CS3223 Finals Cheatsheet

by: Jiahao

▷ notation

r	relational algebra expression
$\lVert r\rVert$	# tuples in output of r
$\lvert r\rvert$	# pages in output of r
b_d	# data records (full record) on page
b_i	# data entries ((k, RID)) on page
F	average # pointers to child nodes
h	height of B^+-tree index
B	# available buffer pages
p'	primary conjunct
p_c	covering conjunct
N_0	initial sorted runs
$A(\cdot)$	attributes of relation/predicate
T_i	Xact i
$R_i(O)$	T_i reading object O
$W_i(O)$	T_i writing object O
Commit_i	T_i terrminates successfully
Abort_i	T_i terrminates unsuccessfully
$S_i(O)$	T_i requests for S-lock on O
$X_i(O)$	T_i requests for X-lock on O
$U_i(O)$	T_i releases lock on O

▷ cost analysis notes

- if format 2, include cost of RID lookups unless covering index
- if format 2/3, use b_i instead of b_d
- if unclustered, duplicate cost of page lookup
- if clustered, cost of RID lookups → 1 per page
- think in terms of cost to read/write separately
- blocked I/O → read/write in blocks → 1 I/O

▷ external merge sort

given a file of N pages with B buffer pages ($B \geq 3$)

1. creation of sorted runs (temporary tables; pass 0): read and sort B pages in memory
   - $N_0 = \lceil N/B \rceil$ sorted runs; $\leq B$ pages per run
2. subsequent passes: merge sorted runs with $B-1$ pages for input and 1 page for output

analysis: $2 \times N$ read/writes $\times$ # passes

$$2N(\lceil \log_{B-1} N_0 \rceil + 1)$$

blocked I/O: reading/writing in units of buffer blocks of b pages

- trade-off: maximizing merge factor with reducing I/O cost
- exploits speed of sequential I/O
- given N pages, B buffer pages, b block size

$$N_0 = \lceil N/B \rceil; \left\lceil \log_{\lfloor B/b \rfloor + 1} N_0 \right\rceil + 1 \text{ passes}$$

- given j input buffers of m size each, and k output pages, s seek time, and r rotational delay

$$\left\lceil \log_j \lceil N/B \rceil \right\rceil \times N/m + (s + r + k)$$

achieving k merge passes:

$$N_0 = \lceil N/B \rceil \geq B - 1 \Rightarrow B^2 - B - N \leq 0$$

with B^+-tree: depends on format

- format 1: scan leaf pages and return
- format 2/3: scan leaf pages, for each leaf page, retrrieve data records by RID

▷ selection $\sigma_p(R)$

covering index: I for Q if all attributes referenced in Q are part of the key/"include column" of I

- $\Rightarrow Q$ evaluated without RID lookups

index scan: find leftmost element that satisfies query and traverse leaf pages till predicate is no longer true; covering/format 1 index

index scan + RID lookups: for each element, perform an RID lookup; non-covering/format 2/3 index

index intersection: perform index scans for each predicate and find intersection fo results (by RID)

- useful for conjunctive predicates
- does not require same index

CNF predicate: disjunction ($\vee$) and conjunction ($\wedge$)

- term: $R.A$ op c or $R.A_i$ op $R.A_j$
- conjunct: $X_1 \wedge X_2 \wedge ... \wedge X_n$
- disjunctive conjunct: $(a_1 \vee ... \vee a_m) \wedge (b_1 \vee ... \vee b_n)$...

matching predicate (B^+-tree): given $I = (K_1, K_2, ..., K_n)$ and predicate p without OR

$$\forall i \in [1, n], (K_1 = c_1) \wedge ... \wedge (K_i \mathtt{op}_i c_i)$$

- follows I order (no skipping attributes)
- zero or more equality predicates
- at most 1 non-equality predicate (must be last predicate)
- when match, use index scan since data in contiguous order

matching predicate (hash):

$$(K_1 = c_1) \wedge (K_2 = c_2) \wedge ... \wedge (K_n = c_n)$$

- must be fully covered by equality predicates
- cannot work with range based predicates

primary conjunct (p'): $p' \subseteq p$ that I matches

- re-arrange predicates to figure out

covering conjunct (p_c): all attributes in C in p in the key/include column of I; $p' \subseteq p_c$

- need not match

evaluating non-disjunctive conjucts: predicate without OR; table scan, hash index scan, B^+-index scan, index intersection

evaluating disjunctive conjuncts: use covering index/ index intersection, or table scan

cost of B^+-tree index:

1. navigate internal nodes to locate first leaf: height of tree
2. scan leaf pages to access qualifying data entries: number of pages containing qualifying data entries
3. retrieve qualified data records via RID lookups: number of records
   - if format 1 or covering index, then this step counts for nothing

- step (3) cost can be reduced with clustered index since retrieving 1 page → read all records

$$\min\left\{\lVert\sigma_{p_c}(R)\rVert, |R|\right\}$$

- no primary conjunct → perform full leaf scan

cost of hash index: may be more due to thrashing/ collisions

- at least $\left\lceil \lVert\sigma_{p'}(R)\rVert \,/ b_d \right\rceil$ (number of buckets)
  - format 2 → use b_i + cost to retrieve data records
    - 0 if I is covering index, else $\lVert\sigma_{p'}(R)\rVert$
- range scan → table scan
- maximum I/O with overflow: $1 + \#$ overflow pages

▷ projection $\pi_L(R)/\pi^*_L(R)$

↳ projection using sorting

1. extract attributes L from records: cost to scan and output temporary table $|R| + |\pi^*_L(R)|$
2. sort records using attributes of L as sort key: cost of external merge sort $2\,|\pi^*_L(R)|\,(\log_m N_0 + 1)$
3. remove duplicates (optional if π^*_L): cost to read sorted entries and write $|\pi^*_L(R)| + |\pi_L(R)|$

optimization: break step 2 and merge with step 1 & 3

1. create sorted runs with attributes L
2. merge sorted runs and remove duplicates

strengths: results are sorted; good if many duplicates or distribution of hashed values is non-uniform (likely to overflow)

analysis: if $B > \sqrt{|\pi^*_L(R)|}$, perform similar to hash

- $N_0 = \lfloor |R| \,/ B \rfloor \approx \sqrt{|\pi^*_L(R)|}$
- $\log_{B-1} N_0 \approx 1$ merge passes

↳ projection using hashing

building main-memory hash table T to detect and remove duplicates; cost $|R|$ if T fits in memory

- initialize empty hash table T
- for each tuple $t \in R$ do
  - apply hash function $h(\cdot)$ on $\pi_L(t)$
  - let t be hashed to bucket B_i in T
  - if $\pi_L(t) \notin B_i$ then insert $\pi_L(t)$ into B_i
- output all entries in T

partitioning phase: create partitions $R_1, R_2, ..., R_{B-1}$

- $\pi^*_L(R_i) \cap \pi^*_L(R_j) = \emptyset, i \neq j$
- uses 1 buffer for input, $B-1$ for output
- read R one page at a time into input buffer
- for each tuple t
  - project out unwanted attributes: $t \to t'$
  - $h(t')$ to distribute to one of output buffers
  - flush output buffer to disk when full

duplication elimination phase:

- may be done in parallel
- for each partition R_i
  - initialize in-memory hash table
  - read $\pi^*_L(R_i)$ one page at a time; for each tuple t
    - $h'(t)$ to bucket B_j ($h' \neq h$)
    - insert t into B_j if $t \notin B_j$
  - output tuples in hash table

partition overflow: hash table for $\pi^*_L(R_i)$ larger than buffer size

- recursively apply hash-based partitioning to overflowed partition

analysis: effective if B is large relative to $|R|$

- B size: assume uniform distribution with $|R_i| = |\pi^*_L(R)| \,/ B-1$ and size of hash table for R_i $|R_i| \times f$ (fudge factor)
  - $\approx B > \sqrt{f \times |\pi^*_L(R)|}$ to avoid partition overflow
- cost: assume no partition overflow
  - cost of partitioning phase: $|R| + |\pi^*_L(R)|$
  - cost of duplicate elimination phase: $|\pi^*_L(R)|$

↳ projection using indexes

- replace table scan with index scan iff $\exists I$ with search key containing all projected attributes
- if index is ordered by projected attributes, then no sorting needed
  - scan data entries in order and compare adjacent entries for duplicates
- if index is not covering, create k partitions of distinct keys, sort individually, and merge
  - lower I/O cost overall

▷ join $R \bowtie_\theta S = S \bowtie_\theta R$

considerations:

- type of join predicate: equality/inequality
- size of join operands
- available buffer space
- available access methods

general points:

- cost analysis ignores write and assumes R is outer
- outer relation should always have less records
- most optimal join: in-memory $|R| + |S|$

multiple equality join conditions: some alterations

- index nested loop join: use index on all or some of join attributes
- sort-merge join: sort on combination of attributes

inequality join conditions: some alterations

- index nested loop join: index must be B^+-tree index
- cannot use sort-merge or hash-based joins (hash index)

↳ tuple-based $|R| + \lVert R\rVert \times |S|$

for each tuple $r \in R$, for each tuple $s \in S$

↳ page-based $|R| + |R| \times |S|$

for each page $P_r \in R$, for each page $P_s \in S$, for each tuple $r \in P_r$, for each tuple $s \in P_s$

- brings page into memory so it's faster for I/O
- minimum 3 pages required

↳ block nested $|R| + (\lceil |R| \,/ B-2 \rceil \times |S|)$

repeat till no more pages in R: read $B-2$ pages of R into memory, for each page $P_s \in S$, read P_s into memory. for each tuple $r \in R$ (in memory), for every tuple $s \in P_s$

- allocates $B-2$ for R, 1 for S and 1 for output

↳ index nested $|R| + \lVert R\rVert \times J$

for each tuple $r \in R$, use r to probe S index to find matching tuples

- requires index on inner relation
- analysis assumes uniform distribution of matches for outer loop with inner loop & format 1 B^+-tree
- J is the height of tree + search for leaf nodes

$$J = \log_F \lceil \lVert S\rVert \,/ b_d \rceil + \left\lceil \lVert S\rVert \,/ \left(b_d\, \lVert\pi_{B_j}(S)\rVert\right) \right\rceil$$

↳ sort-merge join $2\,|R|\,(\log_m(N_R) + 1) + 2\,|S|\,(\log_m(N_S) + 1) + (|R| + |S|)$

sort both relations based on join attributes and merge them

- sorted relation R consists of partitions R_i of records where $r, r' \in R_j$ iff r and r' have the same values for the join attributes
- each tuple $r \in R_i$ merges with all tuples $s \in S_i$
- two-pointer approach where we match all of $r \in R$ with matching entries of $s \in S$
  - rewind S pointer to beginning of repeats/partition

R:	2	5	7	10	10	13	
S:	4	5	5	10	10	18	22
$R \bowtie S$:	(5,5)	(5,5)	(10,10)	(10,10)	(10,10)	(10,10)	

analysis: cost to sort R + cost to sort S + merging cost

- each partition in S scanned at most once: $|R| + |S|$
- worst case: every tuple in R needs to rescan S (i.e. cross product): $|R| + \lVert R\rVert \times |S|$

optimization: combine merge phase of sorted runs into single run before performing join

- $B > N(R, i) + N(S, j) \Rightarrow$ sorting can stop
  - $N(R, i)$: total # sorted runs of R after pass i
- merged sorted runs of R and S partially, then merge remaining sorted runs of $R\&S$ and join them
- analysis: assume $|R| \leq |S|$ and $B > \sqrt{2|S|}$
  - number of initial sorted runs of $S < \sqrt{|S| \,/2}$
  - total number of initial sorted runs of R and $S < \sqrt{2|S|}$
  - 1 pass sufficient to merge and join initial runs
  - I/O cost: $3 \times (|R| + |S|)$

↳ (grace) hash join $3 \times (|R| + |S|)$

partition R (build relation) and S (probe relation) into k partitions using hash function $h(\cdot)$ and join corresponding pair of partitions

$$R \bowtie S = (R_1 \bowtie S_1) \cup (R_2 \bowtie S_2) \cup ... \cup (R_k \bowtie S_k)$$

probing phase: probes each R_i with S_i

- read R_i to build a hash table, read S_i to probe hash table
  - hash tuples $r \in R_i$ with h' ($h' \neq h$)
  - for each tuple $s \in S_i$, output (r, s) for all tuple r in the same bucket as $h'(s)$ and match

analysis: minimize size of each partition of R_i by using $k = B - 1$

- assume uniform hashing distribution
  - size of partition $R_i : N = |R|/k$
  - size of hash table for $R_i : M = f \times N$ (fudge factor)
  - during probing, $B > M + 2$ (input & output for S_i)
  - $\approx B > \sqrt{f \times |R|}$
- I/O cost: cost of partitioning + cost of probing

partition overflow: hash table R_i does not fit in memory

- recursively apply partitioning to overflow partitions

▷ other set operations $R \cup S; R \setminus S$

- sorting: sort R and S with all attributes; merge sorted operands and discard duplicates
- hashing: similar approach as Grace Hash join

▷ aggregation

↳ simple aggregation

maintain some running information while scanning

aggregate operator	running information
SUM	total of retrieved values
COUNT	count of retrieved values
AVG	(total, count) of retrieved values
MIN	smallest retrieved value
MAX	largest retrieved value

↳ group-by aggregation

sorting: sort relation on gropuing attribute(s); scan sorted relation to compute aggregate for each group

hashing: scan relation to build hash table on grouping attribute(s); for each group, maintain (grouping value, running information)

↳ using index

- use covering index directly to avoid table scan
- if group-by attributes is a prefix of a B^+-tree search key, retrieve each group without explicit sorting

▷ query evaluation

↳ materialized

operator is evaluated only when each of its operands have been completely evaluated/materialized

- intermediate results materialized to disk; stored as temporary tables

↳ pipelined

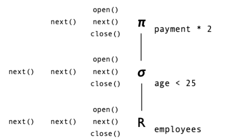
output produced by operator passed directly to parent operator

- execution of operators is interleaved; partial results sent to parent
- blocking operator: O may not be able to produce output until it has received all input tuples from child operator(s)
  - e.g. external merge sort, sort-merge join, Grace hash join

iterator interface: each operator has the following interface

1. `open()`: initializes/resets state of iterator, preparing to deliver first result tuple; contains references to children operators
2. `getNext()`: generates next output tuple; returns `null` when done
3. `close()`: deallocates state information

- initiated by driver which calls `open()` on top-most operator which subsequently calls `next()` on itself and its children



partial materialization: materialize operands that have to be evaluated multiple times

- may have lower I/O cost

▷ query plan optimization

a SQL query has many logical plans and each logical plan has many physical plans

key components:

1. search space: what space of query plans considered?
   - make assumptions to restrict search space
     - e.g. only support hash join, avoid cartesian product, no bushy trees
2. plan enumeration: how to enumerate space of query plans
3. cost model: how to estimate cost of plan

↳ relational algebra equivalence rules

used to transform a query plan to an equivalent other

1. commutativity of binary operators
   1. $R \times S \equiv S \times R$
   2. $R \bowtie S \equiv S \bowtie R$
2. associativity of binary operators
   1. $(R \times S) \times T \equiv R \times (S \times T)$
   2. $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$
3. idempotence of unary operators
   1. $L' \subseteq L \subseteq A(R) \Rightarrow \pi_{L'}(\pi_L(R)) \equiv \pi_{L'}(R)$
   2. $\sigma_{p_1}\left(\sigma_{p_2}(R)\right) \equiv \sigma_{p_1 \wedge p_2}(R)$ (index + RID)
4. commutating selection with projection: pushing projection after selection
   1. $\pi_L(\sigma_p(R)) \equiv \pi_L\left(\sigma_p\left(\pi_{L \cup A(p)}(R)\right)\right)$
5. commutating selection with binary operators
   1. $A(p) \subseteq A(R) \Rightarrow \sigma_p(R \times S)$
   2. $A(p) \subseteq A(R) \Rightarrow \sigma_p\left(R \bowtie_{p'} S\right) \equiv \sigma_p(R) \bowtie_{p'} S$ (reduces cost of join)
   3. $\sigma_p(R \cup S) \equiv \sigma_p(R) \cup \sigma_p(S)$
6. commutating projection with binary operators: $L = L_R \cup L_S; L_R \subseteq A(R), L_S \subseteq A(S)$
   1. $\pi_L(R \times S) \equiv \pi_{L_R}(R) \times \pi_{L_S}(S)$
   2. $A(p) \cap A(R) \subseteq L_R \wedge A(p) \cap A(S) \subseteq L_S \Rightarrow \pi_L\left(R \bowtie_p S\right) \equiv \pi_{L_R}(R) \bowtie_p \pi_{L_S}(S)$
   3. $\pi_L(R \cup S) \equiv \pi_L(R) \cup \pi_L(S)$

↳ types of query plan trees

linear: at least 1 operand for each join operation is a base relation

- left(/right)-deep if every right(/left) join operand is a base relation

bushy: not linear; requires at least 4 tables/3 joins

↳ query plan enumeration $O(3^n)$

given $\{R_1, R_2, ..., R_n\}$, let $\mathrm{OPT}(m)$ be the optimal cost given s relations used denoted by bitmask m

base case: $\mathrm{OPT}(2^i) =$ best access plan for R_i

recurrence: for all possible masks, try all compositions of the mask

$$\mathrm{OPT}(m) = \min_{\forall l, r: l \cup r = m} \{\mathrm{OPT}(l) + \mathrm{OPT}(r) + \mathrm{cost}(l \bowtie r)\}$$

↳ system R optimizer

heuristics: enumerates only left-deep query plans, avoids cross product query plans, considers early selections and projections

enhanced dp: consider sort order of query plan

- $\mathrm{OPT}(m, o)$ where o is the sort order produced by optimal query plan for m
  - `NULL` if output unordered or sequence of attributes
  - cheapest query plan for m with output ordered by o if $o \neq$ `NULL`

▷ cost estimation

involves evaluation cost of each operation and output size of each operation

- cost model depends on size of input operands, available buffer pages, and available indices

assumptions:

- uniformity: uniform distribution of attributes
- independence: independent distribution of values in different attributes
- inclusion: for $R \bowtie_{R.A = S.B} S$, $\lVert\pi_A(R)\rVert \leq \lVert\pi_B(S)\rVert \Rightarrow \pi_A(R) \subseteq \pi_B(S)$

database statistics:

- relation cardinality
- number of distinct values in each column
- highest/lowest values in each column
- frequent values of some columns
- column group statistics
- histograms

↳ size estimation $\lVert q\rVert$

↝ size estimation of projection

suppse $q = \sigma_p(e)$ where $p = t_1 \wedge ... \wedge t_n$ and $e = R_1 \times ... \times R_m$

- t_i filters out some tuples in e
- reduction factor of t_i ($\mathrm{rf}(t_i)$): fraction of tuples in e that satisfies t_i: $\mathrm{rf}(t_i) = \lVert\sigma_{t_i}(e)\rVert / \lVert e\rVert$; aka selectivity factor
- assuming t_i are statistically independent

$$\lVert e\rVert = \prod_{i=1}^{m} \lVert R_i\rVert$$

$$\lVert q\rVert \approx \lVert e\rVert \times \prod_{i=1}^{n} \mathrm{rf}(t_i)$$

estimating $\mathrm{rf}(t_i)$: using uniformity assumption

- may have high margin of error for large $R.A$ that has higher frequencies

$$\mathrm{rf}(t_i) \approx 1 / \lVert\pi_A(R)\rVert$$

↝ size estimation of join

suppose $q = R \bowtie_{R.A = S.B} S$ with $\mathrm{rf}(R.A = S.B) = \lVert q\rVert / (\lVert R\rVert \times \lVert S\rVert)$

- by inclusion assumption, $\forall r \in R, \exists s \in S : r.A = s.B$
- by uniformity assumption, $\lVert S\rVert / \lVert\pi_B(S)\rVert$ tuples in S have $S.B$

$$\lVert q\rVert \approx \lVert R\rVert \times \frac{\lVert S\rVert}{\lVert\pi_B(S)\rVert}$$

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

↳ histograms

partition attribute's domain into sub-ranges (buckets) and assume value distribution within each bucket is uniform

- more buckets → better estimation

estimation formula: for all buckets that contain value, sum of (# tuples in bucket × # value matches in bucket ratio)

- for equality predicates, # value matches is always $\frac{1}{m}$
- for range predicates, depends on number of matches

↝ equiwidth histogram

given n buckets and m values for attribute, each bucket has $\lfloor m/n \rfloor$ values

↝ equidepth histogram

given n buckets and k tuples, each bucket should contain at most $\lceil k/n \rceil$ values

- if a single value has more than k/n tuples, then split it across multiple buckets

↝ histogram with MCV

separately keep track of frequencies of the top-k most common values (MCV) and exclude MCV from histogram buckets

- if bucket contains MCV value, the # values matches ratio calculation should omit the MCV value(s)
- separately add MCV value if predicate contains MCV

▷ transaction (Xact) management

abstraction representing logical unit of work

- all Xacts read and write from T_0 if no other concurrent Xact has written yet

ACID: maintain data even with concurrent access and system failures

1. **Atomicity:** either all or none of actions happen
2. **Consistency:** if each Xact is consistent, then the database starts and ends up consistent
3. **Isolation:** execution of one Xact is isolated from other Xacts
4. **Durability:** if a Xact commits, its effects persist

transaction schedule: list of actions from a set of Xacts where the order of actions within each Xact is preserved

- serial schedule: actions not interleaved; guaranteed to be consisted
- T_j reads O from T_i if last write on O before $R_j(O)$ is $W_i(O)$: $W_i(O) \to R_j(O)$
- T_j reads from T_i if T_j reads any objects from T_i (`rf`)
- T_i performs the final write on O if the last write action on O is $W_i(O)$ (`fw`)

  $R_1(A), W_1(A), R_2(A), W_2(A), \mathrm{Commit}_1, \mathrm{Commit}_2$
  - T_2 reads A from T_1
  - T_2 has the final write on A
- blind write: T_i does not read O before writing to it

correctness: equivalent to some serial schedule

- `rf` and `fw` must be the same

↳ view equivalence

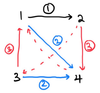
view equivalent: $S \underset{v}{\equiv} S'$ if:

- `rf` are maintained; T_i reads A from T_j in both
- `fw` are maintained; final write of A by T_i in both

view serializable schedule (VSS): S is view equivalent to some serial schedule

heuristic for view serializability: VSG(S) capturing `rf` and `fw` relations among Xacts
- nodes: Xacts
- edges: precedence relations
- rules:
  1. T_i `rf` T_j: $T_j \to T_i$
  2. T_i and T_j update object O & T_i `fw` O: $T_j \to T_i$
  3. T_j reads O from T_k & T_i updates O: $T_i \to T_k$ OR $T_j \to T_i$ (depending on whichever does not create a cycle)
- interpretation: cyclic implies not VSS; acyclic implies VSS iff a there exists a serial schedule produced from topological ordering of VSG(S) that is view equivalent to S

$$W_1(x), R_2(x), R_3(y), W_3(x), W_2(y), W_4(x)$$



↳ conflicts

conflicting actions: on same object, at least one is a write action, actions are from different Xacts

anomalies of interleaved transactions: due to conflicting actions
1. dirty read (`dr`): $W_1(O), R_2(O), ..., \text{Commit}_1$
   - reading object produced by uncommitted Xact
   - T_2 may see inconsistent database state
2. unrepeatable read (`ur`): $R_1(O), W_2(O), \text{Commit}_2, R_1(O)$
   - T_1 may get different value if reading again
3. lost update (`lu`): $W_1(O), W_2(O)$
   - T_1 update lost
   - commits can happen in between
4. phantom read: transaction re-executes query for rows that satisfy search condition and finds rows change due to another committed transaction

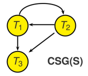
conflict equivalent: $S \underset{c}{\equiv} S'$ if every pair of conflicting actions are ordered the same

conflict serializable schedule (CSS): S is conflict equivalent to some serial schedule; commonly referred to as "serializable"
1. CSS iff CSG(S) is acyclic
2. CSS implies VSS
3. if VSS and no blind writes, then CSS

heuristic for conflict serializability: CSG(S) capturing conflicting actions
- nodes: every committed Xact
- edges: conflicting actions
- rules:
  - action in T_i precedes & conflicts with T_j: $T_i \to T_j$

$$R_1(A), W_2(A), \text{Commit}_2, W_1(A), \text{Commit}_1, W_3(A), \text{Commit}_3$$

conflicts: $T_1 \underset{\text{ur}}{\to} T_2$, $T_2 \underset{\text{lu}}{\to} T_1$, $T_2 \underset{\text{lu}}{\to} T_3$, $T_1 \underset{\text{lu}}{\to} T_3$



↳ more schedules

cascading aborts: if T_i `rf` T_j, then T_i must abort if T_j aborts
- ensures correctness
- undesirable due to cost of bookkeeping to identify and performance penalty incurred
- avoided by permitting reads only from committed Xacts

recoverable schedule: for every Xact T that commits in S, T must commit after T' if T reads from T'
- guarantees that committed Xacts will not be aborted
- cascading aborts still permitted

cascadeless schedule: whenever T_i reads from T_j, Commit_j must precede the read action
1. cascadeless schedule implies recoverable schedule

recovery using before-images: restoring before-images for writes; $W_i(x, v)$ denotes that T_i updates the value of object x to v
- before-image is the value of the object prior to $W_i(x, v)$
- may not always work
- enabled by using strict schedule

strict schedule: for every $W_i(O)$, O is not read or written by another Xact until T_i aborts or commmits
- performance tradeoffs: recovery using before-images is more efficient but concurrent exeutions become more restrictive
1. strict schedule implies cascadeless schedule

▷ concurrency control

transaction scheduler: per input action (read, write, commit, abort), performs
1. output action to schedule
2. postpone action by blocking the Xact
3. reject the action and abort the Xact

↳ lock-based concurrency control

every Xact needs to request for an appropriate lock on an object beefore the Xact can access the object

goals: produces conflict serializable schedules

locking modes:
- shared lock (S) for reading objects
- exclusive lock (X) for writing objects

lock compatibility:

Lock Requested	Lock Held		
	-	S	X
S	✓	✓	✗
X	✓	✗	✗

rules:
1. to read object O, Xact request for S/X-lock on O
2. to update object O, Xact request for X-lock on O
3. if requesting lock mode compatible with lock modes of existing locks on O, lock request is granted
4. if lock request not granted, T is blocked, execution suspended, added to O's request queue
5. when lock released, lock manager checks request of first Xact T on request queue, if it can be granted, T acquires the lock and resumes execution
6. when Xact commits/aborts, all its locks are released and T is removed from any request queue it is in

⇝ two phase locking (2PL) protocol

- rules:
  - to read O, Xact must hold S/X-lock on O
  - to write O, Xact must hold X-lock on O
  - once Xact released a lock, it cannot request any more locks
- transactions have 2 phases:
  - growing phase: before releasing first lock
  - shrinking phase: after releasing first lock
1. 2PL schedule implies conflict serializable

⇝ strict 2PL protocol

used in practice
- rules:
  - to read O, Xact must hold S/X-lock on O
  - to write O, Xact must hold X-lock on O
  - Xact must hold onto locks until Xact commits or aborts
1. strict 2PL schedules implies strict & conflict serializable

↳ deadlocks

cycle of Xacts waiting for locks to be released by each other
- ways to deal with: deadlock detection & deadlock prevention

deadlock detection: create "Waits-for graph" (WFG) where nodes are active transactions
- edge $T_i \to T_j$ if T_i is waiting for T_j to release a lock
- lock manager
  - adds an edge when lock request queued
  - updates edges when lock request granted
- deadlock detected if WFG has a cycle
- break deadlock by aborting Xact in cycle
- alternative: timeout mechanism

deadlock prevention: assume older Xacts have higher priority than younger Xacts
- Xact assigned timestamp when it starts, older Xact have smaller timestamp
- when T_i requests for lock that conflicts with lock held by T_j
- resolutions:
  - wait-die policy: lower priority Xacts never wait for high priority Xacts
    - non-preemptive, younger Xact may be repeatedly aborted, Xact with all the locks is never aborted
    - restarted Xact should use original timestamp
  - wound-wait policy: higher-priority Xact never wait for lower-priority Xacts
    - preemptive

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

lock conversion: increases concurrency
- $\text{UG}_i(A)$: T_i upgrades S-lock on A to X-lock
  - blocked if another Xact holds shared lock on A
  - allowed if T_i has not released any lock
    - leads to conflict serializability
- $\text{DG}_i(A)$: T_i downgrades X-lock on A to S-lock
  - allowed if T_i has not modified A and T_i has not released any locks
- increases number of possible interleaved executions

performance: Xact conflicts resolved via blocking and aborting mechanisms
- blocking causes delays in other waiting Xacts
- aborting and restarting Xacts wastes work done by Xact
- improving:
  - reducing lock granularity
  - reducing time lock is held
  - reducing hot spots (i.e. frequently accessed and modified objects)

concurrency control anomalies: *anomalies of interleaved transactions* prevented with lock-based protocol except **phantom read**

phantom read:
- $R(p)$ reads all objects that satisfy predicate p
- $R(p), W(x)$ conflicts if object x satisfies selection predicate p
- prevented using predicate locking or index locking (in practice)

↳ isolation levels

Isolation Level	Dirty Read	Unrepeatable read	Phantom Read
READ UNCOMMITTED	✓	✓	✓
READ COMMITTED	✗	✗	✗
REPEATABLE READ	✗	✗	✓
SERIALIZABLE	✗	✗	✗

Degree	Isolation Level	Write Locks	Read Locks	Predicate Locking
0	RC	long	none	none
1	RU	long	short	none
2	RR	long	long	none
3	S	long	long	yes

short duration lock: lock released after end of operation before Xact commits/aborts

long duration lock: lock held until Xact commits/aborts

↳ locking granularity

refers to size of data items being locked
- (highest to lowerst) database, relation, page, tuple
- allow multi-granular lock
  - if Xact T holds lock mode M on data granule D, then T also holds lock mode M on granules finer than D
- locking conflicts detected using intention locks (I-lock)

intention locks: before acquiring S/X-lock on data granule G, acquire I-lock on granules coarser than G in top-down manner

Lock Requested	Lock Held			
	-	I	S	X
I	✓	✓	✗	✗
S	✓	✗	✓	✗
X	✓	✗	✗	✗

finer intention locks: locks acquired in top-down order
- IS: intent to set S-lock at finer granularity
- IX: intent to set X-lock at finer granularity
- rules:
  - to obtain S or IS lock on node, IS or IX must be on parent node
  - to obtain X or IX lock on node, IX must be on parent node
  - locks acquired in top-down order, but released in bottom-up order

Lock Requested	Lock Held				
	-	IS	IX	S	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✓	✗	✗
S	✓	✓	✗	✓	✗
X	✓	✗	✗	✗	✗

↳ multiversion concurrency control

maintain multiple versions of each object; does not require locks

notation:
- $W_i(O)$ creates new version of O, denoted by O_i
- $R_i(O)$ reads an appropriate version of O
- initial version: O_0

advantages: read-only Xacts are not blocked by update Xacts and vice versa; read-only Xacts are never aborted

multiversion schedules: schedules differ based on version of object read

multiversion view equivalent: $S \underset{\text{mv}}{\equiv} S'$ if they have the same set of `rf` relationship
- no notion of `fw` since updates are not done in-place

monoversion schedules: every read action in S returns the most recently created object version

serial monoversion schedule: monoversion schedule that is also serial schedule

multiversion view serializable schedule (MVSS): there exists a serial monoversion schedule that is multiversion view equivalent to S
1. VSS implies MVSS (but converse is not always true)
   - there may be issues that can only happen because of multiversion system

⇝ snapshot isolation

each Xact T sees a snapshot of the database that consists of updates by Xacts committed before T starts
- every Xact associated with start(T) and commit(T)
- $W_i(O)$ creates a version of O, O_i
  - O_i more recent compared to O_j if commit(T_i) > commit(T_j) (i.e. T_i committed later than T_j)
- $R_i(O)$ reads either own update or latest version of O created before T_i started

concurrent transactions: concurrent if they overlap: $[\text{start}(T), \text{commit}(T)] \cap [\text{start}(T'), \text{commit}(T')] \neq \emptyset$

concurrent update property: if multiple concurrent Xacts updated the same object, only one of them is allowed to commit
- otherwise schedule may not be serializable
- enforced using First Committer Wins (FCW) rule or First Updater Wins (FUW) rule

First Committer Wins (FCW): before committing Xact T, 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, else commits

First Updater Wins (FUW): whenever Xact T needs to update O, T requests for X-lock on O; all X-locks released upon Xact abort/commit
- if X-lock not held by concurrent Xact, then lock granted
  - if O has been updated by any committed concurrent Xact (i.e. value not the same as snapshot value), T aborts
  - otherwise, T proceeds with execution
- else, T waits till T' (holding X-lock) to abort or commit
  - if T' aborts, then if O has been updated by any committed concurrent Xact, abort T
    - else, T proceeds
  - elif T' commits, T aborts

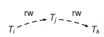
garbage collection: version O_i may be deleted if there exists a new version O_j (commit(T_i) < commit(T_j)) such that for every active Xact T_k that where commit(T_i) < start(T_k), there is commit(T_j) < start(T_k)

tradeoffs:
- similar performance to READ COMMITTED
- does not suffer from lost update or unrepeatable read anomalies
- vulnerable to some non-serializable executions: write skew anomaly and read-only transaction anomaly
- snapshot isolation does not guarantee serializability

write-skew anomaly: value written in Xact T_1 after Xact T_2 has started so the snapshot of T_2 is outdated

read-only transaction: Xact T_3 starts after T_1 is committed but while T_2 is running, so it has the latest values of T_1 but not the latest values of T_2

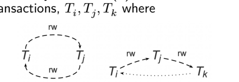
serializable snapshot isolation (SSI) protocol: S is a SI schedule and S is MVSS
- may have false positives
- guarantees serializable SI schedules
  - keep track of `rw` dependencies among concurrent Xacts
  - detect formation of T_j involving 2 `rw` dependencies
  - once detected, abort one of the Xacts involved
  - may result in unnecessary rollbacks due to false positives



transaction dependencies:
1. `ww`: T_1 writes x_i, T_2 writes x_j
2. `wr`: T_1 writes x_i, T_2 reads x_i
3. `rw`: T_1 reads x_i, T_2 writes x_j

- x_j is the immediate successor of x_i if
  - commit(T_i) < commit(T_j)
  - no transaction commits between T_i and T_j producing a version of x

heuristic for SSI: Dependency Serialization Graph (DSG)
- $S = \{T_1, ..., T_k\}$, $V = S$
- edges: $T_i \xrightarrow{\text{ww}} T_j$, $T_i \xrightarrow{\text{wr}} T_j$, $T_i \xrightarrow{\text{rw}} T_j$
  - → for non-concurrent transaction pair, ⇢ for concurrent
- `rw` order of action does not matter, only focus on immediate successor version (does T_2 write a successor to T_1)
- `ww` and `wr` can only occur between non-concurrent transactions
- `rw` can only occur between concurrent transactions
1. if S is a SI schedule that is not MVSS, then
   1. there is at least 1 cycle in DSG(S)
   2. for each cycle in DSG(S), there exists three transactions, T_i, T_j, T_k where



▷ recovery

undo: remove effects of aborted Xact for atomicity

redo: re-install effects of committed Xact for durability

types of failure:
- Xact failure (abort)
- system crash (loss of volatile memory)
- media failure (data lost/corrupted)

↳ recovery manager

process $\text{Commit}(T)$, $\text{Abort}(T)$, and Restart
- on restart, abort all active Xacts and install updates of all committed Xacts that were not installed
- desirable properties: add little overhead to normal processing of Xacts and recover quickly

interactions with buffer manager: default buffer manager behvior is to write dirty page to disk when dirty page is replaced; may be altered
- steal: dirty page updated by Xact can be written to disk before Xact commits
- force: all dirty pages updated by Xact must be written to disk when Xact commits

	force	**no force**
steal	undo & no redo	undo & redo
no steal	no undo & no redo	no undo & redo

↳ log-based database recovery

log: history of actions executed by DBMS; records for write, commit, abort, etc
- stored as a sequential file of records in stable storage with multiple copies
- each log record identified with Log Sequence Number (LSN) (akin to timestamp)

↳ ARIES recovery algorithm

works with steal, no-force approach and assumes strict 2PL for concurrency control

structures required: updated during normal processing
- log file (stable storage)
- transaction table (TT) (volatile memory)
  - each entry → active Xact
  - (XactID, lastLSN, Xact status)
  - lastLSN → most recent log record for Xact
  - Xact status → C (committed), U (not committed)
- dirty page table (DPT) (volatile memory)
  - each entry → dirty page in buffer pool
  - (pageID, recLSN)
  - recLSN → earliest log record that dirties page

log record fields/types: all → (type, XactID, prevLSN)
- update → (pageID, offset, length, before-image, after-image)
- compensation log record (CLR) → (pageID, undoNextLSN, action taken to undo)
  - undoNextLSN → LSN of next log record to be undone (i.e. prevLSN of update record)
- commit → all log records force-written to stable storage
- abort → undo initiated with this Xact
- end → confirmation that a commit/abort is completed
- checkpoint → indicates when to start recovery

protocols:
1. write-ahead logging (WAL) → don't flush uncommitted update to database until log record containing before-image is flushed to log
   - pageLSN → latest log that updated page
   - ensure all log records up to log record corresponding to page's pageLSN are flushed to disk
2. force-at-commit → don't commit Xact until after-images of all its updated records are in stable storage
   - write commit log record and flush all log records to disk

normal operations:
- TT: create entry for T with status U; each new log record for T updates lastLSN field; T commits → status to C; end log record → remove T entry
- DPT: create entry for P if not already in table; remove entry when P flushed to disk

aborts: for each log record of Xact in reverse order, restore log record's before-image via lastLSN and prevLSN traversal
- each undo is logged as a CLR log → ensure action is not repeated during repeated undos

commits: write a commit log record in stable storage

⇝ restarts

analysis phase: reconstruct DPT and TT
- initialize DPT and TT & scan log in forward direction r, performing normal operation

redo phase: reconstruct DB to state at time of crash
- redoLSN = smallest recLSN of DPT
- r is log record with LSN = redoLSN (starting position)
- if r is a redoable action → fetch P and if P pageLSN < r LSN, reapply logged action in r and update P pageLSN to r LSN (i.e. P most recent log record updating it was before r)
- after redo phase, create end log records for all Xacts with status C in TT and delete

undo phase: undo actions that Xact didn't commit
- abort active Xacts at time of crash
- L is the set of lastLSNs with status U in TT

- **update(lsn):**
  - lsn != `NULL` → add to L
  - else → create end record for T and remove T
- repeat till L is empty
  - r = record with largest lastLSN in L
  - r is abort → update(r prevLSN)
  - r is CLR → update(r undoNextLSN)
  - r is update →
    - create CLR r_2 → undoNextLSN = r prevLSN
    - update T lastLSN entry to r_2 LSN
    - undo logged action on P
    - update P pageLSN = r_2 LSN
    - update(r prevLSN)

⇝ checkpointing

performed periodically to speed up restart recovery, adding some overhead to normal processing

simple: during analysis phase, begin from latest checkpoint log record with checkpoint TT and empty DPT
1. stop accepting new operations
2. wait till all active operations finish
3. flush all dirty pages in buffer
4. write checkpoint log record with TT
5. resume accepting operations

fuzzy checkpointing: let DPT' and TT' be snapshots
- assumption: no log records between begin and end
1. write begin_checkpoint log record
2. write end_checkpoint log record with DPT' and TT'
3. write special master record with LSN of begin_checkpoint log record to a known place in stable storage

changes to phases:
- analysis: start with begin_checkpoint log record from master record, using DPT' and TT' from end_checkpoint log record and proceed
- redo: exploit information in DPT to avoid retrieving P
  - optimization condition: (P not in DPT) or (P recLSN in DPT > r LSN)
  - optimization holds → update r already applied to P so update can be ignored
  - otherwise if optimization does not hold and r is redoable
    - attempt to reapply action
    - otherwise, update P recLSN = P pageLSN + 1
      - hopefully next optimization holds so skip fetching
- undo: no change