Out-of-Core Algorithms

Map: Read a batch to an input buffer, process it, write to an output buffer. When input buffer consumed, read more data in; when output buffer consumed, write to disk
Rendevous: Can't keep everything in memory, so make sure codependent items appear in memory at the same time (*time-space rendevous*)
2-Way External Mergesort

- Pass 0 (Conquer): Read a page, sort it, write it to disk. Only one buffer page used.
- Pass > 0 (Merge): Requires 3 buffer pages (2 input, one output). Merges pairs of runs into runs twice as long.
- Total cost: $2N(\lceil log_2 N \rceil + 1)$

External Mergesort

- Pass 0 (Conquer): Use B buffer pages and produce $\lceil N/B \rceil$ sorted runs of B pages each
- Pass > 0 (Merge): Merge $B-1$ runs at a time
- # of passes: $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$
- Total cost: $2N * (\#ofpasses)$
- Can sort $B(B-1)$ pages in 2 passes

Internal Sort
Heapsort: Keep two heaps $H1$ and $H2$ in memory, read $B-2$ pages and insert into $H1$. Drain $H1$ with variable m and place new records $< m$ into $H2$ and larger entries into $H1$. When $H2$ fully drained, swap $H1$ into $H2$ and repeat until there are no more records. Average run length is $2(B-2)$.
Hashing
Many operations don't require order, so sorting is unnecessary. Use hashing to rendevous matches. $4 * N$ I/Os.
Divide: Use a hash function to stream records to disk partitions
Conquer: Rehash partitions into RAM partitions using a different hashing function
Recursive Partitioning: If disk partition too big after pass 0, divide partition using a different hash function.

SQL

Inner/Natural join: equi-join for each pair of attributes with the same name
Left outer join: Returns all matched rows and all unmatched rows from table on left
Right outer join: Returns all matched rows and all unmatched rows from table on right
Full outer join: Returns all matched and unmatched rows. Unmatched rows from one side have `NULL` for attributes on the other side.
Integrity Constraints: conditions that every legal instance of a relation must satisfy (e.g. domain/primary key/foreign key/general constraints)
Keys: Primary (at most one), candidate (unique), foreign (referential integrity, no dangling references)

Disk Layout

Page: Tables stored as logical files consisting of *pages* of *records*. Pages in managed in memory by buffer manager, on disk by disk space manager.
LRU: Works well for repeated accesses to popular pages (temporal locality). Costly, needs to maintain heap structure.
MRU: Best for repeated scan of big file (fixes sequential flooding)
Clock Replacement: Arrange frames in cycle with ref bit, "second chance LRU"
Best for DBMS to handle page/buffer management over OS because DBMS requires page pinning and flushing pages to disk, which is important for implement CC and recovery. In addition, the DBMS can adjust the replacement policy and prefetch pages based on access patterns in typical DB operations.

Unordered heap	records placed on arbitrary pages
Clustered heap/Hash	records/pages grouped
Sorted files	pages/records in sorted order
Index files	B+ trees, hash tables

Files: Higher levels of DBMS operate on pages of records and files of pages. File must support insert/modify/delete record, fetching record by record id, scanning all records. Can span multiple machines/OSes.
Page header: Contains number of records, free space, next/last pointer, slot table
Fixed-length records: *Packed*: pack records densely, record record id in page, append on add, rearrange on delete.
Unpacked: header contains bitmap of filled slots; on insert fill first empty slot, on delete clear bit.
Slotted page: Slot directory in header, each slot has length of record and pointer to the beginning of the record; header has number of slots and pointer to free space. Delete sets pointers to `NULL`; insert into first large-enough slot. Grow records from end of page so slot directory can be extended on insert. On insert, extend slot directory, add record in free space, update slot counter in header.
Record formats: Assume system catalog with schema (no need to store type information, format stored in another table). For variable length, record header stores pointers to ends of variable-length fields. Also covers NULL fields, useful for fixed-length.

Join Operators

Cost notation:

$[R]$	# pages to store R
p_R	# records per page of R
$\|R\|$	cardinality of R (# records)

Assume in join R is left, S is right, can be swapped if necessary. Assume we have B pages of memory.
Costs
Simple nested-loop: For each record in R, loop through each record in S and compare. Cost: $(p_R \times [R]) \times [S] + [R]$
Page-oriented nested-loop: Same as before, but for each page in R, loop through each page in S. Cost: $[R] \times [S] + [R]$
Block nested-loop: Same as before, but for each block in R of size $B-2$, loop through each page in S. Cost: $[S] \times \lceil [R]/(B-2) \rceil + [R]$. Use when join condition doesn't filter many rows or one table fits in a small number of memory blocks.
Index nested-loop: For each tuple in R, lookup r_i in index on S. Cost: $[R] + ([R] \times p_R) \times$ lookup time for index on S
Grace Hash join: Partition R and S by hashed join key and write to disk. Use new hash function to build size $B-2$ hash table of elements from smaller table (say R), then load each page from S and probe. Cost: $3([R] + [S])$; N if each table has $\leq B$ pages, $3N$ if $B < n \leq B^2$. serial I/O, so no seek overhead. Use if one table fits into memory, or if we can use hybrid hashing.
Sort-merge: Sort both R and S. Walk through both R and S in sequence and probe. Cost: $Sort(R) + Sort(S) + ([R] + [S])$, last term could be $[R] \times [S]$ worst case. Refinement: do join during final merging pass of sort; Cost: $3[R] + 3[S]$. Especially good choice if one or both inputs sorted on join attributes, output required to be sorted on join attribute, input data is skewed and hash join could lead to recursive partitioning, number of pages in memory is small.
Hybrid hashing: Refinement of Grace hash join to take advantage of more memory. Hold partition 0 in memory instead of writing to disks; saves I/Os.

Tree Indexes

Data structure that enables fast lookup of data entries by search key
Alternatives

A1	By value, entire record in leaf
A2	By reference, (key, rid)
A3	By list of refs, (key, [rid])

Clustered index: Index whose search key specifies the sequential ordering of the file
Unclustered index: Table file not ordered by unclustered index field; range lookups much slower
Multiple indexes: Constructed on different individual columns
Multi-column index: Constructed on concatenated columns

Indexed Sequential Access Method (ISAM)

Static structure!
File creation: Leaf (data) pages stored in order by search key, then index pages, then overflow pages
Search: start at root, use key comparisons to go to leaf
Invariant: With a node $[..., (K_L, P_L), (K_R, P_R), ...]$, all tuples in range $K_L \leq K < K_R$ are in tree P_L
Complexity: $\mathcal{O}(\log_F(\#pages))$
Insert: Find leaf that data entry belongs to and add it. Create overflow page if necessary, and link to leaf page with double pointers. *Delete*: Find entry in leaf and delete it. If deleting tuple empties overflow page, deallocate it and remove from linked list.
Pros: Sequential storage, scan all records without touching index. Good for static data setting with big scans. Cons: Doesn't handle insertion/deletion well, degrades to linear search.

B+ tree

Occupancy invariant: Each interior node is partly full ($d \leq \#entries \leq 2d$), where d is the order of the tree (max fan-out $= 2d + 1$)

Order makes little sense with variable-length entries (e.g. single entry fills entire page, different nodes have different number of entries). Use physical criterion - at least half-full; many real systems only reclaim space when page completely empty (reduces I/Os).

Data pages at bottom not in sequential order, require next/last pointers
Typical order 1600, fill factor 67%. Levels 1 and 2 can usually be stored in buffer pool.

Search: Binary search on each page for node split, follow pointer to next node
Insertion: Search for correct leaf, add entry if page not full, sort if necessary. If not enough space, split leaf and redistribute entries evenly (new entry goes on right side of split). Copy up middle key and recursively split index nodes, pushing up middle key each time
Bulk-Loading: Sort input records by key, fill leaf pages to fill-factor and keep updating parent until full. Split parent by pushing all entries to sibling to achieve fill factor, push one entry to parent to combine node and sibling. Occupancy invariant not held on right branch.
Suffix key compression: Increases fan-out by moving common prefix to header.

Relational Operators

One-to-one relationship between SQL query and relational algebra; SQL is declarative expression of query result, relational algebra is operational description of a computation

Pure relational algebra has *set semantics*, but relaxed for system discussion

Closed: result is also a relational instance, enables composition
Typed: input schema determines output

Unary operators:

$\pi_{cols}(In)$	projection
$\sigma_{condition}(In)$	selection
$\rho(Out(old \to new), In)$	renaming

Binary operators:

$\cup$	`UNION`
$\cap$	`INTERSECT`
$-$	set difference, `EXCEPT`
$\times$	cross product
$\bowtie_\Theta, \bowtie$	join

Monotonicity: If not monotone, operation is blocking. Relational operator Q is monotone in R if

$$R_1 \subseteq R_2 \to Q(R_1, S, T, \ldots) \subseteq Q(R_2, S, T, \ldots) \quad (1)$$

Implementing intersect with set difference:
$S1 \cap S2 = S1 - (S1 - S2)$
Hierarchy: Theta join $\to$ Equi-join $\to$ Natural join

Query Optimization

System R query optimization: Left-deep only, avoid cross-products, push selections/projections down. Interesting orders: ORDER BY, GROUP BY, downstream join attributes
Selectivity (RF) estimation: $\frac{|output|}{|input|}$.

- If col = value, $RF = 1/NKeys(R)$
- If col1 = col2, $RF = 1/max(NKeys(R1), NKeys(R2))$
- If col > value,
  $RF = (High(R) - value)/(High(R) - Low(R) + 1)$

Assumptions: Values are uniformly distributed and terms are independent. If missing required statistics, use 1/10.
Histograms: For better estimation on statistics, use a histogram. Equiwidth, equidepth, v-optimal
Single table cost estimation: Sequential scan: $NPages(r)$.
Equality selection on key of index T: $Height(T) + 1$
Selections on clustered index T:
$(NPages(T) + NPages(R)) \prod RFs$.
Selections on unclustered index T:
$(NPages(T) + NTuples(R)) \prod RFs$.

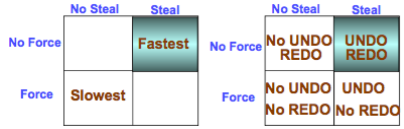
Concurrency

Atomicity: All or none of the actions in xact happen
Consistency: DB must be consistent before/after xact. Expressed as series of integrity constraints (CREATE TABLE/ASSERTION statements)
Isolation: Effects of one xact are isolated from those of others.
Durability: If xact commits, its effects persist.

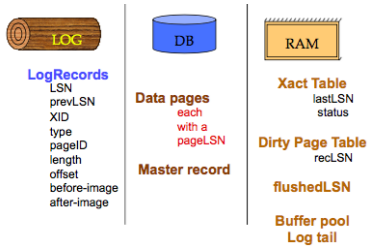
Transaction either commits or aborts (or system crash during xact, treat as abort). Logging: undo actions of aborted/failed xacts, redo actions of committed xacts when system crashes.
Serial schedule: Xact runs from start to finish without intervening actions from other xacts.
Equivalent: Involve some actions of same xacts, leave DB in same state
Conflict equivalent: involve same actions of same transactions, every pair of conflicting actions ordered the same way
View serializability: Same initial reads: T_i reads initial value of A in S_1, then T_i also reads initial value of A in S_2. Same dependent reads: If T_i reads value of A written by T_j in S_1, then T_i also reads value of A written by T_j in S_2. Same winning writes: If T_i writes final value of A in S_1, then T_i also writes final value of A in S_2.
Dependency graph: One node per xact, edge from T_i to T_j if an earlier operation of T_i conflicts with an operation of T_j. Conflict-serializable iff dependency graph is acyclic
Waits-for graph: Edge from T_i to T_j means that T_j is holding a resource that T_i needs, and T_i is waiting on T_j. Periodically check for cycles in the graph and kill a transaction in the cycle.
Two-phase locking: xact must get a shared lock before reading, exclusive lock before writing. Cannot get new locks after releasing any locks. Does not prevent cascading aborts.
Strict 2PL: Locks released only when xact completes (commit or abort). Prevents cascading aborts.

Lock manager: handles lock/release requests. Keeps an entry for each key: (holders, waiters, shared/exclusive).
Deadlock: Deal with by prevention, avoidance, detection, use timeout.
Deadlock avoidance: Wait-die: If T_i higher priority, T_i waits for T_j else T_i aborts. Wound-wait: If T_i higher priority, T_j aborts else T_i waits. Priority usually based on transaction's age. Make sure if transaction restarts, it gets its old timestamp.
Granularity: database $\to$ tables $\to$ pages $\to$ tuples
Intent locks: IS, IX, SIX. To get S or IS, must hold IS or IX on parent. To get X or IX or SIX, mut hold IX or SIX on parent. Must release locks in bottom-up order.

	IS	IX	SIX	S	X
IS	√	√	√	√	-
IX	√	√	-	-	-
SIX	√	-	-	-	-
S	√	-	-	√	-
X	-	-	-	-	-



Recovery



Recovery manager: ensures atomicity and durability (also consistency-related rollbacks)
Force policy: make sure every update on DB disk before committing, slow. No-force: allow commits without flushing dirty pages to disk
No steal policy: Don't allow buffer-pool frames with uncommitted updates to overwrite committed data on disk, slow. Steal policy: system can evict/flush pages with uncommitted udpates.
Logging: REDO/UNDO action pairs. (XID, pageID, offset, length, old data, new data). Possible log record types: update, commit, abort, checkpoint (for log maintenance), compensation log records (CLR) for UNDO, END (commit or abort).
Log sequence number (LSN): always increasing. Each data page contains a pageLSN, LSN of most recent log record for update to that page. System keeps track of flushedLSN, max LSN flushed so far.
Write-ahead logging: Force the log record for an update before the corresponding data page gets to disk. Force all log records for xact before commit. Before page i is written to DB, $pageLSN_i < flushedLSN$.
Transaction table: one entry per xact: (XID, status (run/commit/abort), lastLSN (most recent LSN written by

xact)).
Dirty page table: one entry per dirty page in buffer pool. Contains recLSN (LSN of log record which first caused page to become dirty).
Transaction abort: Get lastLSN of xact from xact table. Write new abort log record at end of log before beginning rollback. Follow chain of log records backwards via prevLSN. Write CLR for each undone operation.

Crash Recovery

Analysis: Start at last checkpoint. Re-establish knowledge of state at checkpoint via transaction table and dirty page table stored in checkpoint. Scan log forward from checkpoint. End record: remove xact from xact table 9done). Other records add xact to xact table, set lastLSN = LSN, change xact status on commit. For update records: if P not in dirty page table, add P to DPT, set recLSN = LSN. recLSN is first LSN that caused page to become dirty. At end of analysis, xact table says which xacts were active at last log flush before crash. DPT says which dirty pages might not have made it to disk.
Redo: Repeat history to reconstruct state at crash. Reapply all updates (even of aborted xacts), redo CLRs. Scan forward from log rec containing smallest recLSN in DPT. For each update log/CLR with given LSN, redo unless: (Affected page not in DPT, affected page in DPT but $recLSN > LSN$, $pageLSN(inDB) \geq LSN$ (slow!)). For each redo action, reapply logged action and set pageLSN to LSN. No additional logging/forcing. At end of redo phase, end type records are written for all transactions with committed status.
Undo: Effects of failed xacts. Keep track of toUndo set: lastLSNs of all xacts in xact table. Repeat until toUndo is empty: Pop largest LSN among toUndo. If LSN is CLR and undoNextLSN = NULL: write END record for xact. If LSN is CLR and undoNextLSN $\neq$ NULL: Add undoNextLSN to toUndo. Else: LSN is an update. Undo update, write CLR, add prevLSN to toUndo. Lets us do one backwards pass of the log.

Distributed Data

2-phase commit:

1. Coordinator tells participants to prepare. Participants respond with yes/no votes. Must be unanimous to procede.
2. Coordinator broadcasts either commit/abort depending on vote. Participants ack.