Sort-Select

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

External Merge sort. Number of sorted runs is $\lceil \frac{N}{B} \rceil$. Total number of passes is $\lceil log_{B-1}(N_0) \rceil + 1$. Total number of IO is $2N \times (\lceil log_{B-1}(N_0) \rceil + 1)$. Total number of merging passes (excludes pass 0), is $\lceil log_f(N) \rceil$, where N is the number of sorted runs and f is the number of way merges (merging factor).
Optimisation with blocked IO. Read and write in units of buffer blocks of b pages. Use 1 buffer block for output. Number of runs that can be merged at each pass is $F = \lfloor \frac{B}{b} \rfloor + 1$. Number of passes is $\rceil log_F(N_0) \rceil + 1$.
Sorting with B-tree. Sequentially scan leaf pages. Format 1 gives records directly, format 2,3 need to perform RID lookup.
Covering Index. An index for a query where all the attributes referenced in the query are part of the key or include columns of the index. The query can be evaluated using the index **without** any RID lookup.
Term. Operations between an attribute and a constant or between attributes.
Conjunct. 1 or more terms conencted by OR.
Disjunctive. Conjunct that contains OR.
Conjunctive Normal Form(CNF) predicate. Consists of 1 or more conjuncts connected by AND.

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

B+ tree Matching predicates. If a B+ tree has the following index $K_1, K_2, \dots$ and we have a non-disjunctive CNF predicate p, then the index matches p if p is of the form

$$\underbrace{(K_1 = c_1) \wedge \cdots \wedge (K_{i-1} = c_{i-1})}_{\text{zero or more equality predicates}} \wedge (K_i \ op_i \ c_i),\ i \in [1, n]$$

where $K_1, \dots, K_i$ is a **prefix**. In a B tree, matching entries are in adjacent pages.
Hash Index Matching predicates. Non-disjunctive CNF predicate p. Then hash index matches p if p is of the form $(K_1 = c_1) \wedge (K_2 = c_2) \dots \wedge (K_n = c_n)$
Primary Conjuncts. Subset(**not necessarily prefix**) of conjuncts in a selection predicate that matches an index. Hash index must contain entire set.
Covered Conjunct. All attributes in the conjunct in the predicate appears in the key or include columns of the index. Primary conjuncts is a proper subset of covered conjuncts.
Cost of evaluating p in B+ tree. Navigate internal nodes to locate first leaf page

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

Scanning leaf pages to access all qualifying data

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

Retrieving qualified data records via RID lookup

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

Reduce cost of RID lookup by first sorting the RID (making it clustered)

$$\lceil \frac{||\sigma_{p_c}(R)||}{b_d} \rceil \leq \text{cost}_{RID} \leq \min\{||\sigma_{p_c}(R)||, |R|\}$$

Cost of hash index evaluation. For format 1: cost to retrieve data records $\geq \lceil \frac{||\sigma_{p'}(R)||}{b_d} \rceil$ For format-2, cost to retrieve data entries $\geq \lceil \frac{||\sigma_{p'}(R)||}{b_i} \rceil$. Cost to retrieve data records is 0 if I is a covering index, $||\sigma_{p'}(R)||$ otherwise

Projection and Join

Notation. $\pi_L(R)$ preserves duplicates while $\pi_L^*(R)$ does not.
Sort-based Approach. Better if there are many duplicates or if distribution of hashed values is non-uniform. Extracting cost - Scanning records $|R|$ and the cost to output the result $|\pi_L^*(R)|$. Sorting cost - $2|\pi_L^*(R)|(log_m(N_0) + 1)$ where N_0 is the number of initial sorted runs and m is the merge factor. Removing duplicates cost - $|\pi_L^*(R)|$
Optimised sorting approach. Split the sorting into 2 steps - creating and merging the sorted runs. Combine the creation step with the extraction and merging with removing duplicates.
Hash-based Partitioning phase. Use 1 buffer for input and remaining for output. Read 1 page at a time into input buffer. For each tuple in input buffer, project out unwanted attributes. Then apply the hash function to distribute the tuple into 1 of the output. Flush the output buffer to disk whenever buffer is full.
Hash-based Duplicate elimination. For each partition R_i, initialise an in-memory hash table. Read $\pi_L^*(R_i)$ 1 page at a time. For each tuple read, hash into a bucket using a different hash function and insert it if its not duplicated. Output the tuples in hash table.
Partition Overflow. Hash table for $\pi_L^*(R_i)$ is larger than memory buffer. Recursively apply hash-based partitioning to overflowed partition.
Avoiding partition overflow. Size of hash table for each $R_i = \frac{|\pi_L(R)|}{B-1} \times f$. Approximately $B > \sqrt{f \times |\pi_L^*(R)|}$
Cost if no overflow. $|R| + |\pi_L^*(R)|$ for partioning. $|\pi_L^*(R)|$ for duplicate elimination.
Comparison with sort-based. If $B > \sqrt{|\pi_L^*(R)|}$, same I/O cost as hash-based approach. $N_0 = \lceil \frac{|R|}{B} \rceil \approx \sqrt{|\pi_L^*(R)|}$ initial sorted runs. $\log_{B-1}(N_0) \approx 1$ merge passes.
Using Indexes. If theres an index whose search key (and any include columns) contains all wanted attributes, use index scan. If index is ordered and search key includes wanted attributes as a prefix, scan data entries in order and compare adjacent data entries for duplicates.
Tuple-based nested loop. For every tuple in R and for every tuple in S, if there is match in the tuple, output result. Cost: $|R| + ||R|| \times |S|$
Page-based nested loop. For every page in R and for every page in S do tuple-based nested loop. Cost: $|R| + |R| \times |S|$.
Block nested loop. Assuming $|R| \leq |S|$, allocate 1 page for S, 1 page for output and remaining for R. R is outer and S is inner. While scanning of R is not done, read the next $B - 2$ pages of R into the buffer. Do page-based nested loop. Cost: $|R| + (\lceil \frac{|R|}{B-2} \rceil \times |S|)$
Index Nested Loop. There is an index on the join attribute(s) of S. Consider $R(A, B) \bowtie_A S(A, C)$. Suppose theres a B+ tree index on $S.A$. Use tuple in R to probe the B+ tree to find matching. Assuming uniform distirbution, Cost:

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

Sort-merge join. Sort both relations on join attributes and merge them if have same value for join attributes. Sorting cost: $2|R|(log_m(N_R) + 1) + 2|S|(log_m(N_S) + 1)$. Merging cost best case: $|R| + |S|$, worse case: $|R| + ||R|| \times |S|$.
Optimisation. Sort until $B > N(R, i) + N(S, j)$. $N(R, i)$ is total number of sorted runs of R at the end of pass i of sorting R. Suppose $|R| \leq |S|$ and if $B > \sqrt{2|S|}$. Number of initial sorted runs of $S < \sqrt{\frac{|S|}{2}}$. Total no. of initial sorted runs $< \sqrt{2|S|}$. 1 pass sufficient to merge and join. Cost: $3(|R| + |S|)$
Grace Hash Join. Partition each relation into k parts. Probe each R_i with S_i. R_i to build and S_i to probe. Minimise size of each partition let $k = B - 1$. Suppose uniform hashing, then require $B > \sqrt{f \times |R|}$. If there is overflow, recursively apply partitioning. Cost: $3(|R| + |S|)$
Multiple equality-join conditions. Index nested loop join by using index on all/some of join attributes. Sort merge join need to sort on combination of attributes. Others unchanged.
Inequality-join conditions. Cannot use sort-merge and hash based.

Evaluation Optimiser

Aggregation. Maintain running information while scanning table. If there's covering index, aggregation can be done from index data entries instead.
Group by operation. Sorting approach: Sort relations on grouping attributes and scan to compute aggregate for each group. Hashing approach: scan relation to build hash table on grouping attributes. For each group maintain grouping value and running info.
Materialised Evaluation. Operator evaluated only when each of its operands have been evaluated. Intermediate results written to disk.
Pipelined evaluation. Output produced by operator passed directly to parent. Execution is interleaved.
Blocking operator. Operator may not be able to produce output until it received all input tuples from its children(external merge sort, grace hash joins, sort-merge join).
Iterator interface. Top-down, demand driven. 3 methods: open (initialisation), getNext(generate next output), close(deallocate).
Hybrid. Materialise if repeatedly scanned (eg in nested loop join).
Join plans. LHS is outer/probe relation and RHS is inner/build relation.
Idempotence of unary ops. $\pi'_L(\pi_L(R)) = \pi'_L(R)$ if $L' \subseteq L \subseteq$ attr(R). $\sigma_p1(\sigma_p2(R)) = \sigma_{p1 \wedge p2}(R)$
Commutating selection with proj. $\pi_L(\sigma_p(R)) = \pi_L(\sigma_p(\pi_{L \cup \text{attr(p)}}(R)))$
Commutating selection with binary op. $\sigma_p(R \text{ op } S) = \sigma_p(R) \text{ op } S$ where op $\in \{\times, \bowtie\}$ and attr(p) $\subseteq$ attr(R). $\sigma_p(R \cup S) = \sigma_p(R) \cup \sigma_p(S)$

6. **Commutating projection with binary operators**
Let $L = L_R \cup L_S$, where $L_R \subseteq attributes(R)$ and $L_S \subseteq attributes(S)$
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 $attributes(p) \cap attributes(R) \subseteq L_R$ and $attributes(p) \cap attributes(S) \subseteq L_S$
6.3 $\pi_L(R \cup S) \equiv \pi_L(R) \cup \pi_L(S)$

Optimisation idea. Binary ops (except set ops) are commutative (A+B=B+A) and associative ((A+B)+C=A+(B+C))Push the selection by reducing the size of tables to be joined. Apply selections early to reduce size of tables. Typically done using commutating selection. Possible to push projections too via 6.2.
Query plan trees. Linear if at least 1 operand is a base relation, otherwise bushy. Left-deep if every right join operand is a base. Right-deep if every left join operand is a base.
Query Plan Enumeration. Use DP to get best plan. Start by performing on 1 relation, then on every permutation of 2 relation joins until all the joins are used.
System R Optimiser. Enumerate only left-deep query plans. Avoid cross product and considers early selection and projection.
Enchance DP. Consider sort order of query plan output. optPlan(S_i, o_i) compared to optPlan(S_i) is the cheapest query plan for S_i with output ordered by o_i if $o_i \neq$ null.
Cost est assumptions. Uniformity and independence assumption. For $R \bowtie_{R.A=S.B} S$, if $||\pi_A(R)|| \leq ||\pi_B(S)||$ then $\pi_A(R) \subseteq \pi_B(S)$. Every R tuple joins with some S tuple (inclusion assumption).
Size estimation. For query $q = \sigma_p(e)$, where $p = t_1 \wedge t_2 \dots$.
Selectivity factor. Same as reduction factor. Fraction of tuples e that satisfies t_i, $rf(t_i) = \frac{||\sigma_{t_i}(e)||}{||e||}$. So $||q|| \approx ||e|| \times \prod_{i=1}^n rf(t_i)$
Join Selectivity factor. $rf(R.A = S.B) \approx \frac{1}{max\{||\pi_A(R)||, ||\pi_B(S)||\}}$
Equiwidth histogram. Each bucket has almost equal number of **values**.
Equidepth histogram. Each buck has almost equal number of **tuples**. Subranges of adjacent buckets might overlap.
Histogram with MCV. Separately keep track of frequencies of top-k most common value and exclude MCV from histogram buckets.

Transaction Management

ACID Properties. **Atomicity** (all or nothing of the actions in Xact happen), **Consistency** (If each Xact is consistent and DB starts consistent, DB ends up consistent), **Isolation** (Isolated execution of Xact) and **Durability** (Committed Xact have effects persisted).
Serial schedule. Actions of Xacts not interleaved
Read from. T_j reads O from T_i if last action on O before $R_j(O)$ is $W_i(O)$. T_j reads from T_i if T_j has read some object from T_i.
Final write. T_i performs final write on O if the last action on O is $W_i(O)$.
View Equivalent. If 2 schedules have the same read

from and final write on all objects.
View Serialisable. View equivalent to some serial schedule.
Testing view serialisability. Create a DAG where nodes are Xact and edges are precedence. If T_i read from T_j then $T_j \to T_i$. If both T_i, T_j update same object O and T_i performs final write, then $T_j \to T_i$. If T_j read object O from T_k and T_i update object O, then either $T_i \to T_k$ or $T_j \to T_i$. If cyclic, then not VSS. Otherwise, there must be some topo ordering that is view equivalent to S.
Conflict. 2 actions on the same object. At least 1 is a write and the actions are from different Xact.
Dirty Read. T_2 read an obj modified by T_1 and T_1 **hasn't committed**, WR conflict.
Unrepeatable Read. T_2 updates obj that T_1 read and T_2 commits first, RW conflict.
Lost update. T_2 overwrite value of object modified by T_1 while T_1 in progress, WW conflict.
Conflict equivalent. Ordering of every pair of conflicting actions of 2 **committed** Xacts are the same.
Conflict serialisable schedule. Conflict equivalent to a serial schedule. Conflict serialisable is view serialisable, but otherway may not hold.
Testing CSS. Nodes represent committed Xacts. Edges contain (T_i, T_j) if an action T_i happens before and conflicts with T_j. Conflict serialisable if graph is acyclic.
Blind writes. A write on object O by T_i if T_i did not read O prior to writing. If S is view serialisable and no blind writes, then S is conflict serialisable.
Cascading aborts. If T_i read from T_j and T_j abort, then T_i must abort for correctness.
Recoverable schedule. If for every Xact T commits in S, T must commit after T' if T reads from T'.
Cascadeless Schedules. All read operations are non-dirty (ie no WR). Cascadeless schedule is also a recoverable one.
Before-images. Log before action happens and restore that (schedule must be strict).
Strict schedule. All read and write operations are non-dirty (ie no WR and no WW).

Concurrency

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

Locking modes. s-lock for reading and x-locks for reading and writing.
Lock-based CC. If lock request not granted, T becomes blocked. Its execution is suspended and T is added to O's request queue.
2PL Protocol. Can release locks anytime. Once Xact releases a lock, Xact cannot request any more locks. 2PL schedules are conflict serialisable. Growing (before releasing 1st lock) and shrinking(after releasing 1st lock) phase.
Strict 2PL. Xact must hold on to locks until Xact commits/aborts. Strict 2PL schedules are strict and conflict serialisable.
Deadlock. Cycle of Xacts waiting for locks to be released by each other.
Waits-for graph. Node represent active Xacts. Add an edge $T_i \to T_j$ if T_i is waiting for T_j to release lock. Deadlock detected if WFG has cycle. Break deadlock by aborting a Xact in cycle.

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. Lower priority Xact never wait for higher priority. Non-preemptive (only Xact requesting for lock can get aborted). Younger Xact may get repeatedly aborted. A Xact that has all the locks it needs is never aborted.
Wound-wait. Higher priority Xact never wait for lower-priority.
Lock conversion. Increases concurrency. Only in growing phase.
Lock upgrade. $UG_i(A)$. Upgrade request is blocked if another Xact is holding s-lock on A. Upgrade request allowed if T_i has not release any lock.
Lock downgrade. $DG_i(A)$. Allowed if T_i has not modified A and T_i has not released any lock.
Phantom Read. $R(p)$ reads all objects that satisfies a selection predicate, p. $R(p), W(x)$ conflict if object x satisfy p.

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. Lock acquired for an operation could be released **after** operation **before** Xact commits/abort.
Long duration lock. Lock acquired for an operation is held **until** Xact commits/aborts.
Lock granularity. Size of data items being locked. From highest to lowest: database > relation > page > tuple.
Multi-granular lock. If Xact T holds a lock mode M on a data granule D, then T **implicitly** holds lock mode M on granules finer than D.
Intention locking. Before acquiring locks on data granule G, need to acquire I-locks on granules coarser than G in a **top-down** manner.

Lock compatability matrix

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

Lock compatability matrix

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

Protocol. To obtain S/IS lock, must already hold IS/IX lock on its parent. To obtain X/IX lock, must already hold IX lock on its parent. Locks are released in bottom up but acquired top down.

MVCC

Maintian multiple versions of each object. $W_i(O)$ creates new version. $R_i(O)$ reads an appropriate version. Read-only Xacts not blocked by update Xacts and vice versa. Read-only Xacts are never aborted.
Multiversion Schedule. A read action can return **any** version
Multiversion View Equivalence. 2 schedules, S and S', over the same set of Xacts are multiversion view equivalent if they have the same set of read-from relationships. $R_i(x_j)$ occurs in S iff $R_i(x_j)$ occurs in S'.
Monoversion Schedule. Every read returns latest version.
Serial Monoversion Schedule. A monoversion schedule that is also a serial schedule.
Multiversion View Serialisability. A serial monoversion schedule (over the same set of Xacts) that is multiversion view equivalent to a multiversion schedule, S. A VSS is also a MVSS but not necessarily the other way.
Snapshot isolation. Each Xact T sees a snapshot of DB that consists of updates by Xacts that committed before T starts.
Concurrent Transactions. $[\text{start}(T), \text{commit}(T)] \cap [\text{start}(T'), \text{commit}(T')] \neq \emptyset$
Writes. $W_i(O)$ creates a version O_i. O_i is a newer version compared to O_j if commit(T_i) ¿ commit(T_j)
Reads. Read either its own update or the latest version of O that is created by a Xact that committed before T_i started.
Concurrent Update Property. If multiple concurrent Xacts update same object, only 1 xact can commit. Otherwise, schedule may not be serialisable.
First Committer Wins. Before T commit, check if ∃ committed concurrent Xact, T' that updated some object that T updated. If T' exists, abort T, otherwise commit T.
First Updater Wins. T requests for X-lock. If lock not held, follow FCW. Otherwise, wait until T' finish. If T' abort, get lock and follow FCW. Otherwise, abort T.
Garbage collection. Delete a version O_i if ∃ a newer version O_j (commit(T_i) < commit(T_j)) st for every active Xact T_k that started after commit of T_i, we have commit(T_j) < start(T_k)
Tradeoffs. Similar to Read Committed but dont have lost update/unrepeatable read anomalies. Does not guarantee serialisability.
Anomaly. $T_1 \overset{rw}{\dashrightarrow} T_2 \overset{rw}{\dashrightarrow} T_1$ (write skew), $T_3 \overset{rw}{\dashrightarrow} T_2 \overset{rw}{\dashrightarrow} T_1 \xrightarrow{wr} T_3$ (read-only)
Serialisable SI. A schedule is that is produced by SI and is MVSS.
Detection. Keep track of rw depedencies. If $T_1 \overset{rw}{\dashrightarrow} T_2 \overset{rw}{\dashrightarrow} T_3$, abort one of them. Possible for false positives.
Xact dependencies. ww - T_1 write a version and T_2 writes to immediate successor. wr - T_1 write a version and T_2 reads this version. rw - T_1 reads a version/some data item and T_2 create immediate successor.
Immediate successor. x_j immediate successor of x_i if T_i commit before T_j and no xact commits between T_i and T_j that produces another verison of x.
Dependency Serialisation graph. Nodes are committed Xacts. Edges represent dependencies. $\dashrightarrow / \to$ for concurrent/non-concurrent
Non-MVSS SI schedule. If S is SI but not MVSS, then there is a cycle in DSG and for each cycle, $\exists T_i, T_j, T_k$ st T_i and T_k may be the same xact (write-skew), $T_i \overset{rw}{\dashrightarrow} T_j \overset{rw}{\dashrightarrow} T_k$

Crash Recovery

Undo. Remove effects of aborted Xact to preserve atomicity.
Redo. Re-installing effects of committed Xact for durability.
Steal Policy. Allow dirty pages updated by Xact T to be replaced from buffer pool before T commits. No steal means no undo - may run out of buffer pages.
Force Policy. Requires all dirty pages updated by Xact T to be written to disk when T commits. Force policy means no redo - incurs random IO.
ARIES. Steal and no force. Strict 2PL for concurrency control. Data structures are log files, Xact table(TT) and dirty page table(DPT).
TT. 1 entry for each active (XactID, lsn of most recent LR for this Xact, C/U status)
DPT. 1 entry for each dirty page in buffer pool (pageID, LSN of earliest LR for an update that caused page to be dirty)
LRs. (type, XactID, prevLSN, other info)
Write-ahead Protocol. For implementing abort. Dont flush an uncommitted update to the DB until LR containing its before-image been flushed to log.
Enforcing. Each db page contains **pageLSN** (LSN of latest update). Before flushing db page to disk, ensure all LRs ≤ P's pageLSN have been flushed to disk.
Force-at commit. Dont commit a Xact until after-imgs of all its updated records are in stable storage. Commit LR is created if Xact considered committed if CLR has been written to stable storage.
Implementing Restart. Analysis,redo,undo phase.
Analysis phase. Initialise empty DPT and TT. Iterate through logs in a forward direction. If r is end LR, remove T from TT. Otherwise, add an entry in TT for T if T isnt in TT. Update lastLSN of entry to be r's LSN. Update status of entry to C if r is a CLR. If (r is a redoable LR for page P and P not in DPT), create an entry for P in DPT with pageID = P.pageID and entry.recLSN = r.LSN.
Redo phase. RedoLSN = smallest recLSN in DPT. Iterate through records starting from RedoLSN. If (r is update LR || r is CLR) then fetch page P associated with r. If (P.pageLSN < r.LSN) then reapply logged action in r to P, P.pageLSN = r.LSN.
Undo Phase. Abort active Xacts at time of crash (loser Xact) by undoing actions in reverse order. Initialise L to be set of lastLSN(with status = U) from TT. Repeat until L empty. Let r be the LR of largest lastLSN and delete it from L. If r is an update LR for Xact T on page P then create a CLR r_2 for T (r_2.undoNextLSN = r.prevLSN). Update TT - T.lastLSN = r_2.LSN. Undo logged action on P. P.pageLSN = r_2.LSN. UpdateLandTT(r.prevLSN). Else if r is CLR for Xact T then UpdateLandTT(r.undoNextLSN). Else if r is abort LR for xact T then updateLandTT(r.prevLSN).
updateLandTT(lsn). if lsn is not null then add lsn to L. Else create an end LR for T and remove T from TT.
Simple Checkpointing. Stop accepting and wait for all ops to stop. Flush everything and write checkpt LR containing Xact table. Then resume. Analysis phase can use Xact table from this.
Fuzzy Checkpointing. Write begin and end CPLR. At the end CPLR, write DPT' and TT'. Write special master record containing LSN of begin CPLR to a known place on stable storage.
Analysis Phase. Initialise DPT and TT from ECPLR content.
Optimisation condition. $P \notin DPT$ or DPT P.recLSN > r.LSN. Update of r already applied to P.
Optimised Redo. Change if condition to be r is redoable and optimisation condition dont hold. And add a else to update P.recLSN = P.pageLSN + 1 for the P in DPT.
Fuzzy Undo. Between updating T in TT and undoing logged action on P, create DPT entry for P (with recLSN = r_2.LSN) if P not in DPT.