CS3223 Cheatsheet AY24/25 —— @JasonYapzx

- **Covering Index**
  - An index I on R is a covering index for a query Q on R if all R's attributes referenced in Q are part of the key or include columns of I
  - No need to do another RID lookup for other attributes relevant to the query
- **Covered Conjuncts**
  - Given a predicate p on R and index I on R with key K, a conjunct C in p is a convered conjunct if all attributes in C appear in K/ any include columns of I
- **Primary Conjuncts**
  - an index I matches only a subset of the conjuncts in a selection predicate p
  - the subset of conjuncts in p that I matches are called primary conjuncts
  - Primary conjuncts $\subseteq$ Covered conjuncts

Cost of B+ Tree index scan $= N_{internal} + N_{leaf} + N_{lookup}$

- $N_{internal}$ denotes the number of internal nodes accessed in index
- N_{leaf} denote the number of leaf index nodes accessed in index
- N_{lookup} denote number of pages accessed to retrieve matching data records

Cost of hash index scan $= N_{dir} + N_{bucket} + N_{lookup}$

- N_{dir} denote no. of index's directory pages accessed
  - $N_{dir} = 1$ if the index is an extendible hash index; and 0 otherwise
- N_{bucket} denote no. of index's primary/overflow pages accessed
- N_{lookup} be no. of pages accessed to retrieve the matching data records

5. Projection and Join

Projection: $\pi_{A_1}, \cdots, \pi_{A_m}(R)$

- $\pi_L(R)$ projects columns given by list L from relation R ($\pi^*_L(R)$ keeps dupes)
  1. Remove unwanted attributes AND Eliminate any duplicates tuples produced
  2. Done by projection based on sorting or hashing

Sort based approach

Consider $\pi_L(R)$ where L denote some sequence of attributes of R

- Non-opt: run size $= B$, $N_0 = \lceil |\pi^*_L(R)|/B \rceil$
- Opt: run size $= B - 1$, $N_0 = \lceil |\pi^*_L(R)|/(B-1) \rceil$
- Cost (both): $|R| + 2|\pi^*_L(R)|(\lceil \log_{B-1}(N_0) \rceil + 1)$

Hash-based

- Consists of two phases:
  1. Partitioning phase: partitions R into $R_1, R_2, \ldots, R_m$. R is red in 1 page at a time into input buffer, projected out required attributes
     - $\pi^*_L(R_i) \cap \pi^*_L(R_j) = \emptyset$ for each pair R_i & R_j, $i \neq j$
  2. Duplicate elimination phase: eliminates duplicates from each $\pi^*_L(R_i)$. Init an in-memory hash table to store and output unique tuples
- $\pi_L(R) =$ duplicate-free union of $\pi_L(R_1), \pi_L(R_2), \ldots, \pi_L(R_{B-1})$
- Partition R by hashing on $\pi_L(t)$ for each tuple $t \in R$

Hash-based Approach: Cost Analysis

- Approach effective if no. of allocated memory pages B is large relative to $|R|$
  - Assume that h distributes tuples in R uniformly, each R_i has $\frac{|\pi^*_L(R)|}{B-1}$ pages
  - Size of hash table for each $R_i = \frac{|\pi^*_L(R)|}{B-1} \times f$, $f =$ fudge factor, $f > 1$
  - $\therefore$ to avoid partn overflow, $B > \frac{|\pi^*_L(R)|}{B-1} \times f \approx B > \sqrt{f \times |\pi^*_L(R)|}$
- Analysis: assume there is no partition overflow
  - Cost of partitioning phase: $|R| + |\pi^*_L(R)|$, Cost of duplicate elimination phase: $|\pi^*_L(R)|$ $\therefore$Total I/O cost $= |R| + 2|\pi^*_L(R)|$

Sort vs Hash

- Hash-based (assume $B > \sqrt{f \times |\pi^*_L(R)|}$; i.e., no partition overflow)
  - Cost $= \underbrace{|R| + |\pi^*_L(R)|}_{\text{partitioning phase}} + \underbrace{|\pi^*_L(R)|}_{\text{duplicate elimination phase}}$
- Sort-based
  - Output is sorted, Good if there are many duplicates or if distribution of hashed values is non-uniform
  - If $B > \sqrt{|\pi^*_L(R)|}$,
    - ★ Number of initial sorted runs $N_0 = \lceil \frac{|R|}{B} \rceil \approx \sqrt{|\pi^*_L(R)|}$
    - ★ Number of merging passes $= log_{B-1}(N_0) \approx 1$
    - ★ Sort-based approach requires 2 passes for sorting
    - ★ Cost $= \underbrace{|R| + |\pi^*_L(R)|}_{\text{pass 0}} + \underbrace{|\pi^*_L(R)|}_{\text{pass 1}}$
  - ★ Both hash-based & sort-based methods have same I/O cost

Index-based Projection: If index I on R covers $\pi_L(R)$, scan I directly. If I is a B$^+$ Tree and L is a prefix of its key K, entries are sorted $\rightarrow$ scan I to deduplicate.

Join: $R \bowtie_\Theta S$

- Things to consider when choosing a join algorithm
- Equality predicates e.g. $R.A_i = S.B_j$/Inequality predicates e.g. $R.A_i < S.B_j$
- Size of join operands, Allocated memory pages, Available access methods
- Given join $R \bowtie_\Theta S$, left R is outer, right operand S is inner relation

Iteration-based — Block nested loop

- **Tuple-based Nested Loop Joins:** basically 2 for loops for each tuple

```
for each page P_r in R do
  for each tuple r in P_r do
    for each page P_S in S do
      for each tuple s in P_S do
        if (r matches s) then
            output (r,s) to result
```

- I/O Cost Analysis for: $|R| + ||R|| \times |S|$

- **Page-based Nested Loop Joins:** Swap loops 2 and 3

```
for each page P_r in R do
  for each page P_S in S do
    for each tuple r in P_r do
      for each tuple s in P_S do
        if (r matches s) then
          output (r,s) to result
```

- I/O Cost Analysis: $|R| + |R| \times |S|$, pick smaller table as outer
- **Main Difference:** Tuple-based joins compare individual tuples, leading to higher I/O due to repeated scans of the inner relation. Page-based joins process entire pages, reducing redundant reads and improving I/O efficiency.

Block Nested Loop Join

- Exploit allocated memory pages better to minimize I/O Cost
- Assume $|R| \leq |S|$ so choose R as outer and S as inner
- **Memory pages allocation:** Allocate one page for S one page for output and remaining pages for R

```
while (scan of R is not done) do
  read next (B-2) pages of R into buffer
    for each page P_S of S do
      read P_S into buffer
        for each tuple r of R in buffer, and each tuple s in P_S do
          if (r matches s) then
            output (r,s) to result
```

- I/O Cost Analysis for $R \bowtie S$: $|R| + (\lceil \frac{|R|}{B-2} \rceil \times |S|)$

Index Nested Loop Join

- **Precondition:** there is an *index on the join attributes* of inner relation S
- **Idea:** for each tuple $r \in R$ use r to probe S's index to find matching tuples
- **Analysis:**
  - Let $R.A_i = S.B_j$ be join condition
  - Assume S stored in Heap File
    - I/O Cost: $|R| + ||R|| \times (N_{internal} + N_{leaf} + N_{lookup})$ (R-tuple $\bowtie$ S)
    - N_{leaf} and N_{lookup} estimated for $\sigma_{B_j = c}(S)$ where c is some constant

Sort Merge Join: total cost = cost to sort $|R|$ + cost to sort $|S|$ + cost to merge

- Sort R and S on join attr $\rightarrow$ merge partitions with matching keys
- Pointers scan both; advance the one pointing to smaller tuple
- Rewind S if multiple R tuples match
- Cost: $2|R|(\log_m N_R + 1) + 2|S|(\log_m N_S + 1) +$ merge
- Merge: best $= |R| + |S|$ (if each S part scanned at most once), worst $= |R| + ||R|| \times |S|$ (each R-tuple requires scanning entire S)
- **If each S Partition scanned at most once during merging** $= 2|R|(\lceil \log_m N_r \rceil + 1) + 2|S|(\lceil \log_m N_s \rceil + 1) + |R| + |S|$
- **If each S partition is scanned more than once during merging** $= 2|R|(\lceil \log_m N_r \rceil + 1) + 2|S|(\lceil \log_m N_s \rceil + 1) + |R| + |R| * |S|$

Optimized Sort Merge Join

- **Conventional Sort-Merge Join:** $N_R = \lceil \frac{|R|}{B} \rceil$
  - **Sort R:** create sorted runs of R; merge sorted runs of R
  - **Sort S:** create sorted runs of S; merge sorted runs of S
  - **Join R and S:** merge sorted R & sorted S
- **Optimized Sort-Merge Join**
  - Create sorted runs of R; merge sorted runs of R partially
  - Create sorted runs of S; merge sorted runs of S partially
  - Merge remaining sorted runs of R & S and join them at the same time
- **Analysis**
  - Assume $|R| \leq |S|$, If $B > \sqrt{2|S|}$
    - Number of initial sorted runs of $S < \sqrt{\frac{|S|}{2}}$
    - Total number of initial sorted runs of R and $S < \sqrt{2|S|}$
    - One pass sufficient to merge and join initial sorted runs
    - I/O Costs $= 2 \times (|R| + |S|) + (|R| + |S|) = 3 \times (|R| + |S|)$

Hash Join, $S_{\bowtie S.B = R.A} R$

- **Idea**
  - Partition R and S into k partitions using some hash function h
    - $R = R_1 \cup R_2 \cdots \cup R_k, t \in R_i$ iff $h(t.A) = i$
    - $S = S_1 \cup S_2 \cdots \cup S_k, t \in S_i$ iff $h(t.B) = i$
    - $\pi_A(R_i) \cap \pi_B(S_i) = \emptyset$ for each $R_i \& S_j, i \neq j$
  - Join corresponding pairs of partition: $S \bowtie R = (S_1 \bowtie R) \cup \cdots (S_k \bowtie R)$

Grace Hash Join

- Partition R into $R_1 \cdots R_k$, S into $S_1 \cdots S_k$, Probing phase: probes R_i w/ S_i
  - Read R_i to build hash table — build relation
  - Read S_i to probe hash table — probe relation
- **Partitioning (building) Phase**

```
init hash table T with k buckets
for each tuple in R do
  insert r into bucket h(r.A) of T
write each bucket R_i of T to disk
init hash table T with k buckets
for each tuple s in S do
  insert s into buckets h(s.B) of T
write each bucket S_i of T to disk
```

- **Probing (matching) phase**

```
for i = 1 to k do
  init hash table T
  for each tuple r in partition R_i do
    insert r into bucket h'(r.A) of T
  for each tuple s in partition S_i do
    for each tuple r in bucket h'(s.B) of ↩
          T do
      if r and s matches then output (r,s↩
          )
```

- **Analysis**
  - **To minimize size of each partition of R_i,**
    - Let B be the number of memory pages allocated for the join, $k = B - 1$
  - **Assuming uniform hashing distribution,**
    - size of each partition R_i is $\frac{|R|}{B-1}$
    - size of hash table for R_i is $\frac{f \times |R|}{B-1}$, where $f > 1$ is a fudge factor
    - During probing phase, $B > \frac{f \times |R|}{B-1} + 2$ (with one memory page for S_i's input buffer & one memory page for output buffer)
    - Approximately, $B > \sqrt{f \times |R|}$
  - **Partition overflow problem**
    - Hash table for R_i does not fit in memory
    - Solution: recursively apply partitioning to overflow partitions
  - **I/O cost = Cost of partitioning phase + Cost of probing phase**
    - I/O Cost $= 2 \times (|R| + |S|) + (|R| + |S|) = 3 \times (|R| + |S|)$ if there's no partition overflow problem $\Rightarrow$ we use this condition to check if build rs too large to fit into memory: $\lceil \frac{|R|}{B-1} \rceil \leq B - 2$
    - otherwise, partition again, I/O Cost $= (2L + 1)(|R| + |S|)$, where $L = \lceil \log_{B-1}(\frac{|S|}{B-2}) \rceil$
  - **Join Condition Support**
    - **Multiple equality-join conditions** (e.g., $(R.A = S.A)$ and $(R.B = S.B)$):
      - Supported by: Index Nested Loop Join (with indexes), Sort-Merge Join (sort on combined keys), and other standard join algorithms.
    - **Inequality-join conditions** (e.g., $(R.A < S.A)$):
      - Supported by: Index Nested Loop Join (requires B^+-tree index).
      - Not supported by: Sort-Merge Join, Hash-based Joins.

6. Query Evaluation and Optimization

Query Evaluation

Materialized evaluation:

- Operator only evaluated when each of its operands has been completely evaluated/materialized $\rightarrow$ intermediate results written to disk
- `temp1` = table scan Movies.r > 8, `temp2` = NLJ of `temp1 AND Acts ON title`
- result = Hash-based projection of temp2 on actor, director

Pipelined evaluation:

- Output produced by operator passed to parent operator directly
- Execution of operators are interleaved, top-down demand driven approach
- An operator O is a blocking operator if O may not be able to produce any output until it has received all the input tuples from its child operator(s)
  - e.g. external merge sort, sort-merge join, Grace hash join
- Iterator interface for operator evaluation
  - `open`: initializes state of iterator: allocates resources for operation, initializes operator's arguments (e.g., selection conditions)
  - `getNext`: generates next output tuple, `null` if all output tuples generated
  - `close`: deallocates state information

Query Plans

- A query generally has many equivalent logical query plans
- **Physical Query Plans:**
  - Each logical plan can be implemented by many physical query plans
  - Example: 2 possible physical query plans for $(R \bowtie_A S) \bowtie_B T$
- Three key components: Search space, Plan enumeration, Cost model

Relational Algebra Equivalence Rules

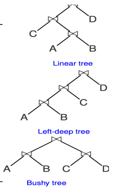
- **attributes(R)** = Set of attributes in schema of relation R
- **attributes(p)** = Set of attributes in predicate p

1. **Commutativity of binary operators**
   - $R \times S \equiv S \times R$ OR $R \bowtie S \equiv S \bowtie R$
2. **Associativity of binary operators**
   - $(R \times S) \times T \equiv R \times (S \times T)$ && $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$
3. **Idempotence of unary operators**
   - $\pi_{L'}(\pi_L(R)) \equiv \pi_{L'}(R)$ if $L' \subseteq L \subseteq$ attributes(R)
   - $\sigma_{p_1}(\sigma_{p_2}(R)) \equiv \sigma_{p_1 \wedge p_2}(R)$
4. **Commutating selection with projection**
   - $\pi_L(\sigma_p(R)) \equiv \pi_L(\sigma_p(\pi_{L \cup \text{attributes}(p)}(R)))$
5. **Commutating selection with binary operators**
   - $\sigma_p(R \times S) \equiv \sigma_p(R) \times S$ if attributes$(p) \subseteq$ attributes(R)
   - $\sigma_p(R \bowtie_{p'} S) \equiv \sigma_p(R) \bowtie_{p'} S$ if attributes$(p) \subseteq$ attributes(R)
   - $\sigma_p(R \cup S) \equiv \sigma_p(R) \cup \sigma_p(S)$
6. **Commutating projection with binary operators:** Let $L = L_R \cup L_S$, where $L_R \subseteq$ attributes(R) and $L_S \subseteq$ attributes(S)
   - $\pi_L(R \times S) \equiv \pi_{L_R}(R) \times \pi_{L_S}(S)$
   - $\pi_L(R \bowtie_p S) \equiv \pi_{L_R}(R) \bowtie_p \pi_{L_S}(S)$ if attributes$(p) \cap$ attributes$(R) \subseteq L_R$ and attributes$(p) \cap$ attributes$(S) \subseteq L_S$
   - $\pi_L(R \cup S) \equiv \pi_L(R) \cup \pi_L(S)$

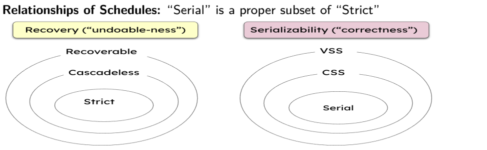
Types of Query Plan Trees

- A query plan is **linear** if at least one operand for each join operation is a base relation; otherwise, the plan is **bushy**.
- A linear query plan is **left-deep** if every right join operand is a base relation.
- A linear query plan is **right-deep** if every left join operand is a base relation.
- Consider the query $A \bowtie B \bowtie C \bowtie D$.
- Dynamic Programming forumation

```
Input: A SPJ query q on relations R_1, R_2, ···, R_n
Output: An optimal query plan for q
for i = 1 to n do
    optPlan({R_i}) = best access plan for R_i
for i = 2 to n do
    for each S ⊆ {R_1,··· ,R_n}, |S| = i do
    bestPlan = dummy plan with cost(bestPlan) = ∞
    for each S_j, S_k, |S_j| ∈ [1, i), S = S_j ∪ S_k do
        p = best way to join optPlan(S_j) and optPlan(S_k)
        if (cost(p) ≤ cost(bestplan)) then
        bestPlan = p
    optPlan(S) = bestPlan
return optPlan({R_1,··· ,R_n})
```



Relationships of Schedules: "Serial" is a proper subset of "Strict"



System R Optimizer

- Uses heuristics to prune search space:
  - Enumerates only left-deep query plans
  - Avoids cross-product query plans
  - Considers early σ (selections) & π (projections)
- Uses enhanced dynamic programming approach that considers sort order of query plan's output
  - Maintains optPlan(S_i, o_i) instead of optPlan(S_i)
  - o_i captures the sort order of output produced by query plan w.r.t. S_i
  - $o_i = \begin{cases} \text{null} & \text{if output is unordered} \\ \langle A_1, \ldots, A_k \rangle & \text{sequence of attributes} \end{cases}$
  - optPlan(S_i, o_i) = cheapest query plan for relations S_i with output ordered by o_i if $o_i \neq$ null

Cost Estimation of Query Plans

1. What is eval cost of each operation? size of input, avail buffer pages/index etc.
2. What is the output size of each operation?
3. How to estimate?: with the following assumptions:
   (a) Uniformity: uniform distribution of attribute values
   (b) Independence: independent distribution of values in different attributes
   (c) Inclusion: 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

- **Reduction factor** (a.k.a selectivity factor) of a term t_i (denoted by $rf(t_i)$) is the fraction of tuples in e that satisfy t_i; i.e., $rf(t_i) = \frac{||\sigma_{t_i}(e)||}{||e||}$
- Assuming terms in p are statistically independent: $||q|| \approx ||e|| \times \prod_{i=1}^{n} rf(t_i)$

Join Selectivity

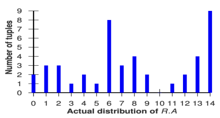
- Join selectivity factor = selectivity factor for join predicates
  - Consider query Q: SELECT * FROM R JOIN S ON R.A = S.B
  - $rf(R.A = S.B) = \frac{||R \bowtie_{R.A=S.B} S||}{||R|| \times ||S||}$
  - Inclusion assumption: Consider $R \bowtie_{R.A=S.B} S$
    - If $||\pi_A(R)|| \leq ||\pi_B(S)||$, then $\pi_A(R) \subseteq \pi_B(S)$
- **Join selectivity estimation**:
  - Assume $||\pi_A(R)|| \leq ||\pi_B(S)||$
  - By inclusion assumption, every R-tuple joins with some S-tuple
  - By uniformity assumption, there are $\frac{||S||}{||\pi_B(S)||}$ S-tuples corresponding to each S.B value
  - Therefore, each R-tuple joins with $\frac{||S||}{||\pi_B(S)||}$ S-tuples
  - Thus, $||Q|| \approx ||R|| \times \frac{||S||}{||\pi_B(S)||}$
- $rf(R.A = S.B) \approx \frac{1}{\max\{||\pi_A(R)||, ||\pi_B(S)||\}}$

Estimation using Histograms

- *histogram* = statistical info maintained by DBMS to estimate data distribution
  - Partition attribute's domain into sub-ranges called buckets
  - Assume value distribution within each bucket is uniform
- Types of histograms:
  - **Equiwidth histograms**: Each bucket has (almost) equal number of values
  - **Equidepth histograms**: Each bucket has (almost) equal number of tuples, sub-ranges of adjacent buckets might overlap

Improved Histogram Estimation with MCV — Most Comon Values

- Separately keep track of the frequencies of the top-k most common values and exclude MCV from histogram's buckets



MCV (k=2)

Value	No. of Tuples
6	8
14	9

Equidepth Histogram (with 3 buckets)

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

7. Transaction Management

1. Atomicity: Either all or none of the actions in transaction happen
2. Consistency: If each txn & DB starts consistent, DB ends up consistent
3. Isolation: Execution of one transaction is isolated from other transactions
4. Durability: If a transaction commits, its effects persist

concurrency control manager $\rightarrow$ isolation, recovery manager $\rightarrow$ atomicity + durability

Transactions — Each Xact must end with either a commit or an abort action

- A transaction (Xact) T_i can be viewed as a sequence of actions:
  - $R_i(O) = T_i$ reads an object O
    - $R_i(O, v) = T_i$ reads an object O whose value is v
  - $W_i(O) = T_i$ writes an object O
    - $W_i(O, v) = T_i$ writes an object O with a value of v
  - Commit$_i$ (or C_i) $= T_i$ terminates successfully
  - Abort$_i$ (or A_i) $= T_i$ terminates unsuccessfully
- A Xact is an active Xact if it is still in progress (i.e., has not yet terminated); otherwise, it is a completed Xact (i.e., committed or aborted)

Transactions Schedules

- Schedule = a list of actions from a set of Xacts, where the order of the actions within each Xact is preserved
- **Example:** Consider two Xacts T_1 and T_2:
- A serial schedule is a schedule where the actions of Xacts are not interleaved

Read From & Final Write — Dirty Read & Dirty Write

- We say that T_j reads O from T_i in a schedule S if the last write action on O before $R_j(O)$ in S is $W_i(O)$
- We say that T_j reads from 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 action on O in S is $W_i(O)$
- dirty read if the read value was produced by an active transaction
- dirty write if the overwritten value of O was produced by an active transaction

Correctness of Interleaved Xact Executions: An interleaved Xact execution schedule is correct if it is "equivalent" to some serial schedule over the same set of Xacts

View Serializable Schedule: if view eqv to some serial schedule over same set of txns

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

Testing for View Serializability — Check WR and Final Write

Theorem 1: S is VSS iff there exists some VSG(S) corresponding to S that is acyclic

- A view serializability graph for a schedule S (denoted by $VSG(S)$) is a directed graph $VSG(S) = (V, E)$ such that the nodes V represent Xacts & the edges E represent precedence relations among Xacts:
  1. If T_i reads from T_j, then $(T_j, T_i) \in E$
  2. If both T_i & T_j update the same object O & T_j performs the final write on O, then $(T_i, T_j) \in E$
  3. If T_i read some object O from T_k & T_j update object O, then either $(T_i, T_k) \in E$ or $(T_j, T_i) \in E$
- Due to (c), there could be multiple VSG(S) corresponding to S
- 2 actions on same O conflict if ≥ 1 is write action OR actions are from diff Xacts

Anomalies with Interleaved Xact Executions

- Anomalies can arise due to conflicting actions:
  1. Dirty read problem (due to WR conflicts)
     - T_2 reads an object that has been modified by T_1 and T_1 has not yet committed (i.e., T_2's read is a dirty read)
     - T_2 could see an inconsistent DB state!
  2. Unrepeatable read problem (due to RW conflicts)
     - T_2 updates an object that T_1 has previously read and T_2 commits while T_1 is still in progress
     - T_1 could get a different value if it reads the object again!
  3. Lost update problem (due to WW conflicts)
     - T_2 overwrites the value of an object that has been modified by T_1 while T_1 is still in progress (i.e., T_2's write is a dirty write)
     - T_1's update is lost!

Conflict Serializable Schedules

- Two schedules S & S' (over the same set of Xacts) are said to be conflict equivalent (denoted by $S \equiv_c S'$) if they order every pair of conflicting actions of two committed Xacts in the same way
- A schedule is a conflict serializable schedule (CSS) if it is conflict equivalent to a serial schedule over the same set of Xacts

Testing for Conflict Serializability — Check WR, RW, WW

- A conflict serializability graph (check $R \rightarrow W$ and $W \rightarrow W$ edges) for a schedule S (denoted by $CSG(S)$) is a directed graph $CSG(S) = (V, E)$ s.t.
  - V contains a node for each Xact in S
  - E contains (T_i, T_j) if action in T_i precedes & conflicts w/ 1 of T_j's actions
- **Theorem 2: A schedule is CSS iff its conflict serializability graph is *acyclic***
- **Theorem 3: A schedule that is CSS is also view serializable**

Blind Writes

- W on object O by T_i is called a blind write if T_i did not read O prior to the write
- **Theorem 4: If S is VSS and S has no blind writes, then S is also CSS**

Cascading Aborts

- For correctness, if T_i has read from T_j, then T_i must abort if T_j aborts
  - Consider the schedule: $W_1(x)$, $R_2(x)$, $W_2(y)$, Abort$_1$
  - T_2 must abort since it read from T_1 which aborted
- We say that T_1's abort is cascaded to T_2
- Recursive aborting process is known as cascading abort

Recoverable Schedules

- A schedule S is said to be a recoverable schedule if for every Xact T that commits in S, T must commit after T' if T reads from T'
- This is a non-recoverable schedule because T_2 commits before T_1 commits even though T_2 reads from T_1

T_1	T_2
$W_1(x)$	
	$R_2(x)$
	$W_2(y)$
	Commit$_2$

Cascadeless Schedules

- **Undesirable**: cost of bookkeeping to identify & performance penalty incurred
- **Theorem 5: A cascadeless schedule is also a recoverable schedule**
- To avoid cascading aborts (or to be cascadeless), DBMS must permit reads only from committed Xacts
- A schedule S is a cascadeless schedule if there is no dirty read in S

```
for every Read Rj(X)
  if there exists a Wi(X) before it
    if Commit(Ti) is after Rj
      return NOT cascadeless
return cascadeless
```

Recovery using Before-Images

- An efficient approach to undo the actions of aborted Xacts is to restore before-images for writes
- We use $W_i(x, v)$ to denote that T_i updates the value of object x to v
- **Example:** Consider the following schedule S:
  - $W_1(A, 100)$, $W_2(A, 200)$, Abort$_2$
  - Assume that the initial value of A is 50
  - Before performing $W_1(A, 100)$, its before-image "$A = 50$" is logged
  - Before performing $W_2(A, 200)$, its before-image "$A = 100$" is logged
  - To recover from Abort$_2$, $W_2(A, 200)$ is undone by restoring the before-image of A (i.e., the value of A is restored to 100)
- However, before-image recovery doesn't always work!
- $W_1(A, 100)$, $W_2(A, 200)$, $Abort_1$
- Here, undoing $W_1(A, 100)$ by restoring A to its before-image of 50 is incorrect!

Strict Schedules

- A schedule S is a strict schedule if there is no dirty read and no dirty write in S

```
for every Read Rj(X) or Write Wj(X)
  if there exists a Wi(X) before it
    if Commit(Ti) is after Rj or Wj
      return NOT strict
return strict
```

- **Example:**
  - S: $W_1(A)$, $W_2(A)$, Abort$_2$, S': $W_1(A)$, $W_2(A)$, Abort$_1$
  - Both S and S' are not strict schedules
- **Performance Tradeoff**: recovery (w/ before-images) more efficient but concurrent executions become more restrictive
- **Theorem 6: A strict schedule is also a cascadeless schedule**

8. Concurrency Control

Lock-Based Concurrency Control

- Each Xact needs to request for an *appropriate lock* before it can access the object
- **Locking modes:**
  - Shared (S) locks for reading objects OR Exclusive (X) locks for writing objects
- If T's lock request is not granted on O, T becomes blocked; its execution is suspended & T is added to O's request queue
- When a lock is released on O, the lock manager checks the request of the first Xact T in the request queue for O. If T's request can be granted, T acquires its lock on O and resumes execution after its removal from the queue
- When a Xact commits/aborts, all its locks are released & T is removed from any request queue it is in

Two Phase Locking (2PL) Protocol

- **2PL Protocol:**
  1. To read an object O, a Xact must hold a S-lock or X-lock on O
  2. To write to an object O, a Xact must hold a X-lock on O
  3. Once a Xact releases a lock, the Xact can't request any more locks
- Xacts using 2PL can be characterized into two phases:
  - Growing: before releasing 1st lock — Shrinking: after releasing 1st lock
- **Theorem 1: 2PL schedules are conflict serializable**
- **Strict 2PL Protocol:**
  1. To read an object O, a Xact must hold a S-lock or X-lock on O
  2. To write to an object O, a Xact must hold a X-lock on O
  3. A Xact must hold on to locks until Xact commits or aborts
- **Theorem 2: Strict 2PL schedules are strict & conflict serializable**

Deadlock: cycle of Xacts *waiting for locks to be released* by each other

- **How to Detect Deadlocks?**
  - **Waits-for graph (WFG)**
    - Nodes represent active Xacts
    - Add an edge $T_i \to T_j$ if T_i is waiting for T_j to release a lock
    - lock request, updates edges when it grants a lock request
  - Deadlock detected if WFG has cycle, break by aborting a Xact in cycle
  - **Lock manager**
    - adds an edge when it queues a
  - Alternative to WFG: timeout mechanism
- **How to Prevent Deadlocks?**
  - Assume older Xacts have higher priority than younger Xacts
    - Each Xact is assigned a timestamp when it starts, older txn has smaller
  - Suppose T_i requests for a lock that conflicts with a lock held by T_j
    - Wait-die policy:
      - non-preemptive: only a Xact requesting for a lock can get aborted
      - a younger Xact may get repeatedly aborted
      - a Xact that has all the locks it needs is never aborted
    - Wound-wait policy: preemptive, higher-priority never wait for lower-priority
    - To avoid starvation, a restarted Xact must use its *original timestamp!*

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 Conversions

- Increase concurrency by allowing lock conversions
- Two types of lock conversions:
  - $UG_i(A)$: T_i requests to upgrade its S-lock on object A to X-lock
  - $DG_i(A)$: T_i requests to downgrade its X-lock on object A to S-lock

2PL with Lock Upgrades

- **2PL with lock upgrades**
  - To perform $R_i(O)$, T_i must be holding a S/X lock on O. If not, T_i requests for a S on O (i.e., $S_i(O)$)
  - To perform $W_i(O)$, T_i must be holding an exclusive lock on O. If not,
    - If T_i is *holding a shared lock* on O, T_i requests for a $UG_i(O)$
    - If T_i is *not holding any lock* on O, T_i requests for a $X_i(O)$
- $UG_i(O)$ is allowed only in the **growing phase** (i.e., T_i has not released any lock)
- $UG_i(O)$ is blocked if another Xact is holding a shared lock on O
- If $UG_i(O)$ is granted, $S_i(O)$ becomes $X_i(O)$

Performance of Locking

- Resolve Xact conflicts by using blocking and aborting mechanisms
- Blocking causes delays in other waiting Xacts
- Aborting and restarting a Xact wastes work done by Xact
- How to increase system throughput?
  1. Reduce the locking granularity — Reduce the time a lock is held
  2. Reduce hot spots — a DB object that is *frequently accessed and modified*

Phantom read problem

A transaction **re-executes a query** and sees new rows (phantoms) that were not there before

- Phantom problem can be prevented by predicate locking
  - Xact 1 is granted a shared lock on the predicate $balance > 1000$
  - Xact 2's request for an exclusive lock on predicate $balance = 3000$ is blocked
  - e.g., T1 reads all accounts with $balance > 1000$; T2 inserts account with $balance = 3000$ $\to$ T1 sees a phantom on re-read
- In practice, phantom problem is prevented via index locking

ANSI SQL Isolation Levels

- **Isolation Level Table:**

Isolation Level	Dirty	Unrepeatable	Phantom
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

- SQL's SET TRANSACTION ISOLATION LEVEL command:

```
BEGIN TRANSACTION;
SET TRANSACTION ISOLATION LEVEL
  { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE };
......
COMMIT;
```

- In many DBMSs, the default isolation level is READ COMMITTED
- **Lock Duration Table:**

Deg	Isolation level	Write lock	Read lock	Predicate lock
0	Read Uncommitted	long	none	none
1	Read Committed	long	short	none
2	Repeatable Read	long	long	none
3	Serializable	long	long	yes

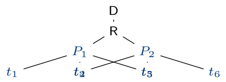
- Short duration: could be released **after end of op.** before Xact commits/aborts
- Long: lock acq for an op. is held **until** Xact commits/aborts

Locking Granularity

- **What to lock?** database $\to$ relation $\to$ page $\to$ tuple
- Locking granularity = size of data items being locked
  - highest (coarsest) granularity = database — lowest (finest) granularity = tuple

Multi-granular Locking

- Allow multi-granular lock instead of fixed granule locking
- 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
- **Example:** Consider database D containing relation R consisting of pages P_1 and P_2 each with 3 tuples



- If T is holding a S on P_1, then T is implicitly holding a S on t_1, t_2, t_3
- If T is holding a X on R, then T is implicitly holding a X on P_1, P_2, t_1, t_2, t_3, t_4, t_5, & t_6

Multi-granular Locking Protocol

- **Idea:** Use a new intention lock (I-lock) mode
- **Protocol:** Before acquiring S-lock/X-lock on a data granule G, need to acquire I-locks on granules coarser than G in a top-down manner
- **Example:** Xact T wants to request X-lock on tuple t_4
  - Must first acquire I locks on D, then R, then P_2 before acquiring X-lock on t_4
- **Problem:** Limited concurrency with lock modes I, S, and X
- **Example:** Suppose T_1 has a S-lock on t_4 $\implies$ T_1 has I-locks on D, R & P_2
  - If T_2 wants to read P_2:
    - its I req on D & R will be granted, but its S request on P_2 will be blocked

Multi-granular Locking Protocol — Refined

- Refine *intention lock* idea with IS & IX lock modes
  - intention shared (IS): intent to set S-locks at finer granularity
  - intention exclusive (IX): intent to set X-locks at finer granularity
- **Lock compatibility matrix:**

Lock Requested \ LockHeld	-	IS	IX	S	X
IS	✓	✓	✓	✓	×
IX	✓	✓	✓	×	×
S	✓	✓	×	✓	×
X	✓	×	×	×	×

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

9. Multiversion Concurrency Control

- $W_i(O)$ creates a new ver. of object O — $R_i(O)$ reads appropriate ver. of O
- **Advantages:**
  - Read-only Xacts are not blocked by update Xacts
  - Update Xacts not blocked by read-only Xacts, Read-only Xacts never aborted
- **Multiversion Schedules:**
  - If there are multiple versions of an obj x, a *read* on x could return *any version*
  - $\therefore$ interleaved exec could correspond to diff MVS depending on MVCC protocol
- **Multiversion View Equivalence:** S and S', over same set of transactions, are defined to be multiversion view equivalent ($S \equiv_{mv} S'$) if they have the same set of read-from relationships ▷ i.e., $R_i(x_j)$ occurs in S iff $R_i(x_j)$ occurs in S'
- **Monoversion Schedules:** A multiversion schedule S is called a monoversion schedule if each read action in S returns the *most recently created object version*
- **Serial Monoversion Schedules:** Monoversion schedule that is also a serial

Multiversion View Serializability

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

Testing for MVSS

- We can generalize the VSG approach (used for testing VSS) to test for MVSS
- A multiversion view serializability graph for a schedule S (denoted by $MVSG(S)$) is a directed graph $MVSG(S) = (V, E)$ such that the nodes V represent Xacts & the edges E represent precedence relations among Xacts:
  - (a) If T_j reads from T_i, then $(T_i, T_j) \in E$
  - (c) If T_j read some object O from T_i & T_l update object O, then either $(T_i, T_l) \in E$ or $(T_l, T_j) \in E$
- Due to (c), there could be multiple MVSG(S) corresponding to S
- Notice that (b) from VSS is missing, as that is for final write
- **Theorem 2: S is MVSS iff $\exists$ some MVSG(S) corresponding to S that is acyclic**

Concurrent Txn: Two Xacts T and T' are defined to be concurrent if they overlap $\to$ i.e., $[start(T), commit(T)] \cap [start(T'), commit(T')] \neq \emptyset$

Snapshot Isolation (SI) — break the "Concurrent Update Property" to not be SI

- Each Xact T sees a snapshot of DB that consists of updates by Xacts that committed before T starts and each Xact T associated with 2 timestamps:
  - start(T): the time that T starts — commit(T): the time that T commits
- $W_i(O)$ creates a version of O denoted by O_i
- O_i more recent/newer version compared to O_j if $commit(T_i) > commit(T_j)$
- $R_i(O)$ reads either its own update (if $W_i(O)$ precedes $R_i(O)$) or the latest version of O that is created by a Xact that committed before T_i started; i.e., If $R_i(O)$ returns O_j, then:
  1. Either $j = i$ if $W_i(O)$ precedes $R_i(O)$;
  2. Or $commit(T_j) < start(T_i)$, and
     - (a) For every Xact T_k, $k \neq j$, that has created a version O_k of O, if $commit(T_k) < start(T_i)$, then $commit(T_k) < commit(T_j)$
- **Concurrent Update Property:** If multiple concurrent Xacts update same object, only 1 of the Xacts is allowed to commit, if not, schedule may not be serializable
- Two approaches to enforce the concurrent update property:
  - First Committer Wins (FCW) Rule OR First Updater Wins (FUW) Rule

First Committer Wins (FCW) Rule

- Before committing a Xact T, the system checks if there exists a committed concurrent Xact T' that has updated some object that T has also updated
- If T' exists, then T aborts

First Updater Wins (FUW) Rule

- Whenever a Xact T needs to update an object O, T requests for a X-lock on O
- If the X-lock is not held by any concurrent Xact, then
  - T is granted the X-lock on O
  - If O has been updated by any committed concurrent Xact, then T aborts
  - Otherwise, T proceeds with its execution
- Otherwise, if the X-lock is being held by some concurrent Xact T', then T aborts or commits
  - If T' aborts, then
    - Assume that T is granted the X-lock on O
    - If O has been updated by any concurrent Xact, then T aborts
    - Otherwise, T proceeds with its execution
  - If T' commits, then T is aborted
- When a Xact commits/aborts, it releases its X-lock(s)

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

Snapshot Isolation Tradeoffs

- Performance of SI often similar to *Read Committed*
- Unlike Read Committed, SI does not suffer from lost update/unrepeatable read
- But SI is vulnerable to some non-serializable executions
  - Write Skew Anomaly OR Read-Only Transaction Anomaly
- Snapshot isolation does not guarantee serializability

Write Skew Anomaly

T_1	T_2
$R_1(x_0)$ $R_1(y_0)$	
	$R_2(x_0)$ $R_2(y_0)$
$W_1(x_1)$ $Commit_1$	
	$W_2(y_2)$ $Commit_2$

The above is a SI schedule that is not a MVSS

Read-Only Transaction Anomaly

T_1	T_2	T_3
$R_1(y_0)$		
	$R_2(x_0)$	
$W_1(y_1)$ $Commit_1$		
	$R_2(y_0)$ $W_2(x_2)$ $Commit_2$	
		$R_3(x_0)$ $R_3(y_1)$ $Commit_3$

The above is a SI schedule that is not a MVSS

Serializable Snapshot Isolation (SSI) Protocol

- A schedule S is a serializable snapshot isolation (SSI) schedule if S is produced by the snapshot isolation protocol (i.e., S is a SI schedule) and S is MVSS

10. Crash Recovery

- **Recovery Manager** guarantees atomicity and durability of transactions
  - **Commit(T)** - install T's updated pages into database
  - **Abort(T)** - restore all data that T updated to their prior values
  - **Restart** - recover database to a consistent state from system failure
    - abort all active Xacts at the time of system failure
    - installs updates of all committed Xacts that were not installed in the database before the failure

Interaction of Recovery & Buffer Managers

- Steal policy: allows dirty pages updated by T to be replaced from buffer pool before T commits
- Force policy: requires all dirty pages updated by T to be written to disk when T commits

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

- No-steal policy $\implies$ no undo
- Force policy $\implies$ no redo

Log-based Database Recovery

- Log (aka trail/journal): history of actions executed by DBMS
  - Contains a log record for each write, commit, & abort
- Log is stored as a sequential file of records in stable storage
- Log is stored in stable storage (storage that can survive crashes & media failures)
  - Implemented by maintaining multiple copies of information (possibly at different locations) on non-volatile storage devices

Log Sequence Number (LSN): monotonically increasing. Recently created log records are buffered in the main memory before they are flushed (i.e., written) to the log file on disk. Recovery manager maintains a metadata known as flushedLSN to keep track of the LSN of the last log record that has been flushed to disk

ARIES Recovery Algorithm: steal, no-force approach, assumes strict 2PL

Recovery-related Structures

LOG

	prevLSN	XactID	type	pageID	length	offset	before image	after image
10	null	T_1	update	P500	3	21	ABC	DEF
20	null	T_2	update	P600	3	41	HIJ	KLM
30	20	T_2	update	P500	3	20	GDE	QRS
40	10	T_1	update	P505	3	21	TUV	WXY

DIRTY PAGE TABLE

pageID	recLSN
P500	10
P600	20
P505	40

XACT TABLE

XactID	lastLSN	status
T_1	40	U
T_2	30	U

Normal Transaction Processing

Updating Xact Table (transID, lastLSN, status)

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

Updating Dirty Page Table (pageID, recLSN)

- When a page P in buffer pool is updated & DPT has no entry for P, create a new table entry for P with recLSN = LSN of *log record corresponding to update*
- When a dirtied page P in buffer pool is *flushed to disk*, remove entry for P

Types of log records

- All log records have the following information:
  1. type of log record (e.g., update, commit, abort),
  2. identifier of Xact, and `prevLSN` (LSN of previous log record for same Xact)
- **Update log record**: created after updating page P, update pageLSN = LSN of r
  - Additional fields in ULR: pageID, offset, length, before-image, after-image
- **Compensation log record (CLR)**
  - When update described by an ULR is undone, create a CLR
  - Additional fields in CLR:
    - page ID, undoNextLSN = LSN of next log record to be undone (prevLSN in ULR), action taken to undo update (length/offset/before-image)
- **Commit log record**: created when Xact is committed
- **Abort log record**: created when Xact is aborted, undo initiated for this Xact
- **End log record**: once the additional follow-up processing initiated by a aborted/committed Xact T has completed, create an end log record for T
- **Checkpoint log record**
- **Update log records** & **CLRs** are classified as redoable log records

Implementing Abort: Undo all updates by Xact to database pages

- **Write-ahead logging (WAL) protocol**
  - Do not flush an uncommitted update to the database until the log record containing its before-image has been flushed to the log
    - Before flushing a **database page** P to disk, ensure that all the log records up to the log record corresponding to P's **pageLSN** have been flushed to disk
    - Ensure that P's pageLSN $\leq$ flushedLSN before P is flushed to disk
- For each log record of Xact in reverse order, restore its before-image to undo all
- **How to efficiently retrieve Xact's log records in reverse order?**
  - Xact Table (TT) - maintains one entry for each active Xact
  - Each TT entry stores the LSN of most recent log record for Xact (lastLSN)
  - Use **lastLSN** to retrieve the most recent log record for Xact; the other log records for Xact are retrieved via the **prevLSN** of each retrieved log record
- **Logging Changes During Undo:** Changes made to database while undoing a Xact are also logged (using compensation log records) to ensure that an action is not repeated in the event of repeated undos

Implementing Commit: Need to ensure that all the updates of Xact must be in stable storage (database or log) before Xact is committed

- **Force-at-commit protocol**: Do not commit a Xact until the after-images of all its updated records are in stable storage (database or log)
- Write a commit log record for Xact, then flush all the log records for Xact to disk
- Considered committed if its commit log record has been written to stable storage

Implementing Restart Recovery from system crashes consists of three phases:

- **Repeat history on redo**: During restart following a crash, first restore system to state before crash, then undo actions of Xacts that are active at the time of crash
- **Log changes during undo**: Each undo action is logged with a CLR

1. Analysis: identifies dirtied buffer pool pages & active Xacts at time of crash
2. Redo: redo actions to restore database state to what it was at time of crash
3. Undo: undo actions of Xacts that did not commit

Analysis Phase — Fuzzy checkpointing highlighted in green

- **Retrieve begin_checkpoint log record** (BCPLR) identified by the master record
- **Retrieve end_checkpoint log record** (ECPLR) corresponding to BCPLR
  - For simplicity, assume that there are no log records between BCPLR & ECPLR (i.e., ECPLR is the next log record after BCPLR)
- Initialize (DPT) & Xact table (TT) to be empty — using ECPLR's contents
- Scan the log in forward direction (starting from ECPLR) to process each **log record r** (for **Xact T**)
  - If r is an end log record $\to$ Remove T from TT
  - Else
    - Add entry in TT for T if not in TT, Update lastLSN of entry to be r's LSN
    - Update status of entry to C if r is a commit log record
  - If (r is a redoable log record for page P) & (P is not in DPT), then
    - Create an entry for P in DPT with pageID of entry = P's pageID and recLSN of entry = r's LSN
- At the end of Analysis phase:
  - Xact table = list of all active Xacts (with status = U) at time of crash
  - dirty page table = superset of dirty pages at time of crash

Redo Phase — Fuzzy checkpointing highlighting in green

- RedoLSN = smallest recLSN among all dirty pages in DPT
- Let r be the log record with LSN = RedoLSN
- Scan the log in forward direction starting from r
- If (r is an update log record) or (r is a CLR) then — If (r is a redoable record) and (condition C is false) then
  - Fetch page P that is associated with r
  - If (P's pageLSN $<$ r's LSN) then
    - Reapply logged action in r to P, Update P's pageLSN = r's LSN
  - Else: Update P's entry in DPT: recLSN = P's pageLSN + 1
- At the end of Redo Phase,
  - Create end log records for Xacts with status = C in Xact Table & remove their entries from Xact Table $\Rightarrow$ System is restored to state at time of crash

Undo Phase — Fuzzy checkpointing highlihighted in green

- **Goal**: abort active Xacts at time of crash (loser Xacts)
  - Abort loser Xacts by undoing their actions in *reverse order*
- Initialize L = set of lastLSNs (with status = U) from TT
- Repeat until L becomes empty
  - delete the largest LSN from L
  - let r be the log record corresponding to the deleted LSN
  - if r is an update log record for Xact T on page P then
    - create a CLR r_2 for T: r_2's undoNextLSN = r's prevLSN
    - update T's entry in TT: lastLSN = r_2's LSN
    - create a DPT entry for P (with recLSN = r_2's LSN if P not in DPT)
    - undo the logged action on page P
    - update P's pageLSN = r_2's LSN
    - **Update-L-and-TT**(r's prevLSN)
  - else if r is a CLR for Xact T then
    - **Update-L-and-TT**(r's undoNextLSN)
  - else if r is abort log record for Xact T then **Update-L-and-TT**(r's prevLSN)
- **Update-L-and-TT** (lsn): if lsn is not null then add lsn to L
  - else create an end log record for T & remove T's entry from TT

Simple Checkpointing

1. Stop accepting any new update, commit, & abort operations
2. Wait till all active update, commit, & abort operations have finished
3. Flush all dirty pages in buffer
4. Write a checkpoint log record containing Xact table
5. Resume accepting new update, commit, & abort operations

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

Fuzzy Checkpointing in ARIES

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

- During restart recovery, *Analysis* begins with the begin_checkpoint log record (BCPLR) identified by the master record
  - Let ECPLR = end_checkpoint log record (ECPLR) corresponding to BCPLR
  - Assume that there are no log records between BCPLR & ECPLR
  - Initialize Xact table with ECPLR's Xact table and DPT with ECPLR's DPT