DATA3404 Scalable Data Management

UNIVERSITY OF SYDNEY · SCHOOL OF COMPUTER SCIENCE

EXAM REVISION
Sem 1 2026 · **SIDE 1 OF 2**
Whole-unit revision · all topics

SIDE 1/2 SINGLE-NODE ENGINE · Storage & buffer · Slotted page · Heap vs sorted cost · Row vs column · B+-tree · Hashing · External sort · The JOIN cost models

REVISION SHEET · ALL TOPICS

Compiled by **AskSia** · mapped to the DATA3404 syllabus · asksia.ai/cheatsheet/data3404

0 · Exam Blueprint READ FIRST

★ DATA3404 is about database-engine internals + scaling out — "learn the principles, not the software." The graded weight: **Final 60%** (paper-based, 2h, semi-open book [confirm permitted materials]) + 2×12% group practicals (SimpleDB engine; SQL + Spark) + 6% quizzes + 5% participation + 5% presentation.

Double hurdle: you must score ≥40% on the exam AND ≥50% overall — fail either and the final mark is capped at 45.

The exam costs engine internals, NOT SQL: it is mostly short-answer (some MCQ) and explicitly *does not assess your code*. Marks live in hand-worked cost models: buffer/CLOCK, B+-tree & hash math, external-sort passes, the join cost formulas, and query-optimisation costing.

SIA → *Every cost is in page transfers (I/Os), output ignored, CPU & seq-vs-random ignored. Write the formula, sub the numbers, box the answer – method marks survive an arithmetic slip.*

1 · Storage & Files LEC 02

DB file = a sequence of **fixed-size pages**; each page holds many records, aligned to block-oriented disk. The **access gap**: disk latency ≫ memory ⇒ minimise page I/O.

FILE ORGANISATIONS

- **Heap** — record placed anywhere with free space; best when access is a full scan.
- **Sorted** — ordered by search key; binary search, good for range/ordered retrieval; insert shifts records (expensive).
- **Index** — tree/hash access path; faster updates than a sorted file.

ACCESS COST (B = #PAGES)

OP	HEAP	SORTED
Scan all	O(B)	O(B)
Equality	**B/2** avg	**$\log_2$B**
Insert	2 (rd+wr last)	$\log_2$B + B
Delete	B/2 + 1	$\log_2$B + 1

Query types: exact-match / point (key ⇒ 0–1 row) vs **range** (BETWEEN, LIKE 'x%' ⇒ many).

Raw flat files (CSV/JSON) = a linear heap: full scan to find a row, inserts shift everything, no index. Hence pages + indexes. Sorted files are rarely used directly — instead use an index-organised (clustered) file.

1b · Records & Slotted Page VAR-LENGTH

Variable-length record best format = an **array of field offsets** in the header (direct i-th field access, cheap NULLs). **NULL-bitmap**: 1 bit/field, 1 byte covers 8.

Slotted page = Page Header + **Slot Directory** (grows one way) + Records (grow the other). Slot entry = (offset, length). **TID = ⟨page id, slot #⟩**; a record can move within a page without changing its TID (the slot still points to it).

Big values: overflow or out-of-line storage (PostgreSQL *TOAST*: ~2 KiB chunks, varlena pointer, compressed). A record must fit a page (≈ 8 KiB − header).

Fixed-length record = array of same-size records (struggles with var strings / NULLs); **variable-length** uses delimiters, length-prefixes, or the offset array.

PostgreSQL row = RowHeader + NULL-bitmap (⌈(#cols+7)/8⌉ B, 0 if all NOT NULL) + RowData (fixed then var cols). Page slot = 4 B ItemId (offset+length).

2 · Row vs Column Store OLTP VS OLAP

	ROW STORE	COLUMN STORE
Layout	all attrs together	one file/attr
Good for	OLTP, full-row	**OLAP, few cols**
I/O	reads whole row	reads only needed cols
Compress	mixed types	**better (uniform)**
Update	cheap	expensive

PAX (Partition Attributes Across) = hybrid: rows in a page, page split into *n minipages*, one per attr ⇒ better cache locality. **Parquet** = open column format (nested data, à la Dremel). **Wide-column** (BigTable, HBase) ≠ column DBMS: column families stored row-wise, for sparse data.

Column DBMS: MonetDB, Vertica, SAP HANA, Druid. Column-store cons: **updates & full-row reads are expensive**; bad for small tables.

3 · Buffer Management LEC 02

Buffer manager loads pages into **frames**. On a request: in buffer? return it; else pick an unpinned frame, if **dirty write it back**, then load. **Pin count** > 0 ⇒ not evictable; requestor must **unpin** & flag modification. Dirty pages written deferred by a background writer.

READ EFFICIENCY (HIT RATIO)
```
(req - physical I/O) / req · aim ≥ 80%
```

Why DBMS not OS: needs to pin, force-to-disk for recovery, tune replacement, prefetch, avoid double-buffering.

3b · Replacement Policies MEMORISE CLOCK

POLICY	EVICTS
FIFO	oldest arrival (by age)
LRU	least-recently-used (locality)
MRU	most-recently-used
LFU	smallest use count
CLOCK	"second chance" (below)
GCLOCK	CLOCK + ref counter

CLOCK = circular list + hand; each frame has a **reference bit** (set 1 on access). On eviction: if R=1 clear it to 0 and advance; if R=0 (unpinned) evict. **GCLOCK** uses a counter decremented each sweep, evict at 0 ⇒ protects hot pages. Most DBMS use a CLOCK variant.

SIA → *In a CLOCK/GCLOCK trace, hits change state too (set the ref bit / bump the counter) – count those or your final buffer is wrong.*

3c · Trace Method HOW TO SCORE IT

Tabulate the reference string left→right with one column per access; mark **hit** (in buffer) or **miss** (disk I/O). On a miss with a full buffer apply the policy to choose the victim. Count **misses = disk I/Os** (cold-start misses included).

FIFO ignores re-use (by age); **LRU/MRU** reorder a recency queue on every access; **LFU/GCLOCK** track counts since load. GCLOCK typically yields fewer misses than CLOCK because its counter protects hot pages.

Classify policies by **age** (FIFO, since arrival) vs **references** (LRU/MRU/CLOCK by last reference; LFU/GCLOCK by all references).

4 · B+-Tree LEC 03 · EVERY YEAR

Dynamic, balanced, multi-level. All root→leaf paths equal length; entries only in leaves; double-linked **sibling pointers** between leaves for range scans. Interior nodes = separators only.

Layout P_0 K_1 P_1 … K_n P_n: go P_0 if k<K_1; P_n if k≥K_n; else P_i for K_i≤k<K_{i+1}. Each node (order d): leaf holds d–2d entries; non-root ≥ half full; **fan-out F** = avg children (often >100).

SEARCH COST (ONE MATCH)
```
≈ ⌈log_F N⌉ + 1 (N = #leaf pages)
= index height + 1
```

In practice order 100, fill 66% ⇒ F≈133. Height 3 ⇒ 133^3 ≈ 2.35M rows; height 4 ⇒ ≈313M. Height rarely > 3–4.

INSERT

Find leaf; if room insert sorted; else **split leaf** (keep first ⌈n/2⌉, copy up new node's min key); if parent full **split index node** (remove middle k_m, push it up). Root split ⇒ height +1.

DELETE

Remove; if ≥ half full done; else **redistribute** (borrow from sibling) else **merge** + drop separator. Many systems skip rebalance on delete.

4b · Worked · Height RE-NUMBERED

Table = **800,000 records**, unclustered B+-tree order 50 (≤101 children, leaf ≤100 entries).

```
#leaf pages = 800,000 / 100 = 8,000
height = ⌈log₁₀₁ 8,000⌉ = 2 ⇒ 3 levels
search 1 match = h + 1 = 3 page reads
```

Top levels are usually cached (L1=1, L2=F, L3=F^2 pages) ⇒ root + interior rarely cost real I/O.

ISAM = the static cousin: sparse separators over clustered leaves, but inserts spawn **overflow chains** ⇒ degrades for dynamic data.

4c · Storage Hierarchy LEC 01

Registers → CPU cache → main memory → **disk** → tertiary. The access gap: secondary-storage latency ≫ memory latency, so data moves in **page** units.

Units trap: SI MB = 10^6 B; IEC MiB = 1024^2 = 1,048,576 B; KiB=1024, GiB=1024^3. OSes report binary but label "MB"; drive makers use decimal.

Prefetching reads several pages ahead on predicted sequential access (e.g. DB2 8/32 pages).

4d · Split Rules · Order 2 LEAF VS NODE

Order d=2 ⇒ node holds 2d=4 entries. Insert into a full leaf {40,40,42,45,50}:

LEAF SPLIT (COPY UP)
```
keep first ⌈5/2⌉=3 = {40,40,42},
new leaf {45,50}, copy up 45 to parent
```

INDEX-NODE SPLIT (PUSH UP)
```
root {18,40,45,73,91} full ⇒ keep {18,40},
new node {73,91}, push up middle 45
as new root ⇒ height +1
```

Key difference: leaf split **copies** the separator up (it still lives in a leaf); index split **moves** it up (removed from the node).

Range search: descend to the first matching leaf, then follow sibling pointers along the leaf level to the end of the range — no re-descending.

5 · Hash Indexing LEC 04

Entries → **buckets** by h(v); address = h(v) mod M. Equality cost = #pages in bucket = 1 if no overflow ⇒ beats a B+-tree for equality. **No range search.**

Static hashing: fixed M, overflow chains grow long (like ISAM). Fixed by dynamic schemes.

EXTENDIBLE HASHING

Directory of size 2^(global depth); pick bucket by the last global-depth bits of h(r). **Local depth** = #bits deciding one bucket; **global depth** = max over all.

Insert: room ⇒ insert. Full ⇒ **split bucket** (rehash on 1 extra bit, both local depths +1).

DIRECTORY-DOUBLING RULE
```
double IFF local depth = global depth
(before split). If local < global → just
re-point one directory entry, NO doubling.
```

Equality = 1 disk access if directory in memory (else 2). Delete: empty bucket merges with split image; halve directory if all entries equal their image.

SIA → *Track local vs global depth on every split – that single comparison decides whether the directory doubles. Skewed hashes & duplicate keys bloat it.*

5b · Worked · Extendible Build RE-NUMBERED

Insert 4,6,10,14 then 22,34,38 with h(v)=v mod 8, bucket capacity 3, start 1 bucket.

- 4,6,10 fill bucket (global=local 0).
- Insert **14** ⇒ full & local=global ⇒ split + double directory to size 2, depth 1.
- Insert **22** ⇒ a bucket's local depth → 2 > global(1) ⇒ double again to size 4, depth 2.
- 34,38 then fit by re-pointing (local < global) — **no doubling**.

5c · Bitmap & Bloom OLAP

Bitmap index: for field with m distinct values, m bit-vectors of length n (#rows); vector v has 1 where row=v. Answer queries with AND/OR/NOT/COUNT bit-ops. Space O(m·n) ⇒ compress (RLE) when m large. Best for **few distinct values**.

e.g. males with rating<4 = (R1 OR R2 OR R3) AND Gender_M, then COUNT the result vector.

Bloom filter = probabilistic membership: no false negatives, possible false positives ("definitely not" / "possibly in"). Element hashed to several bits in a small array.

5d · Hash vs B+-Tree PICK THE INDEX

NEED	USE
Equality only	**hash** (1 page/bucket)
Range / ordered	**B+-tree** (hash can't)
Prefix of composite	B+-tree
Group-by / sort	clustered B+-tree

Extendible-hash equality = 1 disk access if the directory fits in memory (else 2). 1M rows / 4K-row pages ⇒ ~25,000 directory entries ⇒ fits.

Linear hashing = directory-free dynamic scheme (round-robin bucket splits); both fix **static hashing's** overflow chains.

Mini-trace (cap-2, h=v mod 8): 0,4 fill a bucket; insert **1** ⇒ local=global ⇒ split + double (g=1) ⇒ {0,4}/{1}; insert **3** after 5 ⇒ local=global=1 ⇒ double again (g=2) ⇒ {1,5}/{3}.

6 · Index Classification 4 DIMENSIONS

DIMENSION	MEANING
Primary/secondary	key sets file order (often integrated) vs not
Unique/non-unique	over a candidate key or not
Clustered/unclustered	data & index ordered same; ≤1 clustered/table
Dense/sparse	entry per record vs per page

Rules: a main/integrated index is always clustered; unclustered must be dense; sparse needs a candidate-key search key. *Search key ≠ key* (search key need not be unique).

EGSR choice rule = Equality, GroupBy, Sort, Range. Equality attrs first (any type); range ⇒ tree index; group-by ⇒ index on those attrs; sort ⇒ clustered B-tree.

Composite (A1,A2) = lexical; supports prefix-key & **index-only (covering)** plans. Order: equality keys before range keys.

Worthwhile if lookup + retrieval < full-scan cost.

6b · Range-Scan Cost FAVOURITE

10,000 data pages, 100 rows in range, 20 rows/page:

ACCESS PATH	PAGE I/O
Heap	10,000 (whole file)
Sorted on key	$\log_2$10000 + 5 ≈ 19
Unclustered idx	(h+1) + 100 (1 I/O / row)
Clustered idx	(h+1) + 5 (rows packed)

The unclustered penalty = **one random I/O per matching row**; clustered packs matches on consecutive pages.

6c · Multi-Attr Index Match PREFIX RULE

Selection `salary<50000 ∧ age=21 ∧ make='x'`.

Match rule: all predicate attrs must be a prefix of the key, AND any range (inequality) attr must be **last**.

INDEX KEY	MATCHES?
(salary,age,make)	✕ range not last
(age,salary,make)	✕ range not last
(age,make,salary)	✓ prefix + range last

Hash index matches only on equality over *every* key attr; a tree index matches a prefix (e.g. `<a,b,c>` matches a=5 ∧ b>3, not b=3 alone).

6d · Worked · File Scan HEAP VS SORTED

Person: 1,000,000 rows, 10/page ⇒ **100,000 pages**; age uniform 0–99.

- age=30 on heap (1% match) ⇒ still scan all 100,000 (unsorted).
- Sorted on *name* ⇒ no help for an age predicate ⇒ 100,000.
- Sorted on *age* ⇒ binary search + clustered ⇒ ~1,000 pages (range [30,35) ⇒ ~5,000).

Lesson: an index/order only helps if it matches the *query's* predicate attribute — a name-sorted file is useless for an age query.

6e · Covering Index INDEX-ONLY

An **index-only (covering) plan** answers a query from the index alone, never fetching the heap tuple. e.g. `(city,name)` answers `SELECT name WHERE city=:v`. Clustering not needed for index-only plans — usually a tree index.

7 · External Merge Sort LEC 05

For ORDER BY, DISTINCT, sort-merge join, bulk B+-tree load. Sort N pages with B buffer frames:

Pass 0 (sort): read all N (B at a time), sort each chunk, write ⇒ ⌈N/B⌉ runs of B pages. Cost 2N.

Merge passes: merge **B−1** runs at a time (1 output buffer); each pass reads+writes all N ⇒ 2N.

#PASSES & TOTAL I/O
```
#passes = 1 + ⌈log_(B-1) ⌈N/B⌉⌉
Total I/O = 2N · (#passes)
```

Run length after merge pass i = $(B-1)^i$·B. In practice rarely > 2–3 passes. **Clustered** B+-tree on the sort col ⇒ read leaves in order (good); **unclustered** ⇒ random I/O/record (bad).

7b · Worked · Sort COUNT THE PASSES

N = **10,000 pages**, B = 30:

```
Pass 0 runs = ⌈10000/30⌉ = 334
merge B-1 = 29 at a time
P1: ⌈334/29⌉ = 12 runs
P2: ⌈12/29⌉ = 1 ⇒ 2 merge passes
total passes = 1 + 2 = 3
Total I/O = 2·10000·3 = 60,000
```

SIA → *Don't forget the +1 sort pass: #passes = 1 + (merge passes). Forgetting it is the classic off-by-one that drops the I/O total.*

8 · Query Pipeline LEC 05

SQL (declarative) → **Parser → Optimizer → Executor**. SQL → relational-algebra logical tree → physical plan (operators + algorithm). Many plans per SQL; optimiser picks one.

Materialization (set-at-a-time): write temp relation, next op reads it — always works, costly I/O. **Pipelining** (tuple-at-a-time): pass each row on — cheap, but the inner join input, sorts, hash joins & aggregations *can't* pipeline their input.

Reduction factor = #out / #in (= selectivity). Cost = #block transfers, output ignored.

8b · Relational Algebra THE OPERATORS

6 basic: ∪ union, − difference, σ select (rows), π project (cols), × cross-product, ρ rename. **Derived**: ∩, ⋈ join, ÷ division. Set ops need union-compatible schemas (same arity, names, domains).

Join R⋈_θ S = σ_θ(R×S); **equi-join** = only equalities; **natural join** = equi-join on all same-named attrs, keep one copy. Complex conditions → CNF to match access paths.

Access paths: table scan, scan+filter, index scan, index-only (covering) scan.

σ (selection) = subset of rows; π (projection) = subset of columns (strict RA removes duplicates ⇒ SQL DISTINCT). σ and π are usually fused into the access-path loop.

An **expression tree** shows logical RA operators; a **query execution plan** shows the physical operators + algorithms chosen.

Formula Belt SIDE 1

```
heap eq = B/2 · sorted eq = log₂B
B+tree search = ⌈log_F N⌉+1 = h+1
ext-sort #passes = 1+⌈log_(B-1)⌈N/B⌉⌉
total I/O = 2N·#passes
ext-hash: double iff local=global depth
hit ratio = (req-I/O)/req ≥ 80%
```


flip → for side 2 · joins, optimisation & scaling out

DATA3404 Scalable Data Management

UNIVERSITY OF SYDNEY · SCHOOL OF COMPUTER SCIENCE

EXAM REVISION
Sem 1 2026 · **SIDE 2 OF 2**
Joins · optimisation · scaling out

SIDE 2/2 THE JOIN COST MODELS · Query optimisation · Selectivity · Shared-nothing · Partitioning · Distributed joins · HDFS · MapReduce · Spark · NoSQL

REVISION SHEET · ALL TOPICS

Compiled by **AskSia** · mapped to the DATA3404 syllabus · asksia.ai/cheatsheet/data3404

9 · THE JOIN COST MODELS

LEC 06 · EXAM GOLD

R = outer (b_R pages, |R| rows), S = inner (b_S pages). Cost in page I/O, output ignored. Always make the smaller relation the outer.

JOIN	COST (I/O)
Simple NLJ	b_R + |R|·b_S
Page NLJ	b_R + b_R·b_S
Block NLJ	b_R + ⌈b_R/(M−2)⌉·b_S
Index NLJ	b_R + |R|·c
Sort-merge	sort(R)+sort(S)+b_R+b_S
Grace hash	3(b_R + b_S)

INLJ: c = cost of one selection on S = 1 + h + #matches (root + index height + matching rows). **SMJ sorted-only** = b_R + b_S; result is sorted. **Hash & SMJ work only for equi/natural joins** (not outer/inequality).

SIA → *BNLJ uses M−2 (1 input + 1 output page reserved) in the worksheet form – the M−1 variant is for pipelined output. Read which the question wants; the block size drives the whole answer.*

9b · Worked · NLJ vs BNLJ

RE-NUMBERED

R = 100 pg / 1000 rows; S = 400 pg / 10,000 rows; M = 10 (block M−2 = 8).

SIMPLE NLJ

```
R outer: 100 + 1000·400 = 400,100 ✓
S outer: 400 + 10000·100 = 1,000,400
```

BLOCK NLJ

```
R outer: 100 + ⌈100/8⌉·400 =
100+13·400 = 5,300 ✓
S outer: 400 + ⌈400/8⌉·100 = 5,400
```

BNLJ is ~75× cheaper than simple NLJ here — and smaller-as-outer still wins.

9d · Join Types & Stats

SETUP THE COST

θ-join (any cond), **equi-join** (=only), **natural** (all same-named), **outer** (NULL on no match). Hash & sort-merge work *only* for equi/natural.

For R(1000 rows)⋈S(10,000) on a FK: cross-product = 10,000,000 rows; join result = 10,000; avg matches/R-row = 10. The σ_θ(R×S) view would discard 9,990,000 — why **physical join algorithms** beat materialising the cross-product.

Joins matter: ~25% of CPU on a TPC-H workload (Impala) — hash join + sequential scan dominate analytics. That is why the exam drills the cost models.

9e · Which Join When

DECISION

- **No index, any condition** ⇒ block NLJ (smaller as outer block).
- **Index on inner join attr, equi** ⇒ index NLJ if few matches.
- **Both large, equi** ⇒ grace hash (or sort-merge if output should be sorted).
- **One pre-sorted on the key** ⇒ sort-merge (skip a sort).

Output is ignored in cost, but a sort-merge join **leaves the result sorted** — free for a later ORDER BY or another merge join (an "interesting order").

- **Inequality** (R.a<S.b) or **outer** ⇒ hash/SMJ not applicable ⇒ block NL (or INLJ on a clustered B+-tree).
- **Multi-attribute equi-join** ⇒ INLJ indexes the *combination*; SMJ/hash sort/partition on it.

9c · Worked · INLJ, SMJ, Hash

SAME R,S

R = 100 pg/1000 rows, S = 400 pg/10,000 rows, M = 10.

INDEX NLJ — C = 1 + H + MATCHES

```
(a) idx on R, h=1, unique ⇒ c=3;
S outer, R inner: 400 + 10000·3 =
30,400
(b) idx on S, h=2, ~10 matches ⇒
c=13;
R outer, S inner: 100 + 1000·13 =
13,100 ✓
```

SORT-MERGE (BOTH UNSORTED, M=10)

```
sort R = 2·100·3 = 600
sort S = 2·400·3 = 2400
merge = b_R+b_S = 500
total = 3,500 (2,900 if R pre-
sorted)
```

GRACE HASH

```
3(b_R+b_S) = 3(100+400) = 1,500
```

Ranking (best→worst): hash 1,500 < SMJ 2,900–3,500 < BNLJ 5,300 < INLJ 13,100 ≪ simple NLJ 400,100.

10 · Join Internals

HOW EACH RUNS

Grace hash: partition both by h on the join key (choose n so b_R/n fits buffer), then build hash table on each R-partition + probe matching S-partition. Partition = 2(b_R+b_S), match = (b_R+b_S) ⇒ **3(b_R+b_S)**.

Hash vs merge: with enough memory both = 3(b_R+b_S). Hash wins when sizes differ greatly & is highly parallel; merge is skew-robust & outputs sorted.

Radix join = cache-optimised hash: multi-pass partition to cache-sized chunks (histogram → offsets → partition), p = ⌈log(|R|/cache)⌉.

Inequality join (R.a<S.b): hash & SMJ *not* applicable ⇒ **block NL often best**.

10b · DISTINCT, GROUP BY, Set Ops

SORT OR HASH

Implement by **sorting** (duplicates become adjacent) or **hashing** (duplicates land in one bucket):

- **DISTINCT / duplicate removal** — sort then drop neighbours, or hash.
- **Aggregation + GROUP BY** — sort/hash on the group-by attrs; index-only scan if a covering tree index exists.
- **Set ops** (∪, ∩, −) — same machinery; need union-compatible schemas.

Multi-attribute equi-join: SMJ/hash **sort/partition on the combination**; INLJ builds an index on the combination.

10c · MapReduce vs Spark

WHY SPARK

	MAPREDUCE	SPARK
Intermediates	**disk**	memory (RDD)
Model	map+reduce only	rich operators, DAG
Iterative	slow (re-read)	**fast (cache)**
Fault tol.	re-run task	lineage rebuild

Both target **shared-nothing**; both materialise/shuffle by key for grouping. Spark adds lazy DAGs + in-memory reuse for interactive/iterative work.

Spark join strategies (4): broadcast (small side under a threshold), shuffle sort-merge (default), shuffle hash, shuffle-replicate-NL (generic). Set with join hints; AQE re-optimises at runtime. These map 1:1 to the distributed-join approaches in §14.

11 · Query Optimisation

LEC 07

Cost-based: (1) generate equivalent expressions; (2) annotate with physical operators; (3) pick cheapest estimated plan. I/O dominates (I/O ≫ CPU ≫ network). Estimate both op cost & result size; assume **uniform distribution + predicate independence** (often wrong). PostgreSQL ANALYZE samples stats into pg_stats.

RA EQUIVALENCES — APPLY THESE

- **Selections cascade & commute**: σ_{c1∧c2} = σ_c1(σ_c2(·)); push down before joins.
- **Projections cascade**; drop unneeded columns early.
- **Join = σ over ×**: R⋈_θ S = σ_θ(R×S).
- **Joins commute & associate**: reorder freely.

Heuristics: selection early, projection early, most-restrictive selections/joins first.

11b · Selectivity

ESTIMATE SIZES

EQUALITY ON ATTR, V DISTINCT VALUES

```
selectivity ≈ 1/V
est. matches = |R| / V
reduction factors multiply
(independence)
```

e.g. pod has 2500 distinct values ⇒ σ(pod=7) keeps |R|/2500 rows. Pushing this selection before a join is the single biggest cost lever.

Worked: R=40,000 rows, city=200 distinct, paid=2; σ(city='x') keeps 40,000/200 = 200; ∧ paid=T ⇒ RFs multiply ⇒ 40,000/(200·2) = 100. Range ⇒ RF = (hi−lo)/(max−min).

11c · Left-Deep · System-R

DP SEARCH

Left-deep tree: every join's right input is a *base relation* ⇒ fully pipelined (no temp files). System-R considers only left-deep trees.

Dynamic programming (N passes): Pass1 = best access path/relation; Pass2 = best plan per pair; … PassN joins the (N−1)-plan with the last relation. Keep the cheapest plan *plus* the cheapest per **interesting order** (a sort order useful to a later SMJ/ORDER BY).

Search space is huge: 6 relations ⇒ 40,340 bushy trees; left-deep prunes it.

11e · Worked · Push-Down

RE-NUMBERED

Query: `name,site FROM Emp e, Dept d, Site b WHERE e.dept=d.id ∧ d.site=b.id ∧ e.sal>10K`.

- E1: cross-products + one big σ (ρ-rename the two ids).
- E2: rewrite × into joins.
- E3: push σ_(sal>10K) onto Emp before the joins.

Left-deep result: π((σ(Emp)⋈Dept) ⋈ Site). View expansion is handled by the **rewriter** (after parse, before planner).

11f · Estimation Pitfalls

STATS LIE

The optimiser assumes **uniformity + independence** — both often wrong, so cardinality estimates can be far off and a re-sample can flip the plan. Adding a B+-tree on a join key may not change the plan if stats say a scan is cheaper.

Nested queries: optimise each block with the outer tuple as a selection; **un-nesting** (rewrite to a join) usually optimises better. ORDER BY / LIMIT applied last miss optimisation — push LIMIT onto the base relation.

11d · Worked · Cost Chain

RE-NUMBERED

Vehicle(vid,depot): 10k rows, 125 pg, 2500 distinct depot. Trip(tid,vid): 50k rows, 500 pg. Join on vid, filter depot=7.

PLAN	I/O
NLJ no index, no push (page form)	125+125·500 = **62,625**
Push σ(depot=7) (~4 survive)	125+4·500 = **2,125**
INLJ clustered idx Veh.vid (Trip outer)	500+50000·2 = **100,500**
Unclustered idx pair	6+4·8 = 38 ✓

Pushing the selection + the right index pair cut cost from **62,625 to 38** — orders of magnitude. (Note: INLJ with a clustered index here is *worse* — 50k lookups.)

SIA → *More indexes ≠ cheaper. Cost every plan: a clustered-index INLJ can lose to a pushed-selection scan when the outer is large.*

12 · Parallel Architectures

LEC 08

TYPE	TRAIT
Shared-memory	fast; bus limits ~32–64 procs
Shared-disk	fault-tolerant; disk interconnect bottleneck
Shared-nothing	scales to 1000s; comms cost

Shared-nothing = each node owns CPU+memory+disk, network only ⇒ big-data state of the practice.

SCALABILITY GOALS

```
Speed-up = more resources, fixed data
⇒ less time
Scale-up = resources ∝ data ⇒ time
constant
scale-out (horizontal) vs scale-up
(bigger box)
```

12b · Failures at Scale

LEC 08

COMBINED MTBF

```
1 / Σ(1/MTBFᵢ) · for k identical =
MTBF₁/k
```

NODES K	MTBF
1	750,000 h
100	7,500 h
1000	750 h

⇒ at scale failures are **guaranteed** ⇒ replication + fault tolerance are mandatory. The **8 fallacies** (network reliable, latency zero, bandwidth infinite…) are all false.

12c · Replication Design

READS ≫ WRITES

AXIS	OPTIONS
When	**eager** (consistent, slow) / **lazy** (fast, eventual)
Where	primary-copy (master) / update-anywhere
Scope	total / partial

Practice = lazy + primary-copy (log shipping); ideal theory = eager + update-anywhere. Read any copy. MongoDB: async primary-copy + a *Read Preference* for stale secondaries.

Write propagation usually = log-based capture (log shipping). **Consistency**: eager ⇒ 1-copy-serialisable (1-SR); lazy ⇒ eventually consistent (CAP "forfeit C": Dynamo).

13 · Partitioning & CAP

PHYSICAL DESIGN

Horizontal = subset of rows (a *shard* across sites = sharding); **vertical** = subset of columns.

SCHEME	PRUNES FOR
Round-robin	nothing (intra-query par. only)
Hash	equality predicates
Range	range predicates

Co-partitioning: partition two join tables on the same key + method ⇒ co-located local joins, no reshuffle (equi-joins only). **Replication**: read any copy, updates slower ⇒ best when reads ≫ writes; eager (consistent) vs lazy (fast, eventual).

CAP THEOREM

```
Consistency + Availability +
Partition-tolerance:
pick at most 2 of 3
```

At scale failures are guaranteed: k nodes ⇒ MTBF = MTBF₁/k.

14 · Distributed Joins

LEC 09

- **Local-reference** — replicate small S everywhere; R partitioned ⇒ local joins.
- **Broadcast** — ship a selective σ(S) to all R-nodes (small data moved).
- **Shuffle (partitioned)** — re-hash BOTH on the join key, same scheme, then local joins; equi/natural only.
- **Fragment-and-replicate** — m·n grid; works for *any* condition (incl. non-equi).

HDFS: 1 NameNode (metadata) + many DataNodes (blocks). Files split into big **blocks (default 64 MB)**, each replicated 3×. NameNode involved only at start ⇒ not a data-path bottleneck; read from nearest replica.

14b · Worked · Distributed Join

RE-NUMBERED

9 nodes; R = 45,000 pg (hash-partitioned ⇒ 5,000 pg/node); S = 450 pg; local buffer M=46. Re-apply BNLJ **per node**:

METHOD	I/O / NODE
Broadcast σ(S) (~5 pg)	5+1·5000 = 5,005
Shuffle, local hash (50-pg S)	50+5000 = 5,050
Local-reference (S replicated)	450+10·5000 = 50,450

vs a centralised BNLJ ≈ 4,545,000 I/O ⇒ distributing is **orders cheaper**. Method = re-run the same join formula on each node's partition, then divide work across nodes.

14c · Parallelism Types

WHERE SPEED COMES FROM

- **Inter-query** — different queries on different sites independently.
- **Intra-query** — one query spans several sites in parallel.
- **Intra-operator** — one operator split across nodes.

Round-robin gives only intra-query parallelism (no pruning); hash/range also **prune** by matching predicate. Distributed evaluation = materialization (MapReduce) vs pipelining (exchange-operator, Flink).

The grand prize: **co-partition** two join tables on the same key so the shuffle disappears entirely (local joins only).

Cost split = data-movement + parallel local join. Broadcast moves least (small filtered side); shuffle moves each tuple once; fragment-and-replicate moves most but handles any condition.

15 · MapReduce & Spark

LEC 09–10

MapReduce phases: Map → Shuffle & sort → Reduce. map(k,v)→list⟨k',v'⟩; reduce(k',list⟨v'⟩)→out. WordCount: map emits (word,1), reduce sums. Materialises intermediates to local disk; **data locality** (run map on the block's node). ≈ a parallel `SELECT agg GROUP BY`. Weak: scans whole input, joins need code.

APACHE SPARK

RDD = immutable, partitioned, in-memory; fault-tolerant via **lineage** (rebuild lost partition). **Transformations** (map, filter, join, reduceByKey) are lazy — recorded as a **DAG**; **actions** (count, collect, reduce, save) trigger execution. `cache()` reuses an RDD.

DEPENDENCY	EXAMPLES
Narrow (pipeline)	map, filter
Wide (shuffle)	groupByKey, join, reduceByKey

A wide dependency = a stage boundary in the DAG. DataFrame API compiles to RDDs via the **Catalyst** optimiser (3.0 = Adaptive Query Execution). Join strategies: broadcast, shuffle sort-merge (default), shuffle hash, shuffle-replicate-NL.

16 · NoSQL Families

LEC 12

FAMILY	EXAMPLE
Key-value	Redis, Dynamo, RocksDB
Wide-column	BigTable, HBase
Document	MongoDB, CouchDB
Graph	Neo4j

"NoSQL" = Not-Only-SQL: scalability, schema flexibility, no standard model/API. **Decoupled (lakehouse)**: separate compute/storage for elasticity.

Apache Flink = pipelined dataflow, strong for *streaming*; cost-based optimiser but no join re-ordering; manages raw byte[] in 32 KB pages to dodge GC. **YARN** = the cluster resource manager letting MR/Spark/Flink coexist (per-app Application Master).

Data streams = unbounded, process with windows; Spark Streaming / Flink DataStream. RDBMS column on the NoSQL comparison: tuples, schema-first, SQL, ACID, in-place updates.

17 · If You See X, Compute Y

METHOD TRIGGERS

- **Page counts + buffer M, "join"** ⇒ plug all 5 join formulas; smaller = outer.
- **"sort N pages, B buffer"** ⇒ 1+⌈log_(B−1)⌈N/B⌉⌉ passes, 2N·passes.
- **"B+-tree search/height"** ⇒ ⌈log_F N⌉+1.
- **"insert into full bucket"** ⇒ split; double dir iff local=global.
- **"V distinct values, equality"** ⇒ matches = |R|/V; push σ first.
- **CLOCK/GCLOCK trace** ⇒ set ref bit on hits too.
- **"join on N nodes"** ⇒ pick broadcast/shuffle/replicate, re-run the join formula per node, ÷ #nodes.
- **"narrow vs wide dep"** ⇒ wide = shuffle = stage boundary.
- **"k-node cluster MTBF"** ⇒ MTBF₁/k.
- **"reduction / selectivity"** ⇒ RF=1/V; multiply RFs (independence); matches = |R|·∏RF.
- **"range-scan, unclustered idx"** ⇒ (h+1) + 1 I/O per matching row.
- **"INLJ cost c"** ⇒ c = 1 + h + #matches per outer row.

SIA → *Exam discipline: state the formula, sub numbers, box the I/O count, name which relation is outer. The exam tests by-hand cost, not your code.*

good luck. count the I/Os.