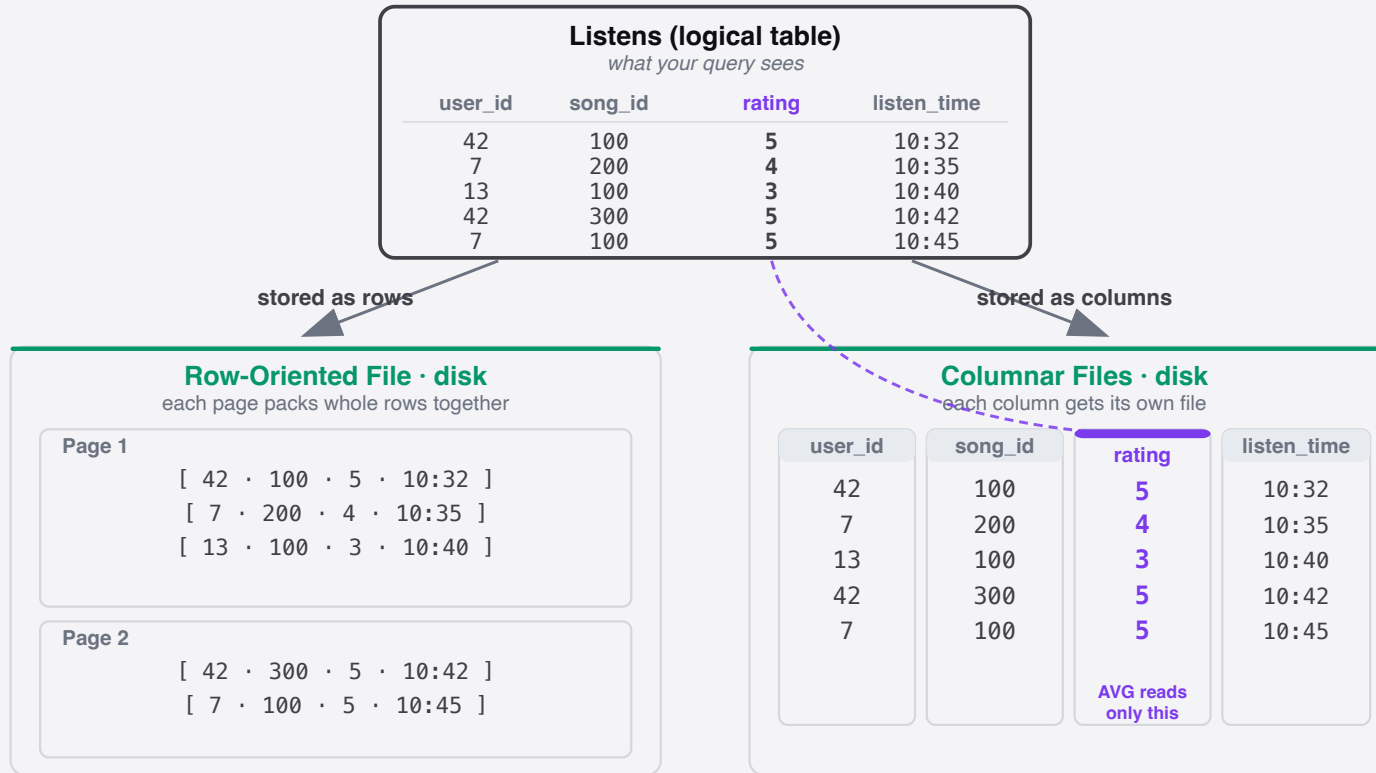


M3

Storage Layout: Row vs Columnar

A storage layout specifies how a table's bytes are stored on disk.

Same logical table, two physical storage layouts



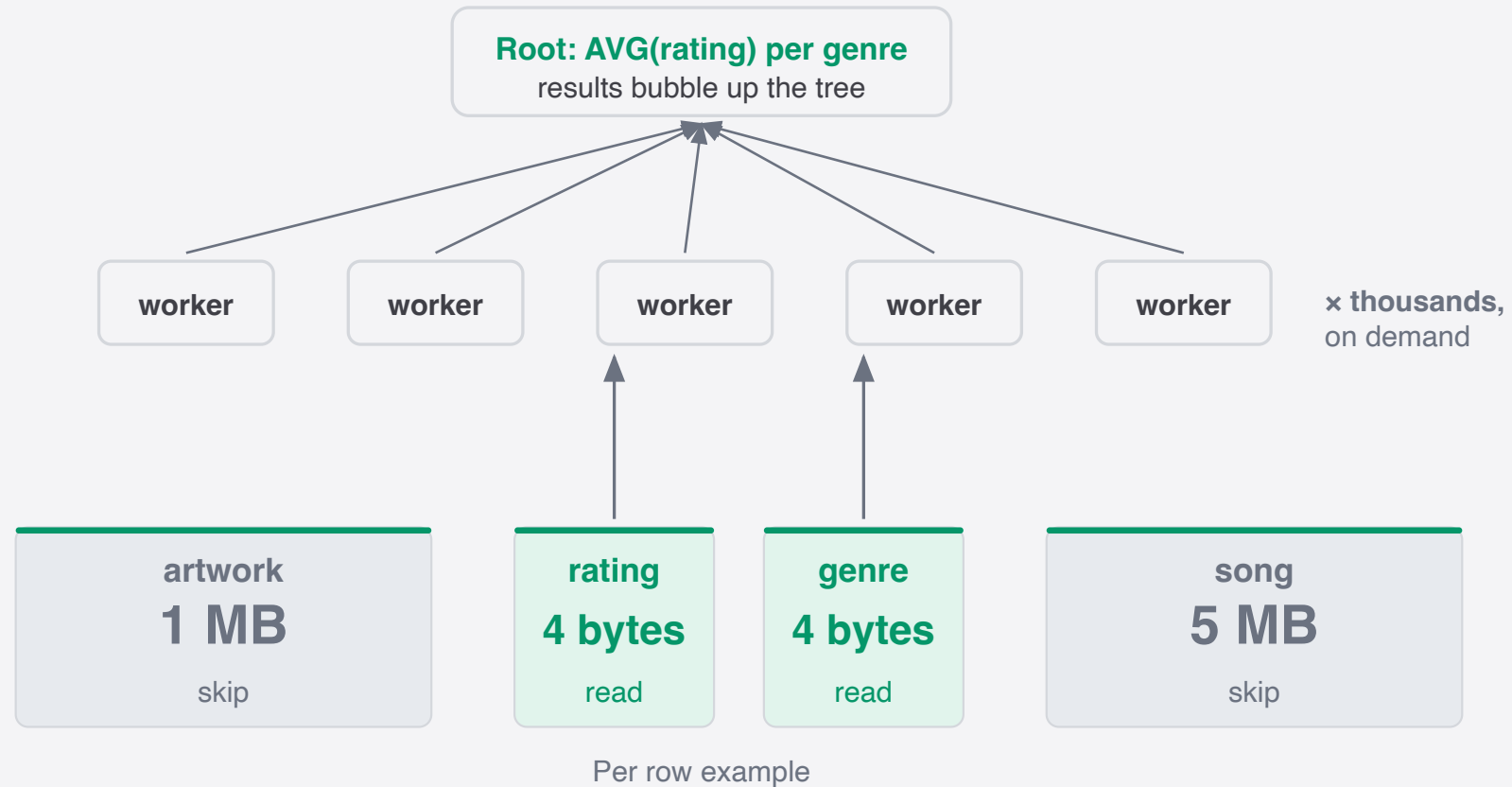
The physical layout decides which bytes a query reads. The same logical Listens table appears in both layouts. Row-oriented storage packs whole rows into a page, so reading row 5 returns all four columns in one read and a write is one append. Columnar storage gives each column its own file, so AVG(rating) touches only the rating file and skips the other three. Which layout wins depends on the query: SELECT AVG(rating) favors columnar (row layout wastes about 75% of the I/O on this 4-column toy, far more on wide tables), while SELECT * WHERE listen_id = 42 and inserts or updates favor row, since one record is one packed page rather than a seek-and-reassemble across four files.

M3

Case Study 3.1: BigQuery, From Web Crawl to Petabyte Analytics

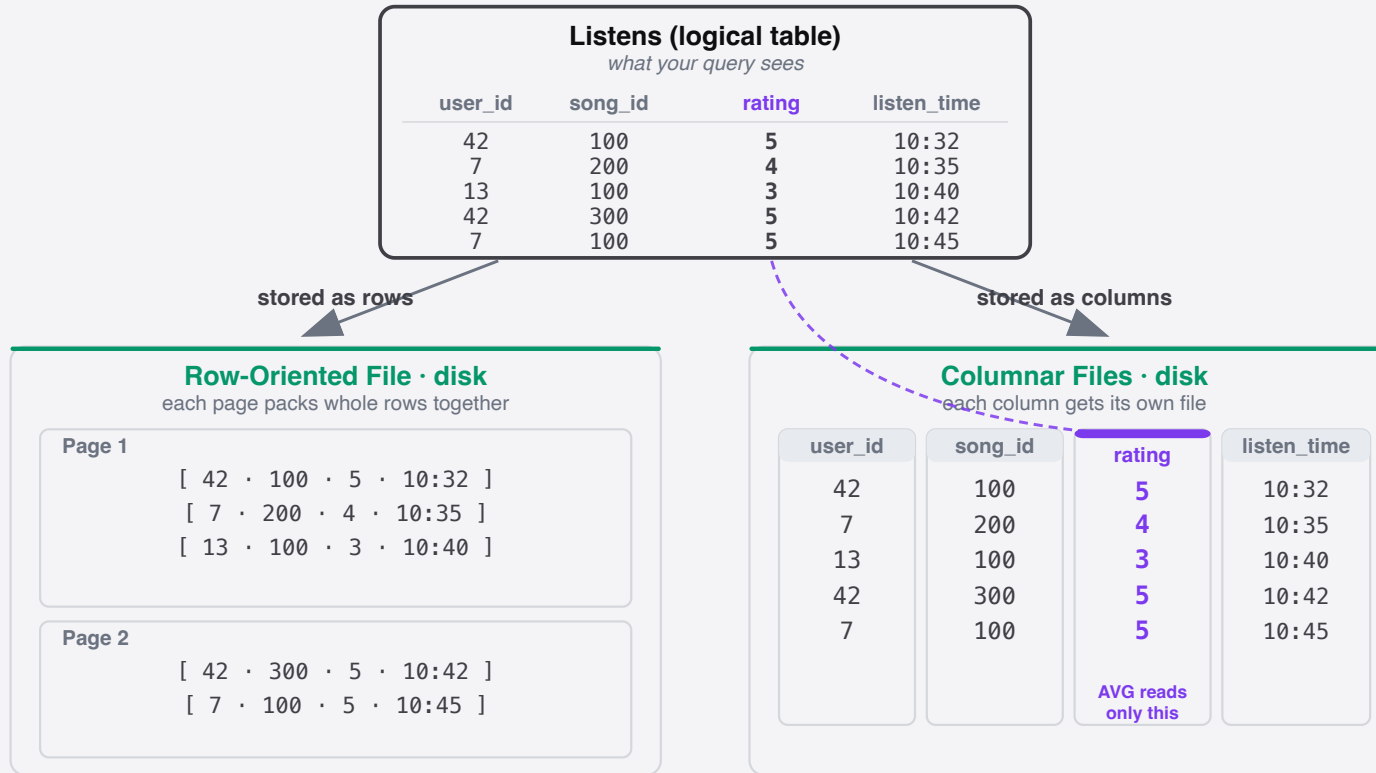
BigQuery scales to trillions of rows by storing data in a columnar format and decoupling storage from compute.

Read 4-byte ratings, skip 5 MB songs



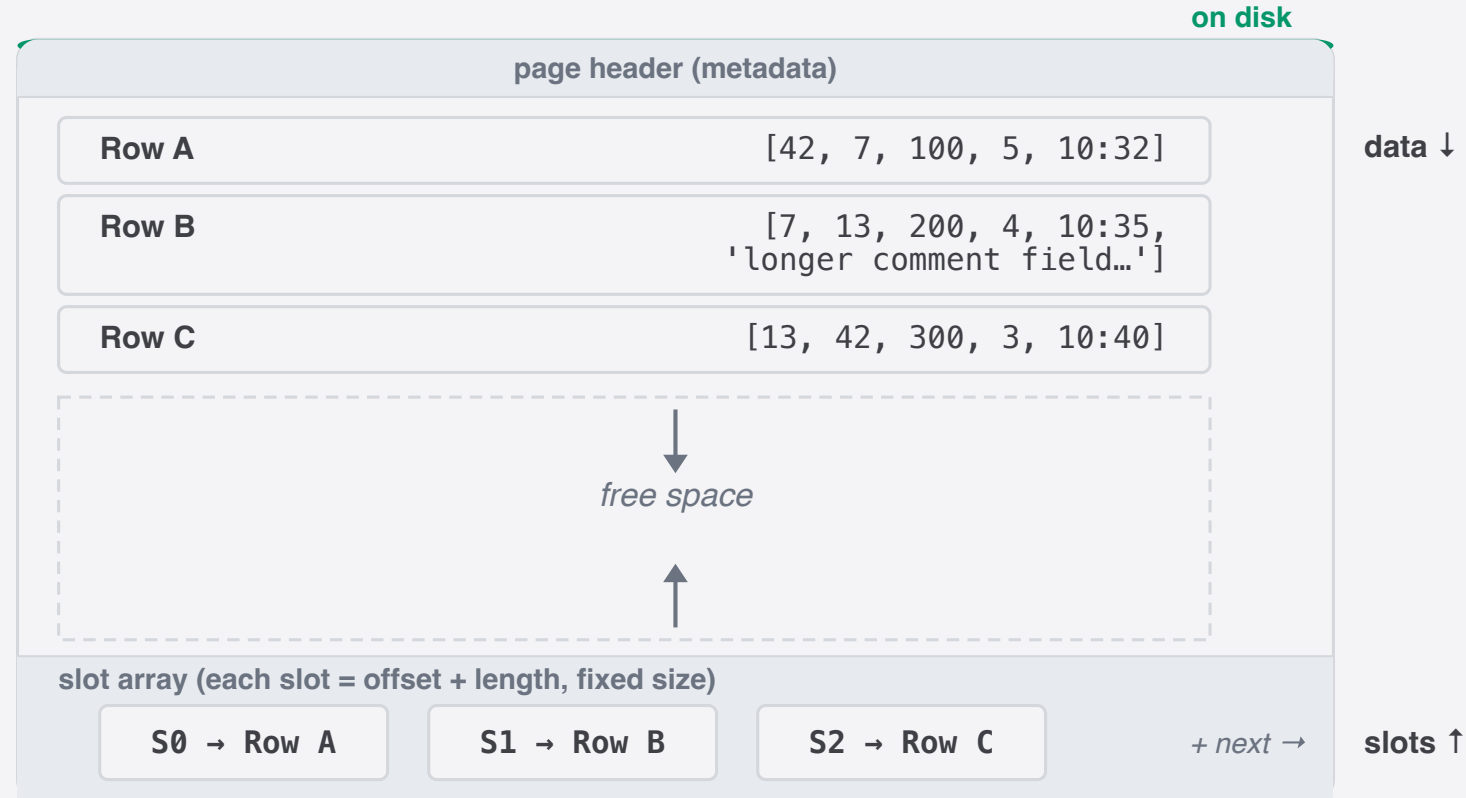
Each row in this illustrative table has a 4-byte rating, a 4-byte genre dictionary code, 1 MB of artwork, and 5 MB of song audio. The query reads the two small columns and skips the media. Workers scan those columns in parallel and combine partial results. Thin green bars identify on-disk columns.

Same logical table, two physical storage layouts



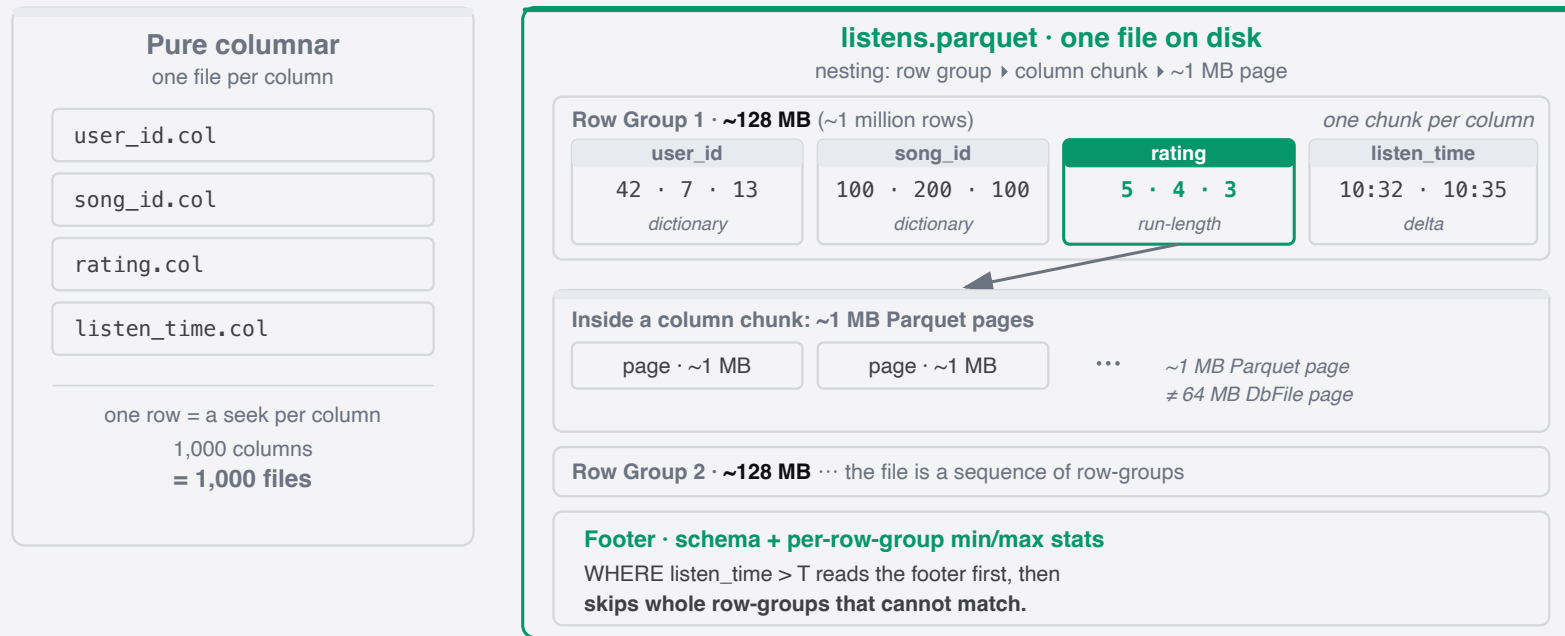
The physical layout decides which bytes a query reads. The same logical Listens table appears in both layouts. Row-oriented storage packs whole rows into a page, so reading row 5 returns all four columns in one read and a write is one append. Columnar storage gives each column its own file, so `AVG(rating)` touches only the rating file and skips the other three. Which layout wins depends on the query: `SELECT AVG(rating)` favors columnar (row layout wastes about 75% of the I/O on this 4-column toy, far more on wide tables), while `SELECT * WHERE listen_id = 42` and inserts or updates favor row, since one record is one packed page rather than a seek-and-reassemble across four files.

Slotted Page (Row-Oriented Layout)



A 64 MB slotted page stores variable-length records from the top downward and a fixed-size slot array from the bottom upward. Free space remains between them. Each slot stores an (offset, length) pointer, so S0 points to Row A and S1 to Row B. An in-place update writes the row without moving it when the row keeps the same size. When a row grows, the DB moves the row to a new location inside the page and updates the slot, so references like page_id, slot_id remain stable.

Pure columnar versus Parquet



The trade

Pay: compression resets at each row-group boundary, a sliver.

Gain: a row stays local · 1 file, not 1,000 files · the row-group is the unit of work for writes and parallel scans.

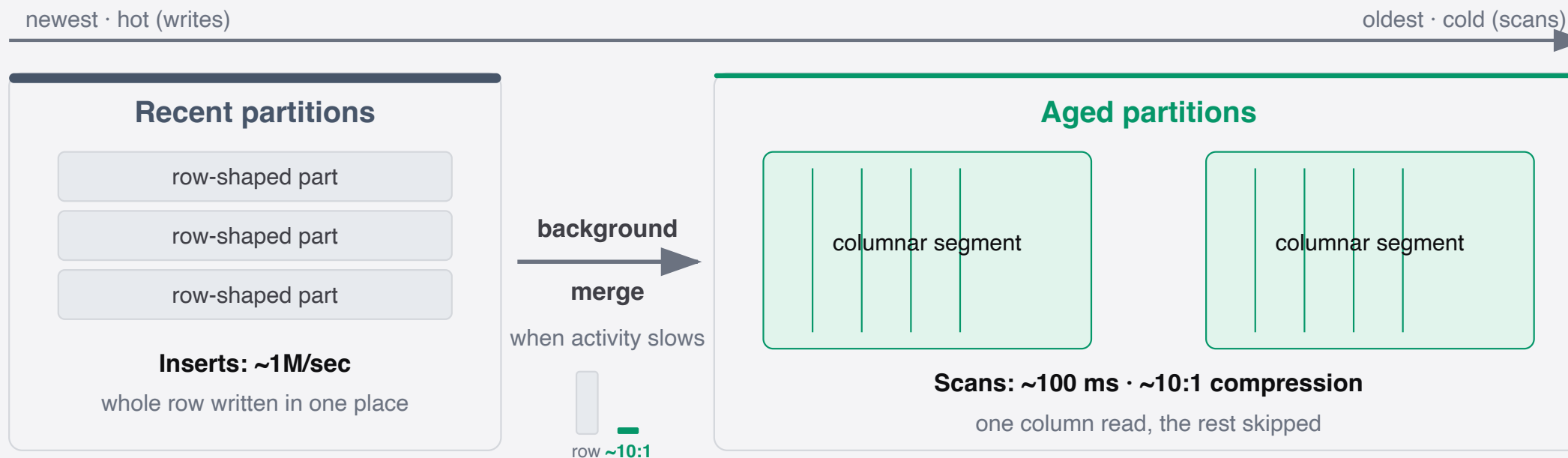
Parquet stores columnar data inside row-groups in one file. A pure columnar layout stores one file per column, so reading one row performs a seek per column and 1,000 columns produce 1,000 files. Parquet keeps per-column compression and scan behavior. A row stays within one row-group, and row-groups support writes and parallel scans. The footer stores per-row-group min/max statistics, and predicates can skip groups without reading their column chunks. Encodings stop at row-group boundaries, which reduces compression slightly.

M3

Hybrid Storage: One Table, Two Layouts

Hybrid storage keeps row-oriented and column-oriented layouts in the same table.

One table, two layouts, picked by data age



Unified query engine: one SQL surface reads both layouts and merges the result

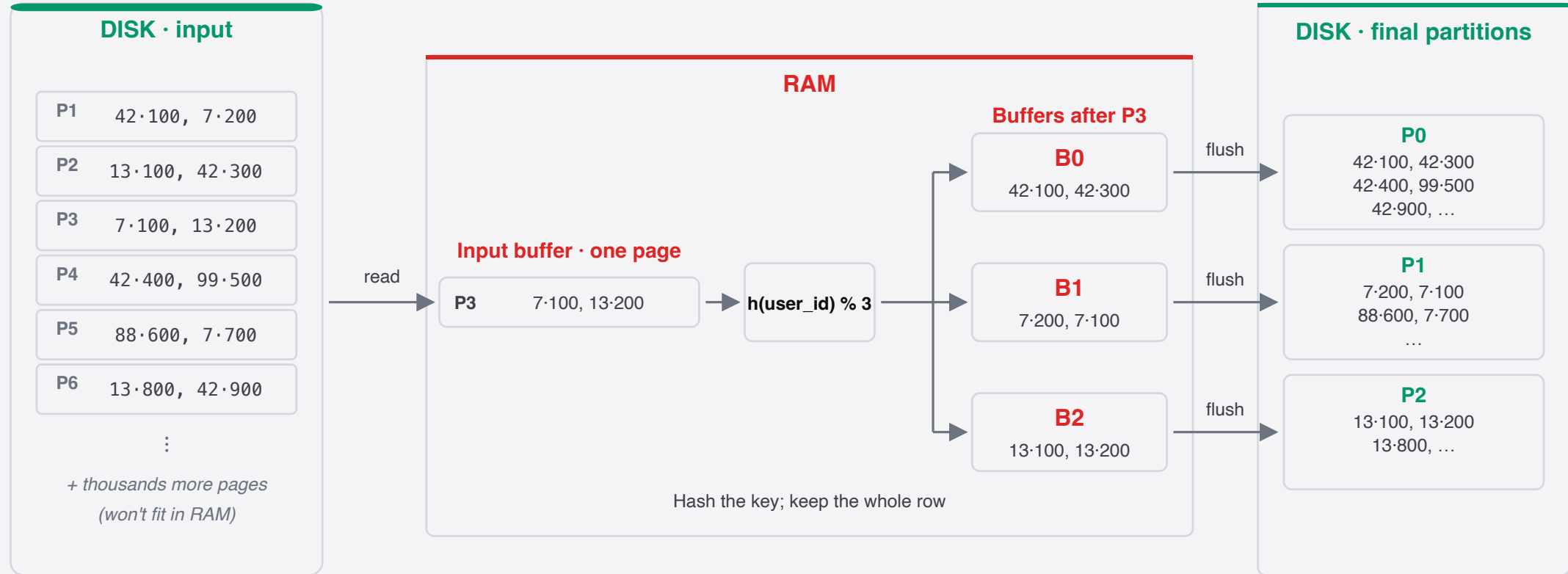
listen_time provides a partitioning dimension. The system writes fresh listens into small row-shaped parts (grey) to accept inserts at about 1M/sec by writing each row contiguously. A background merge rewrites older parts into columnar segments (green) after activity slows. Analytics scan one column and skip others at roughly 10:1 compression. One query engine reads both layouts through the same SQL surface.

M3

Hash Partitioning: Divide and Conquer

Hash partitioning splits a too-big-for-RAM table into N partitions by hashing a key.

Hash Partitioning: split a too-big-for-RAM file into B partitions



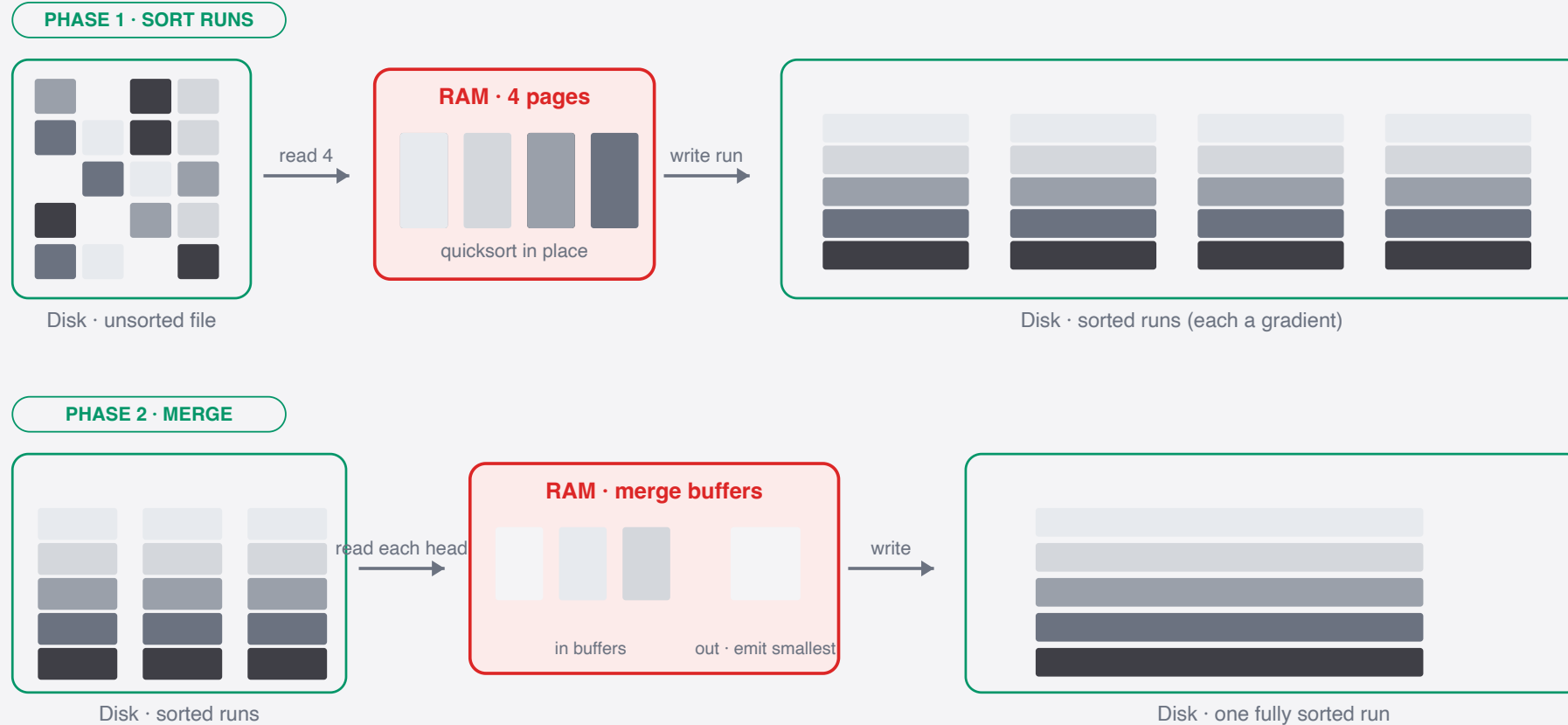
Hash partitioning makes one pass over a too-big-for-RAM file using a small RAM workspace. A page streams into the input slot, each row hashes by $h(\text{user_id}) \% B$ and routes to one of $\sim B$ output buffers, and a full buffer flushes to its partition file on disk. Rows with the same key always land in the same partition, so later stages can sort, join, or group one partition at a time. See the IO Cost Summary for the cost breakdown.

M3

BigSort: Sorting TBs of Data with GBs of RAM

BigSort (or external merge-sort) sorts a table larger than RAM by reading RAM-sized chunks, sorting each chunk in memory, writing each sorted chunk to disk, then merging the sorted chunks.

External Merge Sort: 4 Pages of RAM at a Time



Phase 1 reads four pages into memory, quicksorts them in place, and writes the pages back as one sorted run, repeating until the file becomes a set of sorted runs. Phase 2 keeps the head page of each run in memory, repeatedly selects the smallest remaining value, and writes it to the output to merge the runs into one sorted file. Each tile is a page shaded by its sort key, light for small and dark for large.

BigSort: move every page yourself

Input: Listens

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58
442	195	417	244	173	292	468	265	165	287	371	86	453	484	318	447	44	111	285	485	112	423	479	490	57	14	264	415	458	127	96	37	14	103	449	245	451	399	218	390	22	448	350	148	896	29	219	48	316	297	421	271	453	158	223	335	228	122
486	165	241	62	63	343	141	173	378	249	433	467	52	239	229	128	466	52	220	130	474	426	195	314	273	463	239	73	474	356	346	79	384	356	276	52	135	158	95	67	170	68	113	266	456	378	389	65	443	254	457	30	46	491	159	63	12	
213	141	81	14	88	323	173	345	158	138	245	145	373	242	213	187	159	217	306	360	217	217	327	114	280	487	67	162	316	436	352	40	225	285	292	448	257	460	443	10	478	486	339	493	112	91	72	335	3	12	327	296	21	134	258	297	174	
68	95	80	65	368	120	3	183	403	226	132	247	194	118	279	366	457	5	341	418	218	412	458	128	239	356	484	107	53	9	181	297	294	442	438	227	57	205	200	386	404	238	74	114	347	378	229	367	56	167	210	274	228	438	138	5	241	8
15	264	294	259	122	466	164	311	86	41	86	143	224	254	444	11	404	381	296	220	126	257	443	192	78	439	115	105	464	50	69	192	96	88	207	417	361	313	389	492	286	166	113	429	142	426	164	126	464	204	456	347	77	105	123	345	337	
316	253	391	73	442	229	457	202	177	415	471	221	115	216	271	31	13	105	250	84	4	249	338	443	94	166	236	147	431	71	422	140	98	70	124	401	483	392	323	206	465	253	405	351	43	38	116	289	75	442	261	236	348	140	385	129		
200	13	345	93	353	32	87	207	308	451	313	44	341	54	114	196	274	378	407	497	411	259	411	152	13	26	190	235	140	343	351	52	210	427	438	463	31	452	130	255	85	221	54	123	154	236	76	72	388	843	135	238	103	224	424	421	139	

Split and Sort

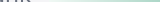
[illegible]

Merge Pass 1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	1	2	3	4	5	6	7	8	9	10
3	52	80	93	122	145	173	200	226	245	265	311	345	378	442	466	4	31	53	78	107	130	181	217	239	274	314	351	404	422	454	474	10	48	68	91	114	140	170	218	238	285	347	378	396	427	448	465	3	46	123	158	224	254	297	345	435	459
13	54	81	95	128	158	177	202	229	247	271	313	345	391	448	467	5	37	57	79	111	140	181	217	249	280	316	352	407	423	457	474	14	52	70	95	116	142	183	219	245	286	350	378	399	429	449	468	5	63	129	159	224	258	297	347	442	467
14	62	86	111	132	164	183	207	229	249	279	316	353	403	446	468	9	40	58	84	112	147	190	220	250	285	327	356	417	426	468	478	22	54	72	96	123	148	200	221	253	292	351	384	401	438	451	478	8	75	134	167	228	261	316	348	446	469
15	63	86	114	138	165	187	213	239	253	287	318	366	415	447	471	13	44	67	94	114	152	192	220	257	294	338	356	411	431	463	486	29	56	72	98	124	154	205	225	255	313	356	386	404	438	452	479	12	77	135	174	229	271	327	385	456	480
32	65	86	115	141	165	194	213	241	254	292	323	368	417	451	484	13	50	69	96	115	159	192	235	259	296	341	360	412	439	463	486	31	57	74	103	126	158	206	227	257	333	361	389	405	442	456	486	12	103	138	204	236	274	335	388	456	486
41	68	87	118	141	174	195	216	242	259	294	341	371	424	455	486	14	52	71	105	126	162	195	236	264	297	343	378	415	443	466	486	38	65	85	112	130	164	207	229	262	336	367	390	417	443	464	486	21	105	139	210	238	289	337	421	457	487
44	73	88	120	143	173	196	221	244	264	308	343	373	433	457	500	26	52	73	105	127	168	217	239	273	308	346	381	413	442	467	497	43	67	88	113	135	166	210	236	276	339	368	392	426	448	464	493	30	122	140	223	241	296	343	427	468	497

Merge Pass 2

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58
3	10	14	30	41	52	56	65	70	75	85	91	98	111	114	122	128	135	141	152	162	167	181	192	202	210	218	224	229	238	242	250	257	265	276	289	297	313	327	339	345	351	360	373	385	396	407	417	426	438	443	449	454	460	467	474	481	488
3	12	14	31	43	52	57	65	71	77	86	93	103	111	115	123	129	138	141	154	164	170	181	194	204	213	219	224	229	238	244	253	258	266	279	292	297	314	327	341	345	351	361	378	386	399	411	419	426	438	443	449	456	463	467	474	481	488
4	12	15	31	44	52	57	67	72	78	86	94	103	112	115	123	130	138	142	158	164	173	183	195	205	213	220	225	229	239	245	253	259	271	280	292	297	316	333	341	346	352	366	378	386	401	411	421	427	439	443	451	456	463	468	479	490	
5	13	21	32	44	52	58	67	72	79	86	95	105	112	116	124	130	139	143	158	165	173	183	195	206	216	220	226	235	239	245	254	259	271	285	294	306	316	335	343	347	353	367	378	389	403	412	421	428	442	444	451	457	463	468	481	491	
5	13	22	37	46	53	62	68	73	80	87	95	105	113	118	126	132	140	145	158	165	173	186	196	207	217	221	227	236	239	247	254	261	273	285	294	308	316	335	343	347	356	368	378	390	404	415	422	431	442	447	451	457	461	469			
8	13	26	38	48	54	63	68	73	81	88	96	105	114	120	126	134	140	147	159	166	174	190	200	207	217	221	228	236	241	249	255	264	274	286	296	311	318	337	343	348	356	369	381	391	404	415	422	433	442	448	451	466	472	485	496		
9	14	29	40	50	54	63	69	74	84	88	96	107	114	122	127	135	140	148	159	166	177	192	200	210	217	223	229	236	241	249	257	264	274	287	296	313	323	338	345	350	356	371	384	392	405	417	424	433	442	448	453	458	466	474	486	497	

small rank  large rank

The official CS145 IO equations

CS 145 Equations – v1.3 (Updated 9/1/25)

Many CS algorithms assume data fits in RAM, and focus on a CPUcost model (e.g., $O(n \log n)$ for Sorting). “Big” data does not fit in RAM.
Our Goal: Model IOcosts for working with data in (slow) SSD/HDDs and RAM (versus fast CPUs)

[1] Basic IO Cost model for RAM, SSD, HDD

Assume: 1 (disk) Block = 1 (RAM) Page.

1. $numPages = \lceil \frac{numMBs}{pageSize} \rceil$
2. ReadCost per page (C_r) = 1 IO.
WriteCost per page (C_w): k IOs
($1 \leq k \leq 10$ – Based on hardware! Default: $k = 1$)
3. Time per IO (secs): $accessLatency + \frac{pageSize}{scanSpeed}$

Example

E.g., $pageSize = 64$ MBs, $numMBs = 630$ MBs
SSD: $accessTime = 10$ us, $scanSpeed = 5$ GBps

- $numPages = \lceil \frac{630MB}{64MBs} \rceil = 10$
- $C_r = 1IO = 10us + \frac{64MB}{5GBps}$ secs
- Time to read 10 pages = $10 * C_r$
- Time to read 100 pages on 20 machines = $5 * C_r$

[2] HP and BigSort ‘R’ on column X

For table ‘R’, $T(R) = \#$ of rows in R. $P(R) = \#$ of pages in R. Buffer (B) = # of RAM pages.

1. **HashPartition(R)**: IOcost = $(C_r + C_w) * P(R)$
2. **BigSort(R)**: Let $N = P(R)$. IOcost = $(C_r + C_w) * N * (1 + \lceil \log_B \frac{N}{2B} \rceil)$

Special case for BigSort(R) ($C_r, C_w = 1$):

- (Small data) $N \leq 2B$? IOcost = $2N$
- If $2B < N \leq 2B^2$? $4N$; If $2B^2 < N \leq 2B^3$? $6N$

Example – $C_r, C_w = 1$

- $P(R) = 1000$, $B = 100$
- $HP(R) = 2 * 1000 = 2000$ IOs
- $BigSort(R) = 2 * 1000 * (1 + \lceil \log_{100} \frac{1000}{2 * 100} \rceil) = 4000$ IOs

[3] Basic JOIN(R, S) on column X

Let OUT be the number of output pages for a JOIN depends on number of matching tuples and row size of JOIN result).

1. **BNLJ(R, S)**:
 $C_r * (P(R) + P(S) * \lceil \frac{P(R)}{B} \rceil) + C_w * OUT$
(Pick BNLJ(S, R), if cheaper – see Box[6])

Example – $C_r, C_w = 1$

- $P(R) = 1000$, $P(S) = 2000$, $B = 100$
- $BNLJ(R, S) = 1000 + 2000 * \lceil \frac{1000}{100} \rceil + OUT$
- $BNLJ(S, R) = 2000 + 1000 * \lceil \frac{2000}{100} \rceil + OUT$

[4] HPJ(R, S) on column X

1. **HPJ(R, S)**: $HP(R) + HP(S) + PJ(R, S) + C_w * OUT$

2. $HP(R) = (C_r + C_w) * P(R)$ // Ditto for $HP(S)$

3. $PJ(R, S) =$
 - (“Ideal”-case) $C_r * (P(R) + P(S))$
 - Skew? Try different hash.
 - Many duplicates? Try different algo!

4. $HPJ(R, R)$. I.e., a self-join
 $HPJ(R, R) = HP(R) + C_r * P(R) + C_w * OUT$

Special case – $C_r, C_w = 1$

- $HPJ(R, S) = 3(P(R) + P(S)) + OUT$

Example – $C_r, C_w = 1$

- $P(R) = 1000$, $P(S) = 2000$, $B = 100$
- $HPJ(R, S) = 3 * (1000 + 2000) + OUT$
- $HPJ(R, R) = 3 * 1000 + OUT$

[5] SMJ(R, S) with JOIN on column X

1. **SMJ(R, S)**: $BigSort(R) + BigSort(S) + MergeJoin(R, S) + C_w * OUT$
2. MergeJoin(R, S) with JOIN on column X:
 - (a) (Best-case: No Backup) $C_r * (P(R) + P(S))$
 - (b) (Worst-case: All Backup) $C_r * P(R) * P(S)$
3. $SMJ(R, R)$. I.e., a self-join
 - (a) (Best-case: No Backup) $BigSort(R) + C_r * P(R) + C_w * OUT$
 - (b) (Worst-case: All Backup) $BigSort(R) + C_r * P(R) * P(R) + C_w * OUT$
4. If R is pre-sorted, $BigSort(R) = 0$. (Same as S.)

Example – $C_r = C_w = 1$

- $P(R) = 1000$, $P(S) = 2000$, $B = 100$
- $SMJ(R, S) = 4000 + 8000 + 3000 + OUT$ IOs
 - $BigSort(R) = 2 * 1000 * (1 + \lceil \log_{100} \frac{1000}{2 * 100} \rceil) = 4000$
 - $BigSort(S) = 2 * 2000 * (1 + \lceil \log_{100} \frac{2000}{2 * 100} \rceil) = 8000$
 - $MergeJoin(R, S) = 3000$ IOs, no backup
 - $MergeJoin(R, S) = 1000 * 2000$ IOs, full backup

[6] Indexing: Hash Index and B+ Trees

- n : Number of data rows, N : Number of data pages
- d : Data row size, k : Key size, p : Pointer size
- M : Page size, h : index height

Data Pages: $N = \lceil \frac{n * d}{M} \rceil$

Index Fanout: $f = \lceil \frac{M}{k + p} \rceil$

Hash Index (Unsorted): $P(index) = \lceil \frac{n}{f} \rceil$

Hash Index (Sorted): $P(index) = \lceil \frac{N}{f} \rceil$

B+ Tree Height (Unsorted): $h \approx \lceil \log_f(n) \rceil$

B+ Tree Height (Sorted): $h \approx \lceil \log_f(N) \rceil$

Query IOs: (index IO + data IO) $\approx h + 1$

Example Values:

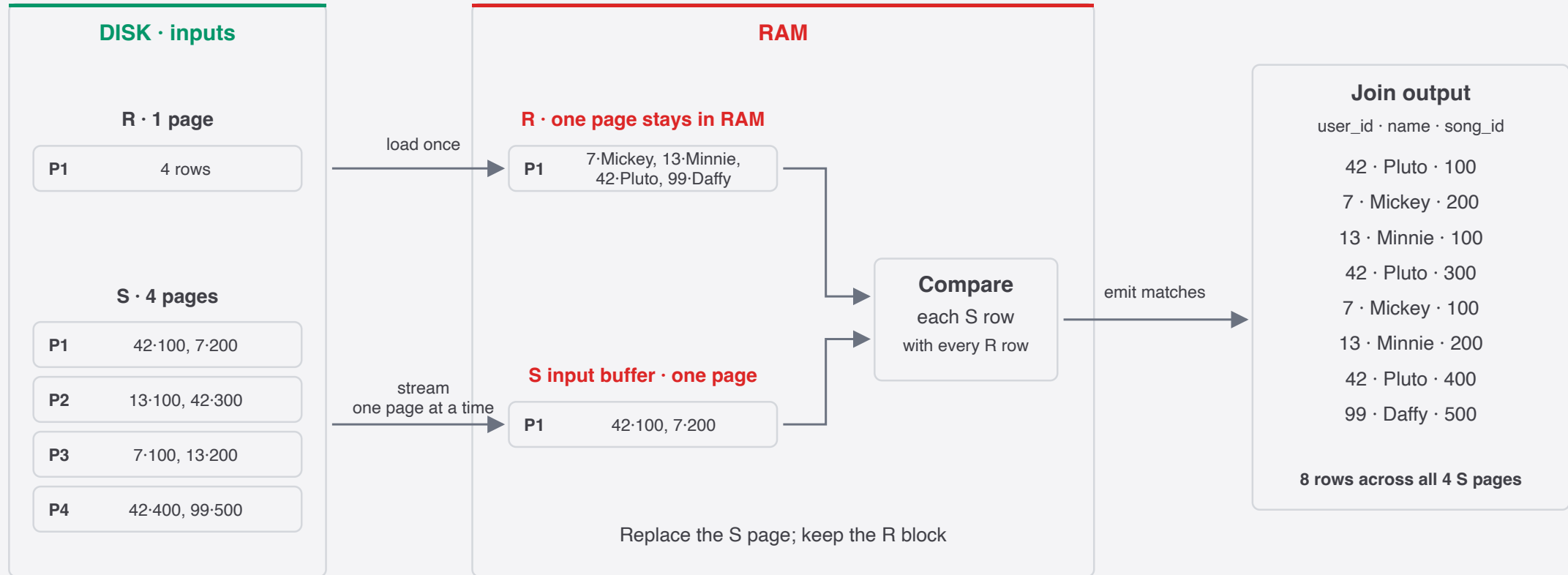
- $M = 64MB$, $k = 8$ bytes, $p = 8$ bytes, $n = 10^{12}$, $d = 1024$ bytes
- $f = \lceil \frac{64 * 1024 * 1024}{8 + 8} \rceil = 4,194,304$
- $N = \lceil \frac{10^{12} * 1024}{64 * 1024 * 1024} \rceil = 15258790$
- B+ Tree height (unsorted): $\lceil \log_{4194304}(10^9) \rceil = 2$
- B+ Tree height (sorted): $\lceil \log_{4194304}(15258790) \rceil = 2$

M3

Block Nested Loop Join: The Simple Double Loop

Block nested loop join loads a block of the outer table into RAM, scans the inner table once per block, and applies the join predicate to candidate row pairs.

Block Nested Loop Join: hold R in RAM, stream S one page at a time



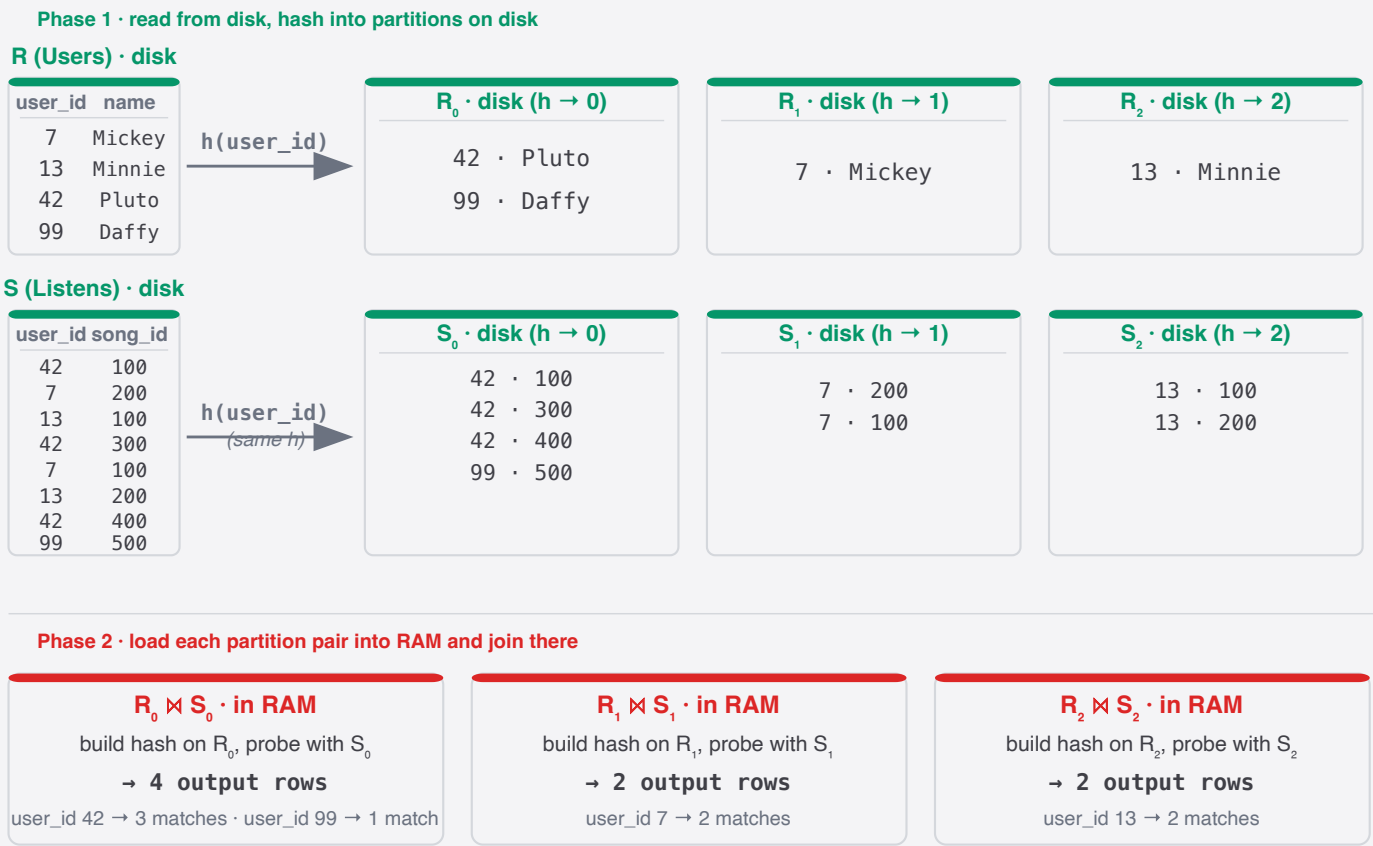
Block Nested Loop Join, $R \bowtie S$. Read the smaller table R from disk and hold it in RAM for the whole join. The larger table S streams past in 4 pages, each loaded into a one-page slot, probed against R, then discarded, producing 8 joined rows (2 per S page here). Best case, when R fits in B pages, is $P(R) + P(S) + OUT$; if R is bigger than B it becomes $P(R) + \lceil P(R)/B \rceil \times P(S) + OUT$, because S is re-scanned once per R-block. Always make the smaller table the outer one.

M3

Hash Partition Join

Hash partition join joins two tables that do not fit in RAM.

Hash Partition Join: $R \bowtie S$ on user_id



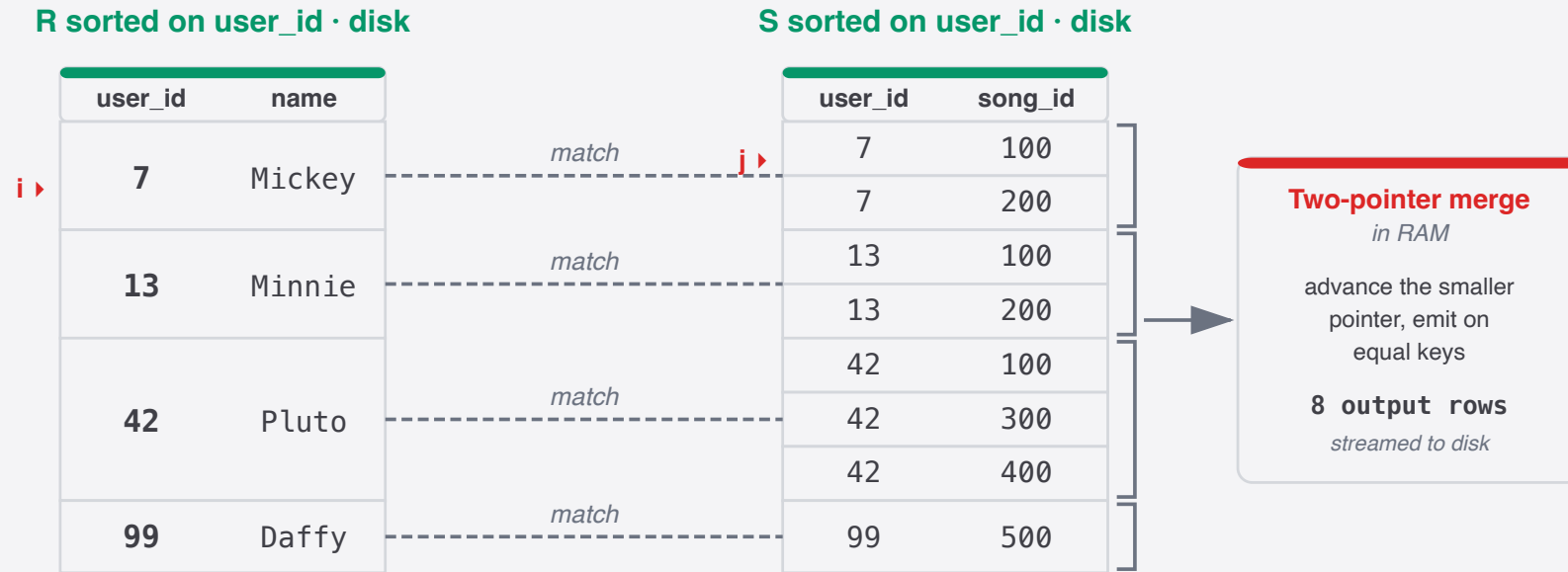
Phase 1 hashes both tables with the same function so matching user_ids always land in the same partition number. Phase 2 joins each (R_i, S_i) pair independently in RAM, building a hash table on R and probing with S , three parallelizable joins that yield $4 + 2 + 2 = 8$ rows. Best-case cost is $3(N+M) + \text{OUT}$ (read each table, write partitions, read them back, write the output). The worst case is hash skew, where a hot key overflows a partition past RAM and recursive re-partitioning or a sort-merge fallback adds another pass.

M3

Sort-Merge Join: When Order Matters

Sort-merge join sorts both tables on the join key, then scans them in lockstep and emits matching rows for equal keys.

Sort-Merge Join: walk both sorted tables in one pass



Sort-Merge Join, $R \bowtie S$ on user_id (Phase 2 shown). Phase 1 (not shown) BigSorts both R and S on the join key (see the big sort page). Phase 2 scans the two sorted runs with two pointers. Equal keys form a matching key group, and the operator emits every pair in the group. Unequal keys advance the smaller side. The example advances through user_id 7 → 2, 13 → 2, 42 → 3, 99 → 1, total 8 rows. A key that appears on only one side emits nothing under inner join semantics. A key with m matches on one side and n on the other emits the $m \times n$ cross-product for that key. The output stream stays sorted on the join key, so a downstream operator that requires that order does not add a sort. Best case is $P(R) + P(S) + OUT$ for one linear scan per run, plus the two Phase-1 BigSort costs. Worst case is $O(P(R) \times P(S)) + OUT$ when one key dominates and the merge performs a per-key cross-product. OUT can dominate when match rates are high.

M3

IO Algorithms: Cost Analysis & Intuition

The IO cost of every algorithm in this module (BigSort, HashPartition, BNLJ, SMJ, HPJ) is a function of table size, buffer size, and the read/write cost ratio.

The official CS145 IO equations

CS 145 Equations – v1.3 (Updated 9/1/25)

Many CS algorithms assume data fits in RAM, and focus on a CPUcost model (e.g., $O(n \log n)$ for Sorting). “Big” data does not fit in RAM.
Our Goal: Model IOcosts for working with data in (slow) SSD/HDDs and RAM (versus fast CPUs)

[1] Basic IO Cost model for RAM, SSD, HDD

Assume: 1 (disk) Block = 1 (RAM) Page.

1. $numPages = \lceil \frac{numMBs}{pageSize} \rceil$
2. ReadCost per page (C_r) = 1 IO.
WriteCost per page (C_w): k IOs
($1 \leq k \leq 10$ – Based on hardware! Default: $k = 1$)
3. Time per IO (secs): $accessLatency + \frac{pageSize}{scanSpeed}$

Example

E.g., $pageSize = 64$ MBs, $numMBs = 630$ MBs
SSD: $accessTime = 10$ us, $scanSpeed = 5$ GBps

- $numPages = \lceil \frac{630MB}{64MBs} \rceil = 10$
- $C_r = 1IO = 10us + \frac{64MB}{5GBps}$ secs
- Time to read 10 pages = $10 * C_r$
- Time to read 100 pages on 20 machines = $5 * C_r$

[2] HP and BigSort ‘R’ on column X

For table ‘R’, $T(R) = \#$ of rows in R. $P(R) = \#$ of pages in R. Buffer (B) = # of RAM pages.

1. **HashPartition(R)**: IOcost = $(C_r + C_w) * P(R)$
2. **BigSort(R)**: Let $N = P(R)$. IOcost = $(C_r + C_w) * N * (1 + \lceil \log_B \frac{N}{2B} \rceil)$

Special case for BigSort(R) ($C_r, C_w = 1$):

- (Small data) $N \leq 2B$? IOcost = $2N$
- If $2B < N \leq 2B^2$? $4N$; If $2B^2 < N \leq 2B^3$? $6N$

Example – $C_r, C_w = 1$

- $P(R) = 1000$, $B = 100$
- $HP(R) = 2 * 1000 = 2000$ IOs
- $BigSort(R) = 2 * 1000 * (1 + \lceil \log_{100} \frac{1000}{2 * 100} \rceil) = 4000$ IOs

[3] Basic JOIN(R, S) on column X

Let OUT be the number of output pages for a JOIN depends on number of matching tuples and row size of JOIN result).

1. **BNLJ(R, S)**:
 $C_r * (P(R) + P(S) * \lceil \frac{P(R)}{B} \rceil) + C_w * OUT$
(Pick BNLJ(S, R), if cheaper – see Box[6])

Example – $C_r, C_w = 1$

- $P(R) = 1000$, $P(S) = 2000$, $B = 100$
- $BNLJ(R, S) = 1000 + 2000 * \lceil \frac{1000}{100} \rceil + OUT$
- $BNLJ(S, R) = 2000 + 1000 * \lceil \frac{2000}{100} \rceil + OUT$

[4] HPJ(R, S) on column X

1. **HPJ(R, S)**: $HP(R) + HP(S) + PJ(R, S) + C_w * OUT$

2. $HP(R) = (C_r + C_w) * P(R)$ // Ditto for $HP(S)$

3. $PJ(R, S) =$
 - (“Ideal”-case) $C_r * (P(R) + P(S))$
 - Skew? Try different hash.
 - Many duplicates? Try different algo!

4. $HPJ(R, R)$. I.e., a self-join
 $HPJ(R, R) = HP(R) + C_r * P(R) + C_w * OUT$

Special case – $C_r, C_w = 1$

- $HPJ(R, S) = 3(P(R) + P(S)) + OUT$

Example – $C_r, C_w = 1$

- $P(R) = 1000$, $P(S) = 2000$, $B = 100$
- $HPJ(R, S) = 3 * (1000 + 2000) + OUT$
- $HPJ(R, R) = 3 * 1000 + OUT$

[5] SMJ(R, S) with JOIN on column X

1. **SMJ(R, S)**: $BigSort(R) + BigSort(S) + MergeJoin(R, S) + C_w * OUT$
2. MergeJoin(R, S) with JOIN on column X:
 - (a) (Best-case: No Backup) $C_r * (P(R) + P(S))$
 - (b) (Worst-case: All Backup) $C_r * P(R) * P(S)$
3. $SMJ(R, R)$. I.e., a self-join
 - (a) (Best-case: No Backup) $BigSort(R) + C_r * P(R) + C_w * OUT$
 - (b) (Worst-case: All Backup) $BigSort(R) + C_r * P(R) * P(R) + C_w * OUT$
4. If R is pre-sorted, $BigSort(R) = 0$. (Same as S.)

Example – $C_r = C_w = 1$

- $P(R) = 1000$, $P(S) = 2000$, $B = 100$
- $SMJ(R, S) = 4000 + 8000 + 3000 + OUT$ IOs
 - $BigSort(R) = 2 * 1000 * (1 + \lceil \log_{100} \frac{1000}{2 * 100} \rceil) = 4000$
 - $BigSort(S) = 2 * 2000 * (1 + \lceil \log_{100} \frac{2000}{2 * 100} \rceil) = 8000$
 - $MergeJoin(R, S) = 3000$ IOs, no backup
 - $MergeJoin(R, S) = 1000 * 2000$ IOs, full backup

[6] Indexing: Hash Index and B+ Trees

- n : Number of data rows, N : Number of data pages
- d : Data row size, k : Key size, p : Pointer size
- M : Page size, h : index height

Data Pages: $N = \lceil \frac{n * d}{M} \rceil$

Index Fanout: $f = \lceil \frac{M}{k + p} \rceil$

Hash Index (Unsorted): $P(index) = \lceil \frac{n}{f} \rceil$

Hash Index (Sorted): $P(index) = \lceil \frac{N}{f} \rceil$

B+ Tree Height (Unsorted): $h \approx \lceil \log_f(n) \rceil$

B+ Tree Height (Sorted): $h \approx \lceil \log_f(N) \rceil$

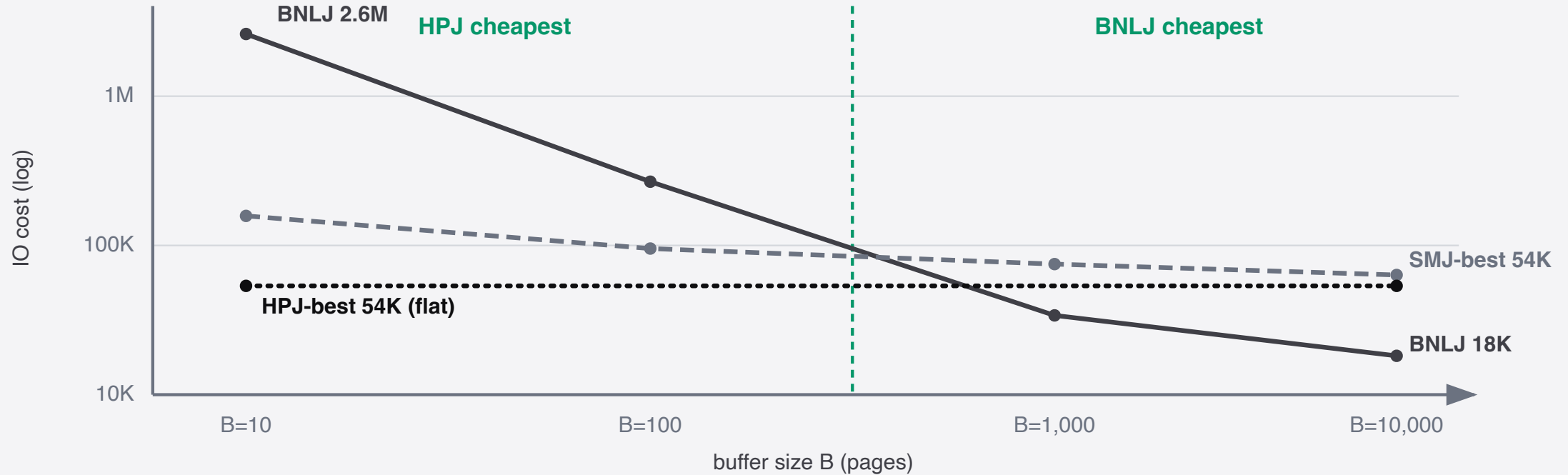
Query IOs: (index IO + data IO) $\approx h + 1$

Example Values:

- $M = 64MB$, $k = 8$ bytes, $p = 8$ bytes, $n = 10^{12}$, $d = 1024$ bytes
- $f = \lceil \frac{64 * 1024 * 1024}{8 + 8} \rceil = 4,194,304$
- $N = \lceil \frac{10^{12} * 1024}{64 * 1024 * 1024} \rceil = 15258790$
- B+ Tree height (unsorted): $\lceil \log_{4194304}(10^9) \rceil = 2$
- B+ Tree height (sorted): $\lceil \log_{4194304}(15258790) \rceil = 2$

Which join wins depends on the buffer, not the data

Songs3 100 GB $\bowtie$ Listens3 1 TB, $C_r = C_w = 1$, log scale. Verified from the M3 colab.



HPJ-best stays near 54K because partitioning does one read and one write independent of B. BNLJ drops from 2.6M to 18K because larger B reduces rescans of the inner table. SMJ-best drops as larger initial runs reduce merge passes. The crossover marks the point where BNLJ becomes cheaper than HPJ at this scale. SMJ with many duplicates can rescan during merge. HPJ with heavy skew can create oversized partitions, each up to 2 to 10x worse.