_{R.A=S.B} S = (R_1 \bowtie S_1) \cup \cdots \cup (R_k \bowtie S_k)$

Grace hash join

for *build relation* R and *probe relation* S,

1. **partition** R and S into k partitions each, $k = B - 1$
2. **probing phase**: hash $r \in R_i$ with $h'(r.A)$ to table T
   - 2.1. $\forall s \in S_i, r \in$ bucket $h'(s.B)$: output (r, s) if match

- I/O cost: $3(|R| + |S|)$ (no partition overflow)
- $B > \frac{f \times |R|}{B-1} + 2$ (input & output buffer) $\approx B > \sqrt{f \times |R|}$
  - during probing, $B >$ size of each partition $+2$
- **partition overflow** if R_i cannot fit in memory
  - recursively apply partitioning to overflow partition

General join conditions

- **multiple equality-join** conditions: $(R.A = S.A) \wedge (R.B = S.B)$
  - index nested loop join: use index on some/all join attribs
  - sort-merge join: sort on *combination* of attributes
  - other algos: no change
- **inequality-join** conditions: $(R.A < S.A)$
  - index nested loop join: requires B$^+$-tree index
  - not applicable: sort-merge join (too much rewinding), hash-based joins
  - other algos: no change

06. Query Evaluation & Optimization

- Set operations can be implemented with joins and sort/hash
  - $R(A, B) \cap S(A, B) = \pi_{R.A, R.B} R \bowtie_p S$
  - where p = (R.A = S.A) $\wedge$ (R.B = S.B)
- Sorting for $R \cup S$
  - Sort R and S using all attributes. Merge the sorted operands.
- Hashing for $R \cup S$
  - Same as grace hash join, k partitions, when probing, discard if in, else add. Write to disk.
- Aggregation: maintain running info while scanning table.
  - Use index scan on covering index whenever possible.
- Group by: Partition by grp attr and run normal aggregation.
- **Materialized evaluation**
  - Evaluate operator only when all operands are evaluated or materialized.
  - Intermediate results are written to disk which can be costly.
- **Pipelined evaluation**
  - Execution of operators is interleaved. Pipelining may not always be possible.
  - Temporary relations are not stored on disk.
  - Output produced by operator is passed directly to parent.
  - **Blocking operator** O cannot start until all receive all input tuples from children operators
  - Examples are external merge sort, smj, grace hash.
  - Use **partial materialization** cheaper to read from temp output relation (due to very selective p)
  - Iterator interface.
    - **open**: initialize iterator state: resources, args.
    - **getNext**: gen next tuple, return null when done
    - **close**: Deallocate state information
- A SQL query has many equiv logical plans, which have many equiv physical plans
- Query optimization is about avoiding the worst plans, not picking the best
- Relational Algebra Equiv rules
  1. Commutativity of binary ops
     - 1.1. $R \times S \equiv S \times R$
     - 1.2. $R \bowtie S \equiv S \bowtie R$
  2. Associativity of binary ops
     - 2.1. $(R \times S) \times T \equiv R \times (S \times T)$
     - 2.2. $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$
  3. Idempotence of unary op
     - 3.1. $\pi_{L'}(\pi_L(R)) \equiv \pi_{L'}(R)$ if $L' \subseteq L \subseteq attr(R)$
     - 3.2. $\sigma_{p1}(\sigma_{p2}(R)) \equiv \sigma_{p1 \wedge p2}(R)$
  4. Commutating selection with projection
     - 4.1. $\pi_L(\sigma_p(R)) \equiv \pi_L(\sigma_p(\pi_{L \cup attr(p)}(R)))$
  5. Commutating selection with binary ops
     - 5.1. $\sigma_p(R \times S) \equiv \sigma_p(R) \times S$
     - 5.2. $\sigma_p(R \bowtie_{p'} S) \equiv \sigma_p(R) \bowtie_{p'} S$
     - 5.3. assuming $attr(p) \subseteq attr(R)$
     - 5.4. $\sigma_p(R \cup S) \equiv \sigma_p(R) \cup \sigma_p(S)$
  6. Commutating proj with binary ops

     Let $L = L_R \cup L_S$ where $L_R \subseteq attr(R), L_S \subseteq attr(S)$
     - 6.1. $\pi_L(R \times S) \equiv \pi_{L_R}(R) \times \pi_{L_S}(S)$
     - 6.2. $\pi_L(R \bowtie_p S) \equiv \pi_{L_R}(R) \bowtie_p \pi_{L_S}(S)$ if attr(p) $\cap$ attr(R) $\subseteq L_R$ and attr(p) $\cap$ attr(S) $\subseteq L_S$
     - 6.3. $\pi_L(R \cup S) \equiv \pi_L(R) \cup \pi_L(S)$
- Summary of RA optimization
  1. Perform selection as early as possible, ideally before joins.
  2. Replace Cartesian Product by join whenever possible
  3. Project out useless attributes early
  4. If there are several joins, perform most restrictive join first.
- Types of Query Plan Trees
  - A query plan is **linear** if at least one operand for each join op is a base relation, else it is **bushy**
  - A linear plan is **left-deep** if every right operand is a base relation
  - A linear plan is **right-deep** if every left operand is a base relation
- Query plan enumeration, assume that optimal sol to smaller set is optimal sol to larger set. may not be true.
- DP algorithm for query enumeration: Calculate optimal 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
  - Enumerate only left-deep query plans. (too much to search o/w)
  - Avoid cross-product query plans.
  - Consider early selections and projections.
  - Uses $dp(S_i, o_i)$ where o_i is null if unordered or a sequence of attrs
  - May be cheaper if a SMJ is sorted on some sequence even if by itself is suboptimal.
- Cost estimation
  - Uniformity assumption: uniform distribution
  - Independence assumption: diff attrs are independent
  - Inclusion assumption: For $R \bowtie_{R.A=S.B} S$, if $\|\pi_A(R)\| \leq \|\pi_B(S)\|$ then $\pi_A(R) \subseteq \pi_B(S)$
- **Size estimation**: $\|q\| \approx \|e\| \times \prod_{i=1}^n rf(t_i)$
  - $rf(t_i) = \frac{\|\sigma_i(e)\|}{\|e\|}$, reduction/selectivity factor
  - $rf(R = c) = \frac{\|c\|}{\|R\|}$ uniformity assumption
- **Join selectivity**: $rf(R.A = S.B) = \frac{\|R \bowtie_{R.A=S.B}\|}{\|R\| \times \|S\|}$
  - Inclusion assumption: Assume that $\|\pi_A(R)\| \leq \|\pi_B(S)\|$
  - By uniformity assumption, every R-tuple joins with $\frac{\|S\|}{\|\pi_B(S)\|}$
  - $\therefore rf(R.A = S.B) \approx \frac{1}{\max(\|\pi_A(R)\|, \|\pi_B(S)\|)}$
- Using histograms. Partition domain into sub-ranges (**buckets**) and assume uniform value distribution within each bucket.
  - **Equiwidth**: Each bucket has almost equal num values
  - **Equidepth**: Each bucket has almost equal num tuples, sub-range of adj buckets can overlap
  - **MCV**: Keep track of exact top-k common values and exclude from buckets

07. Transaction

- An active Xact is a Xact still in progress
- **Schedule** = A list of actions from a set of Xacts where the order of the actions within each Xact is preserved
- **Serial Schedule** = A schedule where the actions of Xacts are not interleaved
- We say that T_j **reads O from** T_i in a schedule S if the last write on O before $R_j(O)$ in S is $W_i(O)$
- We say that T_j reads T_i if T_j has read some object from T_i
- We say that T_i performs the final write on O in a schedule S if the last write on O is $W_i(O)$
- An interleaved Xact schedule is **correct** if it is "equivalent" to some serial schedule over the set of Xacts

Two schedules S and S' (over the same set of Xacts) are **view equivalent** denoted by $S \equiv_v S'$ if they satisfy:

1. If T_i reads A from T_j in S, then T_i must also read A from T_j in S'
2. 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'

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

We can also construct a directed VSG(S) to capture read-from and final-write relations among Txn. Each node in VSG repr a Txn with the following edges

1. (T_j, T_i) if T_i reads from T_j
2. (T_j, T_i) if both T_i and T_j update the same O and T_i does final write on O
3. (T_j, T_i) if T_j read O from initial db and T_i update O

If VSG(S) is cyclic, then S is not VSS. If VSG(S) is acylic, then S is VSS iff there exists a serial schedule produced from a topo ordering of VSG(S) that is view equivalent to S.

Two actions on the same O **conflict** if at least one is a W and both are from diff Xacts

1. Dirty read (due to WR conflicts)
   - T_2 reads O modified by T_1 and T_1 has not yet committed.
   - $W_1(x), R_2(x)$
   - T_2 can see an inconsistent DB state
2. Unrepeatable read (due to RW conflicts)
   - T_2 updates O that T_1 reads and T_2 commits while T_1 is still in progress
   - $R_1(x), W_2(x), \text{Commit}_2, R_1(x)$
   - T_1 can get a different value if it reads O again
3. Lost update (due to WW conflicts)

- T_2 overwrites O that was modified by T_1 while T_1 is still in progress
- $R_1(x), R_2(x), W_1(x), W_2(x)$
- T_1's update is lost

4. Phantom read
   - T re-executes a query on a predicate and gets a different set of results due to a recently commited T.
   - Can be prevented by **predicate locking**: Grant T an S on p, another T' request for X on p is blocked. Also see **index locking**

- **conflict equivalent** denoted by $S \equiv_c S'$ if they order every pair of conflicting actions of two **committed** Xacts in the same way.
- **conflict serializable schedule (CS)** if it is conflict equivalent to a serial schedule over the same set of Xacts.
- **Conflict serializability graph** denoted as CSG(S) $= (V, E)$ 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 action.
- **Theorem**: A schedule is **CS** iff its CSG is acyclic.
- **Theorem**: $CSS \subsetneq VSS \subsetneq MVSS$.
- **Note**: CS3223 uses **serializable** to mean **CS**
- A write on O by T_i is a **blind write** if T_i did not read O prior to the write
- **Theorem**: If S is **VS** and S has no blind writes, then S is also **CS**.
- **Cascading abort**: For correctness, if T_i read from T_j then T_i must abort if T_j aborts. We say that T_j's abort is cascaded to T_i.
- **Recoverable schedule**: For every Xact T that commits in S, T must commit after T' if T reads from T'.
- Recoverable schedules guarantee that committed Xacts will not be aborted but cascading aborts of active Xacts are possible.
- **Cascadeless schedule**: If whenever T_i reads from T_j in S, Commit$_j$ must precede this read action.
- **Theorem**: A cascadeless schedule is recoverable.
- **Before-images**: Log the before-images of writes to undo the aborted Xacts. See Chap 10. But this does not always work.
  - $W_1(A), W_2(A)$, Abort$_1$. Undoing $W_1(A)$ is incorrect.
- **Strict**: For every $W_i(O)$ in S, O is not read or written by another Xact until T_i either aborts or commits.
  - Recovery using before-images is more efficient.
  - Concurrent executions are more restrictive.
- **Theorem**: Strict $\subsetneq$ Cascadeless $\subsetneq$ Recoverable.

08. Concurrency Control

Transaction Scheduler: For each input action (R, W, C, A) to the scheduler, either output action to S, postpone(block Xact) or reject (abort Xact)

Lock compatability matrix

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

Lock-Based concurrency control

1. If lock request not granted, T becomes **blocked**, execution is suspended and T is added to O's request queue.
2. When a lock on O is released, lock manager checks request of first T in request queue for O. If can be granted, T acquires lock on O and resumes after popped from queue.
3. When an Xact commits/aborts, all locks are released and T is removed from any request queue it's in.
4. $S_i(O)$: Xact T_i requests S on O
5. $X_i(O)$: Xact T_i requests X on O
6. $U_i(O)$: Xact T_i releases lock on O

Two Phase Locking (2PL) Protocol

1. To read O, T must hold S or X on O.
2. To write O, T must hold X on O.
3. Once T releases a lock, T cannot request anymore.
4. 2PL = growing and shrinking phase.
5. **Theorem:** 2PL S are CS.

Strict 2PL

- Same as 2PL points 1 and 2.
- T must hold onto locks until T commits or aborts.
- **Theorem:** Strict 2PL S are strict and CS.
- **Deadlock Detection: Waits-for graph (WFG)**
  - Nodes represent active T.
  - Add edge $T_i \to T_j$ if T_i waiting for T_j to release lock.
  - Remove edge when lock request is granted.
- Deadlock detected if WFG has a cycle.
- Break deadlock by aborting a T in the cycle.
- Alternative: timeout.
- **Deadlock Prevention**: Older T have higher priority than younger T
  - Each T is assigned ts when started. Older T has a smaller ts.
- Suppose T_i requests for a lock that conflicts with a lock held by T_j
-

Prevention Policy	T_i has higher priority	T_i has lower priority
Wait-die	T_i waits for T_j	T_i aborts
Wound-wait	T_j aborts	T_i waits for T_j

- Wait-die policy
  - non-preemptive: only requesting T can be aborted
  - younger T may abort repeatedly
  - T that has all locks is never aborted
- Wound-wait policy is preemptive
- To avoid starvation, a restarted T must use original timestamp

Lock Conversion Increases concurrency by allowing lock conversions, previously serial only schedules can become interleaved.

- UG$_i(A)$: T_i upgrades S on A to X.
  - Blocked if another T is holding S on A.
  - Allowed if T_i has not released any lock.
- DG$_i(A)$: T_i downgrades X on A to S.
  - Allowed if T_i has not modified A and T_i has not released any lock.

Lock-based Isolation levels

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Read
READ UNCOMMITTED	possible	possible	possible
READ COMMITTED	not possible	possible	possible
REPEATABLE READ	not possible	not possible	possible
SERIALIZABLE	not possible	not possible	not possible

Degree	Isolation level	Write Locks	Read Locks	Predicate Locking
0	Read Uncommitted	long duration	none	none
1	Read Committed	long duration	short duration	none
2	Repeatable Read	long duration	long duration	none
3	Serializable	long duration	long duration	yes

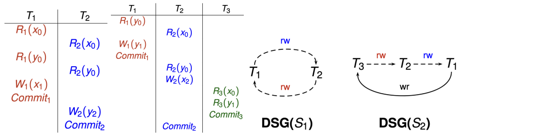
- **Short duration** lock acquired for an operation can be released after operation ends before T commits/abort
- **Long duration** lock acquired for an operation is held until T commits/abort
- **Lock Granularity**: Highest(coarsest) to lowest(finest): db, relation, page, tuple
- If T holds M on D, T implicitly holds M on granules finer than D.
- **Protocol**: Before acquiring S/X on G, acquire IS/IX on G coarser than G top-down. Release locks bottom-up.

09. MVCC

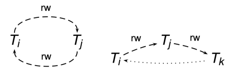
- $W_i(O_i)$: create new version of O denoted by O_i. O_0 is initial version.
- $R_i(O_j)$: read an appropriate version of O
- Read-only T not blocked by update T and vice versa
- Read-only T never aborted

- **Multiversion view equivalent (MVE)** denoted by $S \equiv_{mv} S'$ if they have the same set of reads-from.
- $R_i(x_j)$ occurs in S iff $R_i(x_j)$ also occurs in S'
- **Monoversion schedule**: each read in S returns most recently created version. Also can be serial.
- **Multiversion view serializable s (MVSS)**: a S where there exists a serial monoversion S' that is MVE to S.
- To show not MVSS, suppose there exists a S' and show that there is a cycle in the precede graph.

- **MVCC Protocol: Snapshot Isolation (SI)**
- Each T has two timestamps $start(T)$ and $commit(T)$
- Each T sees a snapshot of DB that consists of updates by T' that committed before T starts
- T and T' are **concurrent** if they overlap
- O_i is a **newer version** compared to O_j if commit$(T_i) >$ commit(T_j)
- If $R_i(O)$ returns O_j then (either own update or latest version of O created by T that committed before T_i starts)
  - $j = i$ if $W_i(O)$ precedes $R_i(O)$ OR
  - commit$(T_j) <$ start(T_i)
  - For every T_k, $k \neq j$, that has created a version O_k of O if commit$(T_k) <$ start(T_i)thencommit$(T_k) <$ commit(T_j)
- **Concurrent update property:** If multiple concurrent T updated same O, only one T allowed to commit. If not, S may not be serializable.
- **First Committer Wins (FCW)**: Before committing T, if another committed concurrent T' that has updated some O that T has updated exists, abort T. Else commit T.
- **First Updater Wins (FUW)**: When T needs to update O, T requests for X. If X is not held by another concurrent T'
  - T is granted X on O. If O has been updated by any concurrent T", T aborts. Otherwise T proceeds.
- Else if X is held by some concurrent T', T waits until T' aborts or commits.
  - If T' aborts, then assume T is granted X on O. If O has been updated by any concurrent T", T aborts. Otherwise T proceeds.
  - If T' commits, then T is aborted.
- (FUW): When T commits/aborts, releases its X-lock(s).
- **Garbage collection** A version O_i may be deleted if there exists a newer version O_j commit$(T_i) <$ commit(T_j) s.t for every **active** T_k that started after T_i committed, we have commit$(T_j) <$ start(T_k)
- SI has similar performance to Read Committed but SI does not suffer from lost update or unrepeatable reads.
- But SI is vulnerable to non-serializable executions such as **write-skew anomaly** (left pic) and **read-only txn anomaly** (middle pic). Both pics are SI S that isnt MVSS
- SI also does not guarantee serializability. The rightmost picture shows the DSG for both anomalies (S1 = write-skew, S2 = read-only)



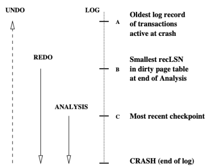
- **Serializable Snapshot Isolation (SSI)**: Guarantees serializable SI schedules.
- Transactional dependencies
  - $T_1 \dashrightarrow_{ww} T_2$: T_1 writes a version of x. T_2 later writes **immediate successor** of x.
  - $T_1 \dashrightarrow_{wr} T_2$: T_1 writes a version of x. T_2 later reads this version of x.
  - $T_1 \dashrightarrow_{rw} T_2$: T_1 reads a version of x. T_2 later writes **immediate successor** of x.
  - x_j is the **immediate successor** of x_i if T_i commits before T_j and no txn that commits between T_i's and T_j's commits produces x.
- Maintain a Dependency Serializable Graph (DSG) to keep track of **rw dependencies** among concurrent T
- DSG(S) = (V, E), V = $\{T_1, ..., T_k\}$ and E
- $\dashrightarrow$ concurrent txn
- $\rightarrow$ non-concurrent txn
- If there exists a T_j is involved in two rw concurrent dependencies, abort one of T_i, T_j, T_k.
- May result in unnecessary rollbacks due to false positives of SI anomalies.
- **Theorem Non-MVSS SI Schedules:** If S is a SI schedule that is **not MVSS**, then
  - There is at least one cycle in DSG(S), and
  - For each cycle in DSG(S), there exists three txns T_i, T_j, T_k where



10. Crash Recovery

- Types of failure
  - Transaction failure: T aborts
  - System crash: loss of volatile memory contents
  - Media failure: data is lost on non-volatile storage
- **Recovery manager**: Supports Commit, Abort and Restart (abort all losers, and install updates from committed T not written to disk)
- desiderata: little overhead and recover quickly
- **Steal policy**: allow dirty pages updated by T to be replaced from buffer pool before T commits.
  - No steal: poor throughput. Steal: how atomic?
- **Force policy**: force dirty pages updated by T to written to disk when T commits
  - poor response time but provide durability
-

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

- 

- **Log** Sequential file of records in **non-volatile/stable storage** (multi copies)
  - LSN: unique identifier, LSN of an earlier log record $<$ later log record
  - type of log record (update, commit, abort)
  - identifier of Xact
  - pageID: what page is modified
  - prevLSN: LSN of the previous log record for same Xact
  - undoNextLSN: See CLR
- **Update log record**
  - id of page being updated, byte offset within page indicating when updated portion start, length: number of byes for updated portion of page
  - before-image and after-image of update
- **Compensation Log Record (CLR)**
  - When an update (described by a log record) is undone, create a CLR. Track action taken to undo update.
  - undoNextLSN = LSN of next log record to be undone (this skips over the undone action in the LL)
- **Abort log record**: When Xact is to be aborted, start undoing this Xact.
- **End log record**: When the extra processing for aborted/committed Xact is done
- **Checkpoint log record**: See checkpointing
- **Transaction Table (TT)**
  - One entry for each **active Txn** which contains
    - XactID
    - lastLSN: LSN of most **recent** log record for this Xact
    - Xact status (C = committed or U = has not committed)
- **Dirty Page Table (DPT)**
  - One entry for each **dirty page** in buffer pool which contains
    - pageID: page ID of dirty page
    - recLSN: LSN of the **earliest** log record for an update that dirtied page
- **Write-ahead logging (WAL) protocol**
- Uncommitted update to db not flushed til log with before-image is flushed.
- Each db page contains LSN of most recent log record (**pageLSN**) that describes update to the page. Before flushing db page p to disk, ensure that all log r' up to log r corresponding to p's pageLSN has been flushed.
- **Force-at-commit protocol**
- Do not commit a Xact until after-images of all its updated records are in stable storage (db or log). Enforced by writing a **commit log record** r for Xact and flushing all log records (up to and including r) for Xact to disk.
- An Xact is considered committed if its commit log is written to stable storage
- **Simple checkpointing**
- Stop accepting any new update, commit and abort ops.
- Wait till all active update, commit and abort ops are done.
- Flush all dirty pages in buffer.
- Write a checkpoint log record containing Xact table.
- Resume accepting new update, commit and abort.
- Analysis phase finds latest checkpoint log and resumes from there, init Xact to checkpoint and DPT to empty.
- **Fuzzy checkpointing**: Used by ARIES in analysis start up.
- Write begin checkpoint log record (BCPLR).
- Write end checkpoint log record (ECPLR) containing current DPT and TT.
- Write a master record containing LSN of begin checkpoint to stable storage.
- We assume there are no log records between BCPLR and ECPLR.

ARIES: Analysis

1. Retrieve begin checkpoint log record (BCPLR) identified by master record
2. Retrieve end checkpoint log record (ECPLR) corresponding to BCPLR
   - We assume there are no log records between BCPLR and ECPLR
3. Initialize DPT and TT using ECPLR's contents
4. Scan log record in forward direction to process each record r
5. If r is an **end log record**
   - Remove T from TT
6. Else
   - Add an entry in TT for T if T is not in TT
   - Update lastLSN of entry T to be r's LSN
   - Update status of entry T to C if r is a commit log record
7. If (r is a redoable log record for page P) and (P is not in DPT) then
   - Create entry for P in DPT with pageID = P's pageID, recLSN = r's LSN
8. At the end of Analysis, 1) TT = list of all active Txn at time of crash, 2) DPT = **superset** of dirty pages at time of crash

ARIES: Redo

1. Set **RedoLSN** = smallest recLSN among all dirty pages in DPT
2. Let r be the log record with LSN = RedoLSN
3. Scan the log in forward dir starting from r
4. If (r is a **redoable** (update or CLR) log record) and (condition C is false)
   - Fetch page P associated with r (this requires a page access!!!)
   - If (P's pageLSN $<$ r's LSN) then
     - Reapply logged action in r to P
     - update P's pageLSN = r's LSN
   - Else
     - Update P's entry in DPT: recLSN = P's pageLSN + 1
5. At the end of Redo Phase, 1) Create **end log records** for Xacts with status = C in TT and remove them from TT, 2) System is restored to state at time of crash. (Still have losers)
6. **Condition C**: (P is not in DPT) or (P's recLSN in DPT $>$ r's LSN)

ARIES: Undo

1. Abort losers (active Xacts at time of crash) by undoing action in reverse order
2. Initialize L = set of lastLSNs (with status = U) from TT
3. Repeat until L becomes empty

   3.1. Delete largest lastLSN from L

   3.2. Let r be the corresponding log record

   3.3. If r is an **update log record** for T on page P
   - i. Create a CLR r_2 for T: r_2's undoNextLSN = r's prevLSN and prevLSN = T's lastLSN in TT
   - ii. Update T's entry in TT: lastLSN = r_2's LSN
   - iii. Undo logged action on page P (WAL principle)
   - iv. update P's pageLSN = r_2's LSN
   - v. $Update-L-and-TT$(r's prevLSN)

   3.4. else if r is a **CLR** for T then
   - i. $Update-L-and-TT$(r's undoNextLSN)

   3.5. else if r is an **abort log record**
   - i. $Update-L-and-TT$(r's prevLSN)

   3.6. **def** $Update-L-and-TT$(lsn)
   - i. if lsn is not null then add lsn to L
   - ii. else create an **end log record** for T and remove T from TT