CS3223
AY22/23 Sem 2
github.com/SeekSaveServe

L4: Query Evaluation - Sort, Select

External Merge Sort - Bocked I/O

- Read and write in blocks of **b** buffer pages (replace b with 1 for unoptimised)
- $\lfloor\frac{B-b}{b}\rfloor$ blocks for input, 1 block for output
- Can merge at most $\lfloor\frac{B-b}{b}\rfloor$ sorted runs in each merge pass
- $F = \lfloor\frac{B}{b}\rfloor - 1$ runs can be merged at each pass
- Num passes = $\log_F N_0$
- New cost: $2N(\lceil\log_F N_0\rceil + 1)$

B+ tree sort

- B+ Tree is sorted by key
- Format 1 (clustered): Sequential Scan
- Format 2/3:Retrieve data using RID for each data entry
- Unclustered implies more I/Os

Access Path refers to the different ways to retrieve tuples from a relation. It is either a **file scan** or a **index plus matching selection condition**. The more **selective** the access paths, the fewer pages are read from the disk.

- Table scan: scan all data pages
- Index scan: scan all index pages
- Table intersection: combine results from multiple index scans (union, intersec). Find RIDs of each predicate and get the intersection

Query: Selection Covering Index

- I is a covering index of $query_Q$ if I contains all attributes of Q
- No RID lookup is needed, Index-only plan
- If data is unclustered, unsorted, no index $\rightarrow$ best way is to collect all entries and sort by RID before doing I/O

CNF Predicate

- Find RIDs of each predicate and get the intersection
- Conjuncts are in the form (R.A op c V R.a op R.b)
- CNF are conjuncts (or terms) connected by $\wedge$

Matching Predicates - B+ Tree

- Non-disjunctive CNF (no $\vee$)
- At most one non-equality comparison operator which must be on the **last attribute in the CNF**
- $(k_1 = c_1) \wedge (k_2 = c_2) \wedge ...k_i opc_i | I = (k_1, k_2...k_n)$
- The order of k matters, and there cannot be missing K_i in the middle of the CNF
- Having inequality operator before equality operator makes the query to be less selective

Matching Predicates - Hasing

- No inequality operators
  $(k_1 = c_1) \wedge ...k_i = c_n | I = (k_1, k_2...k_n)$
- Unlike B+ tree, **all predicates must match**

I=(age, weight, height), p=$(age \geq 20 \wedge age \geq 18 weight = 50 \wedge height = 150 \wedge level = 3)$

Primary Conjuncts : The subset of conjuncts in p that I matches

Primary Conjuncts: $age \geq 20 \wedge age \geq 18$

Covered Conjuncts : The subset of conjuncts in p that I covers (conjuncts that appear in I). Primary conjunct $\subseteq$ covered conjunct

Covered Conjuncts: $age \geq 20 \wedge age \geq 18 \wedge height = 150$

Cost Notation

Notation	Meaning
r	relational algebra expression
$\|\|r\|\|$	number of tuples in output of r data records
$\|r\|$	number of pages in output of r
b_d	number of data records that can fit on a page [d+1, 2d+1]
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

Cost of B+-tree index evaluation of p

Let p'=primary conjuncts of p (**matching**) — p_c=covered conjuncts of p

1. Navigate internal nodes to locate the first leaf page

$$Cost_{internal} = \begin{cases} \lceil log_F(\lceil\frac{||R||}{b_d}\rceil)\rceil | Format1 \\ \lceil log_F(\lceil\frac{||R||}{b_i}\rceil)\rceil | Otherwise \end{cases}$$

1.1 This is traversing the height of B+ tree

2. Scan leaf pages to access all qualifying data entries

$$Cost_{leaf} = \begin{cases} \lceil\frac{||\sigma_{p'}(R)||}{b_d}\rceil | Format1 \\ \lceil\frac{||\sigma_{p'}(R)||}{b_i}\rceil | Otherwise \end{cases}$$

- This is the cost of reading qualifying conjuncts
- Using p_c would be wrong since covering conjuncts may be non-matching which results in more reads from the leaves.
- Conversely, non-matching (but covered) conjuncts cannot be derived from the B+ tree and needs to be read from disk
- Retrieve qualified data records using RID lookups. 0 if I is covering OR format 1 index. $||\sigma_{p_c}(R)||$ otherwise
- includegraphics[width=5cm, height=1.3cm]optimisation.png

Cost of Hash index evaluation of p (all covered index are matched)

- Format 1: cost to retrieve **data entries** is at least$\lceil\frac{||\sigma_{p'}(R)||}{b_d}\rceil$
- Format 2: cost to retrieve **data entries** is at least $\lceil\frac{||\sigma_{p'}(R)||}{b_i}\rceil$
- Format 2: Cost to retrieve **data records** is 0 if it is a covering index (all information in data entry) OR $||\sigma_{p'}(R)||$ otherwise

L5:Query Evaluation - Projection and Join

- $\pi^*(R)$ refers to projection without removing duplicates
- $\pi(R)$ involves 1.Removing unwanted attributes 2. Removing duplicates
- Sorting is better if we have many duplicates or if hte distribution is nonuniform(overflow more likely for hashing paritions)
- Sorting allows results to be sorted
- If $B > \sqrt{|\pi^*_L(R)|}$, then both sorting and hashing has similar I/O costs ($\lceil\frac{|R|}{B}\rceil \rightarrow |R| + 2 * |\pi^*_L(R)|$)

Approach 1: project based on sorting

- **Naive**: Extract attributes L from records $\rightarrow \pi^*_L(R) \rightarrow$ Sort attributes $\rightarrow$ Remove duplicates
- Cost: Cost to scan records ($|R|$) + Cost to output to temporary result ($|\pi^*_L(R)|$) $\rightarrow$ cost to sort records $(2|\pi^*_L(R)|\log_m(N_0) + 1) \rightarrow$ Cost to scan data records $|\pi^*_L(R)|$
- **Optimisation**: Create Sorted runs with attributes L only (Pass 0) $\rightarrow$ Merge sorted runs and remove duplicates $\rightarrow \pi_L(R)$

Approach 2: project based on hashing

- Build a main-memory hash table to detect and remove duplicates. Insert to the hashtable if then entry is not already in it.
- 1. Partition R into $R_1, R_2...R_{B-1}$, hash on $\pi_L(t)$ for $t \in R \leftarrow (\pi^*_L(R_i)$ does not intersect $\pi^*_L(R_j), i! = j)$

1.1 Use 1 buffer for input and (B-1) for output

1.2 Read R 1 page at a time, and hash tuples into B-1 partitions

1.3 Flush output buffer to disk when full

- 2. Eliminate duplicates in each partition $\pi^*_L(R_i)$
- $\pi_L(R) = \cup_i^{B-1}(\pi_L(R_i))$

2.1 For each partition, Initialise an in-memory hash table and insert each tuple into B_j if $t \notin B_j$

Parition overflow: Hash table $\pi^*_L(R_i)$ is larger than available memory buffers.

Solution: Recursively apply hash-based partitioning to overflowed partitions.

Analysis: Effective (no overflow) when B $> \frac{|\pi^*_L(R)|}{B-1} * f \approx \sqrt{f * \pi^*_L(R)|}$

If no partition overflow: (partition)$|R| + \pi^*_L(R)|$ + (duplicate elimination)$|\pi^*_L(R)|$

Join

$R \bowtie_\theta S$, where R is the outer relation and S is the inner relation

Optimal join

- Cost: $|R| + |S|$
- load smaller relation into memory
- requires: $|S| + 2$ buffers

Tuple-based

- Cost: $|R| + ||R|| * |S|$
- for each tuple r in R
- for each tuple s in S
- if (r matches s) then output (r, s)4 to result

Page-based

- Load P_R and P_S to main memory
- Cost: $|R| + |R| * |S|$
- for each page P_R in R
- for each page P_S in S
- for each tuple $r \in P_R$
- for each tuple $s \in P_S$
- if (r matches s) then output (r, s)4 to result

Block nested-loop

- Allocate 1 page for S, 1 for output, B-2 for R $|R| \leq |S|$
- Cost: $|R| + (\lceil\frac{|R|}{B-2}\rceil * |S|)$
- $|R| \leq |S|$
- while Scanning R
- read next (B-2) pages of R to buffer
- for P_S in S
- read P_S into Buffer
- for $r \in buffer \wedge s \in P_S$
- if (r matches s) then output (r, s)4 to result
- Without materialisation: $\lceil\frac{|R|}{B-2}\rceil * |T|$

Index Nested Loop Join

- There is an index on the join attributes of S
- Uniform distribution: r joins $\lceil\frac{||S||}{||\pi_{B_j}(S)||}\rceil$ tuples in S
- format 1 B+Tree:$|R| + ||R|| * J$
- Assuming unclustered:
- J = height of tree + reading leaf pages + RID Look up
- J = $\log_F(\lceil\frac{||S||}{b_d}\rceil)$ (tree traversal)+$\lceil\frac{||\pi_{p'}(S)||}{b_i}\rceil$ (search leaf nodes) + RID lookup
- for $r \in R$
- use r to probe S's index to find matching tuples

Sort-Merge Join

- Sort R and S on join attributes and merge
- Cost: $2|\pi^*_L(R)|(\log_{B-1}(\frac{|R|}{B}) + 1)$ + $2|\pi^*_L(S)|(\log_{B-1}(\frac{|S|}{B}) + 1)$ + $|\pi^*_L(R)| + |\pi^*_L(S)|$
- merging cost is $|R| + ||R|| * |S|$ if each tuple of R requires a full scan of S
- Optimisation: $B \geq N(R, i) + N(S, i) + 1 \geq \sqrt{|R| + |S|}$
- We can choose which relation to partition again if this is not met
- Cost: cost of getting R, S + k(write out ($|R| + |S|$)) + m(merge ($|R| + |S|$))
- If sorted on join column: $|R| + |S|$

Grace Hash Join

- Partition into $B - 1$ partitions
- If no partition overflow ($B > \sqrt{f * |S|}$):
- k(Cost to partition R, S) + Cost of probe phase
- Partition cost = cost of getting R + cost to write partitions($|R|$)
- Probe cost = $|R| + |S|$

L6: Query Optimisation

1. Search space: Queries considered

- **Search Place** Queries being considered
- **Linear** if at least one operand for each join is a base relation, bushy otherwise
- **Left-deep** if every right join operand is a base relation
- **Right-deep** if every left join operand is a base relation

2. Plan enumeration - for joins between 2 tables

- Basic DP is not always Optimal since it goes for lowest current cost and ignores the sorted property of outputs
- **Enhanced DP:** left-deep only, avoids cross products, considers early selection and projections, considers order of output

3. Cost Model

- **Uniformity:** uniform distribution of all values
- **Independence:** Independent distribution of values in different attributes
- **Inclusion:** for $R \bowtie S, if ||\pi_A(R)|| \leq ||\pi_B(S)||$ then $\pi_A(R) \subseteq \pi_B(S)$

Plans-no join, 1 table

- **Table scan** Scan the entire table. Cost: $|R|$
- **Index scan** Scan the index. Cost: 2 + |leaf pages satisfying the predicate| + ||entries satisfying predicate|| (unclustered)
- **Index intersection with** I_a I_b Cost to find relevant entries from index and materialise(R and s) + cost to intersect partitions 1,2 (block nested, grace hash, sort merge)+ cost to RID lookup (if more attributes are needed)
- cost to partition: Scan index for matching pages + cost to write partitions from matching entries

Histogram

- **Equiwidth** Each bucket has equal number of values

- Estimate: $\frac{1}{|bucket|} * ||bucket||$
- **Equidepth** Each bucket has equal number of tuples
- Sub-ranges can overlap, tuples of the same value can be in 2 adjacent buckets
- $\frac{1}{|bucket_A|} * ||bucket_A|| + \frac{1}{|bucket_B|} * ||bucket_B|| + ...$
- **MCV** Separately track the top-k MCV and exclude them from the bucket

Size of query

- **Join** $||R|| * ||S|| * \frac{1}{max(||\pi_b(R)||, ||\pi_b(S)||)}$
- **Select - OR** $(1 - (p(\,a! = x) * p(\,b! = y)) * ||R||)$
- **Select - AND** $p(a = x) * p(b = y) * ||R||$

L7: Transaction Management

View Equivalent

- If R_i reads A from one write W_j in S, then R_i must also read A from the same write W_j in S'
- For each data object A, Xact (if any) that performs final write on A in S must also perform final write on A in S'

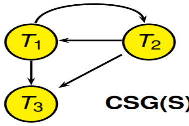
Conflicting actions - WW, WR

- **Dirty Read-WR** T2 read uncommited write from T1. $W_1(X), R_2(X)$
- **Unrepeatable Read-WR** T2 updates an object that T1 has previously read and T2 commits while T1 is still in progress $\rightarrow$ T1 can get a different value from read. $R_1(X), W_2(X), C_2, R_1(X)$
- **Lost Update-WW** T2 overwrites the value of an object that has been modified by T1 while T1 is still in progress. $R_1(X), R_2(X), W_1(X), W_2(X)$
- **View serializable - view equivalent to some serial S** cannot be view serializable if the above anomalies occur. Conversely, no anomalies occur if view serializable.
- **Blind write** $R_1(X), W_2(Y), W_1(X)$ Blind write: $W_2(Y)$
- **Conflict Serializable** Conflict equivalent to serial schedule, view serializable and not blind write
- **Non Conflict Serializable** find conflicting action pairs(R1(x) W2(x)), (R2(x) W1(x))
- Conflicting actions does not mean not serializable, there needs to be a cycle
- $CSS \subsetneq VSS \subsetneq MVSS$. The more restrictive, the less concurrent and less resources are needed to check serializability

Conflict Serializability Graph

- V contains a node for each committed Xact in S
- E contains (T_i, T_j) if an action precedes and conflicts with one of T_j's actions

$R_1(A)$, $W_2(A)$, $Commit_2$, $W_1(A)$, $Commit_1$, $W_3(A)$, $Commit_3$



CSG(S)

ACID

- **Atomicity** Either all actions of a transaction are committed or none are
- **Consistency** Each transaction is consistent and DB begins in a consistent state $\rightarrow$ DB ends in a consistent state
- **Isolation** Execution of one xact is isolated from other Xacts
- **Durability** Once a transaction is committed, its effects persists
- Concurrency control ensures isolation
- Recovery manager ensures atomicity and durabilitt
- Consistency is ensured by constraints, cascades and triggers

Schedules

- **Cascading aborts** T_i read from $T_j \rightarrow T_j$ aborts $\rightarrow T_i$ aborts. This requires high book keeping efforts, so we turn to recoverable schedules instead.
- **Recoverable** $\forall T \in S$ T2 must commit after T1 if T2 reads from T1 (or T2 aborts before T1). Can still have cascading aborts. Is compulsory.
- **Cascadeless** Whenever T_i reads from T_j in S, Commit must precede this action. Desirable, not compulsory.
- Theorem 4: Cascadeless $\rightarrow$ Recoverable (not iff)
- **Recovery with before image** Store the before image before write, restore this before image if write aborts. This can lead to **lost update** anomaly if the before image overwrites another Xact's write.
- **Strict** to use before-images, $\forall W_i(O) \in S$, O is not read or written by another Xact until Ti either aborts or commits. This ensures no lost update anomaly during recovery.
- Strict schedules allows recovery using before images to be more efficient but restricts concurrency
- Theorem 5: Strict $\rightarrow$ Cascadless (not iff)

L8: Concurrency Control

Lock based concurrency control

- If lock request is not granted, then T becomes blocked and gets added to O's request queue

2PL

- To read an object O, a Xact must hold a S-lock or X-lock on O
- To write to an object O, a Xact must hold a X-lock on O
- Once a Xact releases a lock, the Xact can't request any more locks
- Theorem 1: 2PL is conflict serializable

Strict 2PL

- A Xact must hold on to locks until Xact commits or aborts
- Theorem 2: Strict 2PL is strict and conflict serializable
- Strict 2PL prevents cascading rollback and deadlock and ensures recoverability

Anomalies not dealt by strict 2PL

- **Phantom Read:** T1 reads a set of objects, T2 inserts a new object in that set, T1 reads the set again and gets a different set of objects. 2PLS cannot prevent this as locks are held at the object level.
- Solved by **predicate locking**, which is done in practice by **index locking**

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

Deadlock Detection

- Waits-for graph (WFG) $\rightarrow$ Deadlock is detected if WFG has a cycle. $(V_i, V_j \rightarrow T_i waits - for T_j)$
- Breaks a deadlock by aborting a Xact in cycle
- Alt: timeout mechanism

Deadlock Prevention

- Each Xact is assigned a timestamp when it starts
- Assume older (smaller time stamp) Xacts have higher priority than younger Xacts
- Tie between blocked/restarted xact brokered by priority, original timestamp is maintained to prevent starvation
- **Wait-die** Higher priority waits for lower priority, lower priority dies if higher priority holds lock (lower never waits)
- **Wound-wait** Higher priority kills lower priority, lower priority waits for higher priority to release lock (higher never waits)

Lock Conversion

- Allows for greater concurrency
- Conversion is only allowed if the Xact has not released any lock
- **Upgrade(A)** blocked if another Xact holds shared or exclusive lock on A
- **Downgrade(A)** allowed if Xact has not modified A and Xact has not released any lock

Improve System Throughput

- Reduce Lock Granularity, Reduce time of lock being held, reduce hotspots in DB by changging schema design

Multi Version Serializable Schedle (MVSS)

Benefits

- $W_i(O)$ Creates new version
- Read-only Xact are not blocked by update xact (vice versa). Mono version (e.g. 2PL) blocks
- Read-only xacts are never aborted (no deadlocks due to Multi-version) or blocked
- **multiversion view equivalent** if S and S' have the same set of read-from relationships
- i.e. Ri (xj) occurs in S iff Ri (xj) occurs in S'
- **Monoversion Schedule** each read action returns the most recently created object version. Not necessarily serializable
- **MVSS** if there exists a serial Monoversion schedule that is multiversion view equivalent to S
- Note that a MVSS is not necessarily conflict serializable schedule if it is not a valid monoversion schedule
- E.g. W1(x1), R2(x0), R2(y0), W1(y1), C1, C2 is MVSS with (T2, T1) but contains conflicting actions W1(x1) and R2(X0)

Snapshot Isolation (SI) [NOT always serializable]

- Similar performance as Read Committed, does not suffer lost update, unrepeatable read.
- Each Xact has a snapshot of the database at the start of the Xact and sees only versions from that snapshot and **its own writes**
- **Concurrent Update Property** If multiple concurrent exacts on same object, only one can commit
- **FUW** T needs to acquire X-lock on O (if not - wait), and if O has been updated by a concurrent T' then T aborts
- **FCW** (no locks) before commiting T checks if O has been updated, abort if it has been updated
- **Write-skew anomaly**, not MVSS: $R_1(X_0), R_2(X_0), R_1(Y_1), R_2(Y_2), W_1(X_1), C_1, W_2(Y_2), C_2$
- **Read-only anomaly**,not MVSS: $R_1(b)$, $R_2(a)$, $W_1(b), C_1, R_2(b), W_2(a), R_3(a), R_3(b), C_3, C_2$
- **Serializable** MVSS to some monoversion

Transaction Dependencies - Making SI serializable

- **WW** from T1 to T2: T1 commits some version of X and T2 writes the immediate successor
- **WR** from T1 to T2: T1 commits some version of X which is read by T2
- **RW** from T1 to T2: T1 reads some version of X and T2 commits the immediate successor
- **DSG** V = xacts, E = Dependencies, use $-->$ for concurrent transactions and $\rightarrow$ for non-concurrent
- **Non-MVSS SI** At least one cycle in DSG(s) with T_i, T_j, T_k s.t T_i, T_k are possibly the same xact, T_i, T_j are concurrent with an edge T_i rw T_j and T_j rw T_k

Locking Granularity

(Most coarse) DB, relation, page, tuple (Finest, more concurrency, less scalable)

Locks are acquired in a top down manner.

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

L9-Crash Recovery

Policies

- **Steal:** Allows dirty pages to write to disk before commit
- No Steal reduces number of free buffer pages
- **Force:** All dirty pages must write to disk when commit
- Force incurs more random disk IO
- Steal: undo, Force: no redo. Pgsql: steal and no-force

Restart: analysis, redo, undo

- Analysis: identifies dirtied pool pages and active Xacts at time of crush
- Redo: redo actions to restore db to pre-crush
- Undo: undo actions of Xacts that did not commit

Analysis: Xact table

1. When the first log record is created, create a new entry T with status U
2. Update lastLSN for T to be r's LSN
3. Remove T if end log is seen

Analysis: Dirty Page Table

1. New dirtied page will be added to the DPT with recLSN=r.LSN
2. Remove entry when it is flushed to disk

LOG

	prevLSN	XactID	type	pageID	length	offset	before image	after image
10	-	T_1	update	P500	3	21	ABC	DEF
20	-	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

Redo Phase (DPT)

1. Redo LSN = min(recLSNs), then fetch page LSN,
2. if r.LSN $>$ pageLSN and Page is in DPT, redo
3. Update pageLSN to r.LSN

Undo Phase (TT)

1. Start from largest LSN from L
2. if update, create CLR with undoNextLSN=r's prevLSN, update-L-TT(r.prevLSN)
3. if CLR, update-L-TT(r.undoNextLSN)
4. update-L-TT(lsn): add lsn to L if lsn not null, else add end log record for T and remove it from TT

Checkpointing

- Normal(no ECPLR): CPLR's TT, empty DPT
- Fuzzy: BeginCPLR's TT and BCPLR's DPT