DESIGN PROPOSAL — APACHE DATAFUSION

Spilling Hash Join: running any join in a bounded memory budget

Author: Jay Zhan · Draft v0.4 · 2026-08-29 · Baseline: DataFusion 54.0.0 / main@55-dev

Short form for GitHub: hash-join-spilling-proposal.md. Prior art: apache/datafusion #17267, #12952, #1599.

Primer. A hash join loads one input entirely into an in-memory hash table (the **build side** — DataFusion uses the left input) and streams the other input past it (the **probe side**). DataFusion also splits query execution into target_partitions **output partitions**; this design further splits each join's build data into 16 **buckets**. “Partition” in this document always means the former; “bucket” always the latter.

Abstract. HashJoinExec is the last major DataFusion operator that cannot spill: when the build side exceeds the memory budget, the query fails with `Resources exhausted` rather than degrading. This document motivates hash-join spilling with a reproducible experiment, surveys five engines, and proposes a hybrid (adaptive) hash join behind a default-off flag: bit-for-bit identical to today's join while memory suffices — and structurally unreachable without a memory limit — partitioning and spilling build/probe bucket pairs when it does not, with Bloom-negative early resolution, bit-window recursion, role reversal, and a skew fallback. Twelve independently reviewable PRs, with numeric acceptance criteria and a stated path to default-on.

1. Problem statement

Under a configured memory limit, every other memory-hungry operator degrades gracefully: sort spills, aggregation spills, sort-merge join spills (#9359), RepartitionExec spills, and DataFusion 54 added a spilling nested-loop join. Hash join — the default join — kills the query:

```
DataFusion error: Resources exhausted: Failed to allocate additional 95.4 MB for
HashJoinInput[3] with 38.2 MB already allocated for this reservation - 51.9 MB remain
available for the total memory pool: fair(pool_size: 300.0 MB)
```

Users therefore choose between two bad options: keep `prefer_hash_join=true` and lose big joins outright, or set it false and route every join through sort-merge join — which §3 measures at **3.5× slower on joins that fit in memory**. Embedders cannot cap memory safely; memory-constrained CI is impossible (TPC-H/DS die at the first big join); and the failure is instant and binary — no partial progress.

Goal state: the same plan runs at any memory budget — full speed when it fits, incremental slowdown when it does not, hard failure only when disk is also exhausted. All of it behind `enable_hash_join_spilling` (default off).

2. What DataFusion has today

2.1 HashJoinExec anatomy (datafusion/physical-plan/src/joins/hash_join/)

- `collect_left_input()` streams the build side into `Vec<RecordBatch>`, calling `reservation.try_grow(batch_size)` per batch — where ResourcesExhausted surfaces today. It then builds one hash table (`JoinHashMapU32/U64`, or an `ArrayMap` perfect-hash for small dense integer keys) and concatenates everything into the single `JoinLeftData.batch`.
- **Two modes.** `CollectLeft` (small builds): build once behind `OnceAsync<JoinLeftData>`, shared read-only by all probe streams. `Partitioned` (large builds): both inputs hash-redistributed; every output partition builds its own table over disjoint keys and probes it with exactly one stream (`probe_threads_count = 1`).
- **Probe state machine** (stream.rs): `WaitBuildSide → FetchProbeBatch ⇄ ProcessProbeBatch → ExhaustedProbeSide → Completed`, with vectorized hash lookup, join-filter evaluation, and a visited-indices bitmap for outer/semi/anti/mark emission.
- **Dynamic filter pushdown.** The completed build feeds min/max `PartitionBounds`, small-build InList values, or hash-map membership (`PushdownStrategy`) through `SharedBuildAccumulator` into probe-side scans (§6.7 covers the interaction).

2.2 Spill infrastructure already in-tree (all reusable)

Component	Where	What it gives us
`SpillManager` / `InProgressSpillFile` / `SpillReaderStream`	physical-plan/src/spill/	Arrow IPC spill files; zstd/lz4 spill_compression; disk quota (max_temp_directory_size); SpillMetrics; StringView-aware sizing
`SpillPool`	spill/spill_pool.rs	FIFO multi-file spill channel with rotation; used by RepartitionExec and aggregates
`ReplayableStreamSource`	spill/replayable_spill_input.rs	Spill-once / replay-many input wrapper; powers the DF 54 NLJ fallback — reused as the skew fallback (§6.5)
`MemoryConsumer::with_can_spill(true)`	execution/src/memory_pool/	FairSpillPool grants spillable consumers a fair share; hash join does not set it today
NLJ `SpillState` machine	joins/nested_loop_join.rs	In-tree precedent of a join reacting to ResourcesExhausted by switching into a spilling mode

The disk layer, accounting, metrics, and an in-tree “join that spills” precedent all exist; only the hash-join algorithm on top is missing.

3. How large can a join be today? (measured)

Environment: datafusion-python **54.0.0**, two identical Parquet tables of 20M rows (`k BIGINT, payload VARCHAR(~40B)`, ~1.1 GB raw per side), `FairSpillPool(300 MB)`, `target_partitions=4`. Script in Appendix A; timings best-of-3.

Query (same data, same 300 MB budget)	Result	Detail
hash join `t_probe ⋈ t_build ON k`	**FAILS 0.2 s**	Resources exhausted (§1); no partial progress, no spill attempt

Query (same data, same 300 MB budget)	Result	Detail
same join via SMJ (`prefer_hash_join=false`)	**OK 3.7 s**	`SortExec` spills: spill_count=16, 155.3 MB
hash join, build side shrunk to 10M rows (fits)	**OK 0.70 s**	largest build that fits the per-consumer share
that same fitting join forced through SMJ	**OK 2.48 s**	**the workaround's tax: 3.5× on a join that fits**
hash aggregate `count(DISTINCT payload)`	**OK 11.1 s**	aggregation spill path handles the same volume
hash join, build side 12M rows	**FAILS**	fatal allocation is the hash-table build: `+57.2 MB with 22.9 MB held`

Analysis. (1) Today's ceiling: the entire build side (batches + ~8–16 B/row of hash table) must fit the consumer's share of the pool — and the join is not even marked can_spill. (2) The failure fires at **hash-table allocation time**, after the batches were accepted — motivating an explicit table-construction headroom reservation (§6.8). (3) The 3.5× row is the cost of the only workaround: it is paid on every join, including all the ones that fit.

4. How other engines solve it

Engine	Approach	What this proposal borrows
DuckDB v1.2 (PR #4189; "Saving Private Hash Join", VLDB '25)	Adaptive radix-partitioned hash join: keep as many partitions resident as fit; repartition loop for the rest; thread-local sinks with pointer swizzling through a unified buffer manager	Adaptivity — pay nothing when memory suffices; destage-largest; verify by re-running the entire suite with spilling forced (force_external)
Velox (spilling doc)	Parallel builders coordinate to spill the same partition set; recursion advances a hash-bit window (bits 29–31 → 32–35 → …); capacity M·(2^N)^L	Bit-window recursion — one hash, never re-hashed across levels; explicit depth cap
ClickHouse (grace_hash)	Grace hash join: bucket 0 in memory, all other buckets spill both sides up front, processed sequentially	Sequential bucket processing — plus two warnings: not adaptive, and shipped INNER/LEFT-only for years (type completeness must be day-one)
Spark (SPARK-32634)	Shuffled-hash join falls back to sort-based processing when the map exceeds memory; default engine is SMJ because it spills naturally	Confirms demand; sort-fallback rejected here so a hash join stays a hash join
Trino (spill docs)	Revocable memory; a subset of build partitions spills together with matching probe rows (peak ≈ 1/task.concurrency of build)	The hybrid "spill some, keep the rest hot" discipline; honesty that spill is insurance, not a fast path

Classic literature underneath all five: GRACE (Kitsuregawa '83) partitions everything to disk; **hybrid** hash join (DeWitt '84, Shapiro '86) keeps what fits and spills the rest — degrading proportionally instead of in a cliff; dynamic destaging and role reversal (Graefe et al., VLDB '98) pick eviction victims and swap build/probe roles

per partition. No engine that solved this well made spilling a plan-time decision, and none chose sort-fallback as the primary design.

5. Design goals, scope decisions, and non-goals

- **G1 — Zero regression when memory suffices:** when nothing spills the code path is today's — enforced numerically: join benchmarks within 2% of main (T1/T4 acceptance), and with no memory limit configured the spill machinery is structurally unreachable.
- **G2 — Complete join semantics from day one:** all JoinTypes, filters, null_equality, null-aware anti — enforced by a spill-forced test matrix (§10), not promised for later.
- **G3 — Bounded by disk, not RAM:** capacity ≈ share × 16 per recursion level; explicit failure mode only when disk/quota is exhausted.
- **G4 — Reuse, don't reinvent:** SpillManager files, compression, quotas, SpillMetrics, ReplayableStreamSource, FairSpillPool semantics.
- **G5 — Observable:** spill metrics in EXPLAIN ANALYZE like sort and SMJ.
- **G6 — Mergeable in small steps:** every PR independently reviewable and releasable behind the default-off flag.

5.1 Scope decision: CollectLeft mode (headline, not footnote)

Phase 1 implements spilling for Partitioned mode only. To keep the epic's promise honest, T8 adds a planner rule: **when the flag is on, joins are steered to Partitioned unless the build side is provably below the CollectLeft threshold** — so the joins that can hit the limit are exactly the joins that can spill. A statistics misestimate can still land an oversized build in CollectLeft; that keeps today's failure behavior, is called out in the config docs, and coordinated CollectLeft spilling is the first follow-up issue.

5.2 Scope decision: prefer_hash_join is untouched

No planner change to `prefer_hash_join`. What changes after default-on is the guidance: `prefer_hash_join=false` stops being the memory-safety workaround — §3 shows it costs 3.5× on fitting joins.

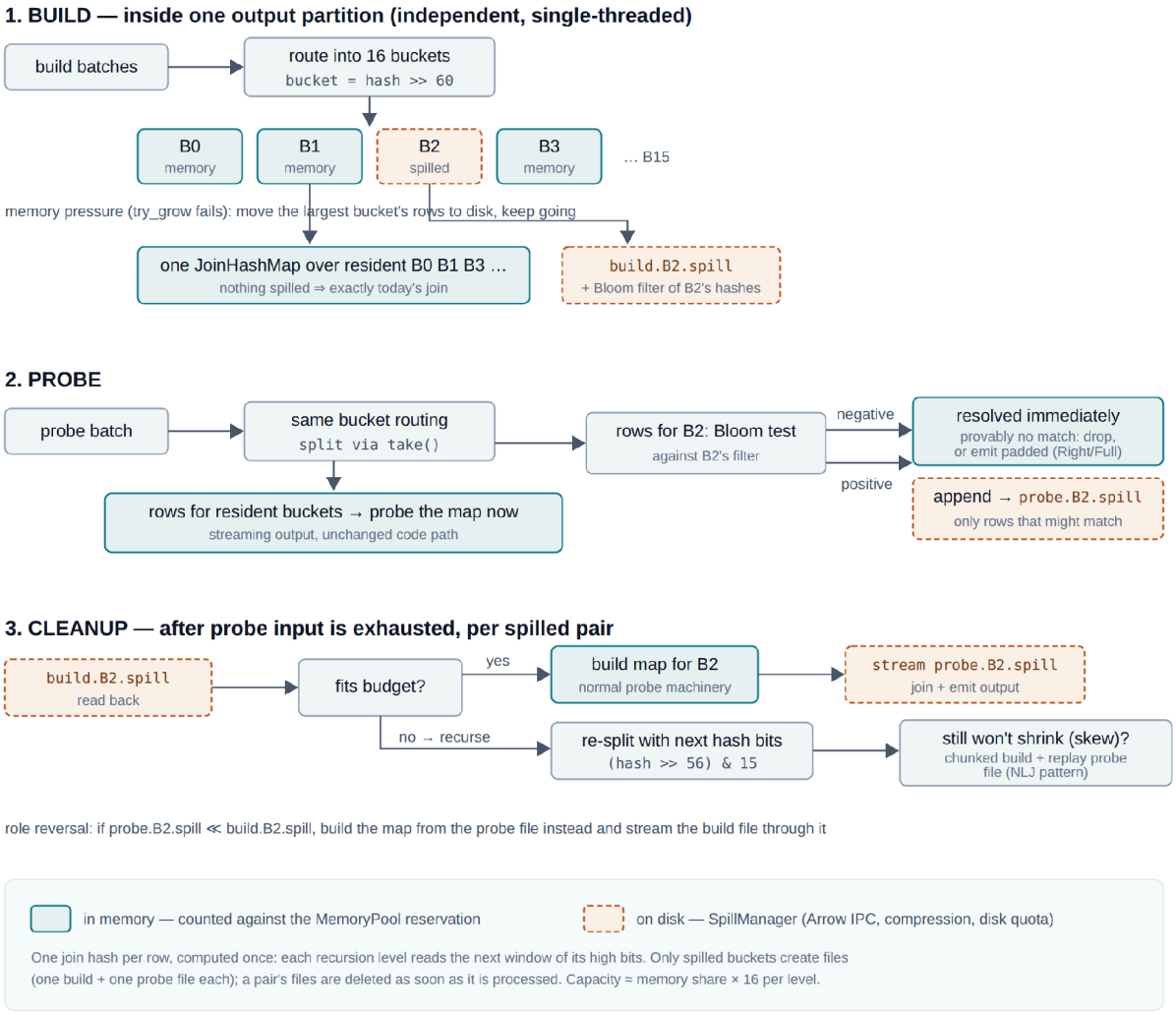
Non-goals (follow-up issues): coordinated CollectLeft spilling; SymmetricHashJoin; per-bucket membership pushdown and the probe re-scan strategy (both build on T7/#19858); global memory arbitration; spill-format changes (#14078).

6. Recommended design: partition-local hybrid hash join

6.1 The DataFusion-specific insight

In `Partitioned` mode — the mode large joins run in — each output partition already owns a private, single-threaded build+probe pipeline over hash-disjoint keys (§2.1). So each output partition can spill and recover **independently**. The cross-thread coordination DuckDB and Velox spend most of their spill machinery on (many threads feeding one shared table; agreeing on a common spill set) — and the main blocker named in #17267 — does not arise: the plan already did the dividing. Fairness between partition streams is the memory pool's existing job (`FairSpillPool` + `with_can_spill(true)`). Figure 1 shows one output partition's lifecycle.

Figure 1 — one output partition's hybrid hash join: build / probe / cleanup (full-resolution PNG attached alongside this document)



6.2 Build phase: what runs before, at, and after the first spill

Phase A — before any spill (G1, structurally). Exactly today's code: batches append to a single list; no routing, no hashing, no extra copies, no files. With no memory limit configured this phase is the whole story.

The transition — first `try_grow` failure (or optional watermark). Compute join hashes for the batches buffered so far (one catch-up pass — work that would have been done at map-build time anyway), split them into 16 buckets by a window of high hash bits, destage the largest bucket — append its batches to that bucket's `InProgressSpillFile`, build a Bloom filter over its hashes, shrink the reservation — and continue.

Phase B — spill-aware building. Incoming batches are routed as they arrive (vectorized `take()` split — the #17267 “dual take” cost, paid only in this phase and measured by T2); rows for spilled buckets stream to their files through a small coalescer; further pressure destages the next-largest bucket. At end of input: reserve table headroom (§6.8), then build **one JoinHashMap over all resident buckets** — when nothing spilled, identical to today.

Disk footprint. Only spilled buckets create files (one build + one probe file each): worst case 32 files per output partition per level — 128 at target_partitions=4, a few thousand on a 32-core machine with everything

spilling. A pair's files are deleted as soon as it is processed; recursion drains a parent before creating children; total bytes are bounded by `max_temp_directory_size`.

6.3 One hash per row, many windows (worked example)

The join computes one 64-bit hash per row (it already does, for the map). Every level of the design reads a different 4-bit window of that same hash, so nothing is ever re-hashed:

```
hash(k) = 0xA3F4_91C2_7B06_55E8

level 0 bucket  = hash >> 60         = 0xA = 10   → bucket B10 of 16
level 1 bucket  = (hash >> 56) & 15  = 0x3 =  3   → child bucket 3, if B10 recurses
level 2 bucket  = (hash >> 52) & 15  = 0xF = 15   → and so on (≤ 4 levels)

RepartitionExec routed rows with hash % target_partitions over a DIFFERENT
RandomState (its own seeds), so output-partition routing and bucket routing
are statistically independent — buckets do not collapse into one.
```

6.4 Probe phase

- Per probe batch: compute hashes once (as today). Nothing spilled → exactly today's code. Otherwise split: rows for resident buckets probe the in-memory map immediately (streaming); rows for spilled bucket i take the **Bloom test** — negative rows are resolved on the spot (§6.11), positive rows append (with their hash) to `probe.Bi.spill`.
- Vectorized lookup, join filters, null_equality, null-aware anti flags: untouched — pass 1 simply sees fewer rows.

6.5 Cleanup, recursion, and the skew guard

- New stream states after `ExhaustedProbeSide`: `RestoreSpilledBucket(i)` → `ProcessSpilledProbe(i)` → … → `Completed`. Restore build.Bi; if it fits, build its map and stream probe.Bi through the normal probe path; if not, re-split with the next bit window (§6.3), max 4 levels.
- **Skew guard:** a bucket that stops shrinking (duplicate-heavy key — more bits cannot split equal hashes) falls back to chunked build + probe replay via `ReplayableStreamSource` — the DF 54 NLJ pattern, correct for every join type with per-chunk bitmap care.
- **Role reversal:** if probe.Bi turned out much smaller than build.Bi, build the map from the probe file and stream the build file through it (§6.11).

6.6 Join-type completeness (mechanism per type — day-one requirement)

Join types	Mechanism under spilling
Inner	Rows joined in pass 1 (resident) or cleanup (spilled); each probe row processed exactly once
Left / LeftSemi / LeftAnti / LeftMark / Full (build side)	Per-bucket visited bitmap sized to that bucket's rows; resident buckets emit unmatched at ExhaustedProbeSide (today's path), each spilled bucket emits at the end of its cleanup iteration. Every build row lives in exactly one bucket, so bitmaps stay local — no merge

Join types	Mechanism under spilling
Right / RightSemi / RightAnti / RightMark / Full (probe side)	Probe-driven, emitted inline wherever the row is processed (pass 1 or cleanup); Bloom-negative rows emit their null-padded result immediately in pass 1
Null-aware anti; null_equality	build_side_has_null and bounds accumulate during collection before routing (all rows, spilled included); global probe-side atomics unchanged

6.7 Two interactions that deserve their own headlines

Dynamic filters — the downgrade happens at one well-defined moment. Dynamic filters are installed at plan time but populated once, at build completion — which is also exactly when the spill outcome is known. So the choice of pushdown strategy is made then, never mid-scan. Min/max bounds and InList variants are unaffected — they are accumulated over all rows, spilled included. Only the full-map membership variant is unavailable when something spilled (it needs the whole map); a per-bucket Bloom membership filter is its natural successor, since T7 builds those Blooms anyway. Worst case: queries that previously failed outright lose one pushdown variant.

Ordering — a plan-level trade, visible even without spilling. `maintains_input_order` currently advertises probe-side order for Inner/Right joins. Cleanup emission breaks it, so with the flag on the operator stops advertising it — and the planner may then insert an explicit sort downstream even for queries that never spill. Mitigations: the property change is tied to the flag (off ⇒ zero plan changes); T8 runs a TPC-H plan-diff audit with the flag on and the config docs call the trade out explicitly.

- **Cancellation & cleanup:** spill files are DiskManager tmp files (freed on drop); the cleanup loop yields between buckets, keeping cancellation responsive.
- **Disk quota:** writes go through SpillManager and respect max_temp_directory_size; quota exhaustion is the one remaining failure mode.

6.8 Memory accounting

- Register the build consumer `with_can_spill(true)` so FairSpillPool treats hash join like sort/agg.
- Reservation lifecycle: grow per buffered batch (as today) → shrink on destage → reserve `hash_join_spill_reservation_bytes` before map construction (default 16 MB — directly fixes §3's measured failure-at-table-allocation) → cleanup sizes restored buckets against the same reservation, prefetching bucket i+1 only when the budget allows.

6.9 Configuration and metrics

Option (datafusion.execution.*)	Default	Meaning
`enable_hash_join_spilling`	false	Master switch; the path to default-on is stated in T11
`hash_join_spill_bucket_count`	16	Buckets per level (power of two)
`hash_join_max_spill_recursion`	4	Depth cap before the chunked-replay skew fallback
`hash_join_spill_reservation_bytes`	16 MB	Headroom reserved for resident hash-table construction (mirrors sort_spill_reservation_bytes)

Metrics: standard `SpillMetrics` (spill_count, spilled_bytes, spilled_rows) plus `spilled_buckets` and `spill_recursion_depth`, in EXPLAIN ANALYZE like sort and SMJ.

6.10 Alternatives considered

Alternative	Why not (as the primary design)
Sort-based fallback (Spark SPARK-32634)	Gives up O(1) probes; SMJ already exists as the manual fallback — and §3 measures it at 3.5× on fitting joins
Grace up-front (ClickHouse grace_hash)	Not adaptive: pays partitioning+spill even when memory suffices, violating G1
Whole-build spill + probe replay (NLJ-style) as primary	Replay cost multiplies with build/memory ratio; retained only as the skew fallback where re-splitting provably cannot help
DuckDB-style shared global table + pointer swizzling	Built for a row-major unified buffer manager; a poor fit for Arrow columnar batches — and unnecessary given partition-locality (§6.1). Spilling whole RecordBatches and rebuilding maps on restore trades a little CPU for a large simplicity win
Symmetric / early probing (probe before build completes)	Only correct if probe rows are retained for late matches ⇒ buffers both sides — strictly worse memory; forfeits dynamic filters (§6.12)
Probe re-scan instead of probe spill	Unbounded (arbitrary probe subtree, re-run per restore group) and unsound if non-deterministic; opt-in follow-up for leaf scans (§6.13)
Velox-style global memory arbitration first	Valuable future direction, but a much larger core change; not required for a correct, useful spilling join

6.11 Sparse-match economics: Bloom-negative early resolution and role reversal

The worry: a build bucket spilled, but few probe rows actually match it — is full spill freight still paid? Two mechanisms say no. **(1) Bloom-negative early resolution.** Each destaged bucket gets a Bloom filter over its retained 64-bit hashes (~10 bits/row, ~1% false positives, zero re-hashing). Because routing is hash-disjoint, a Bloom-negative probe row provably matches nothing anywhere — it is finished in pass 1 for every join type: dropped (Inner/Semi), emitted null-padded immediately (Right/Full), or marking nothing (build-side-emission types). Probe spill volume collapses to roughly the true match volume. **(2) Role reversal.** At cleanup, if probe.Bi ≪ build.Bi, build the map from the small probe file and stream the large build file through it — never re-materializing the big side. Inner/Semi first; other types with inverted bitmap mechanics (a build row streaming through emits or marks inline, since partition-locality guarantees it matches nothing else).

6.12 Why probe starts after build — and what is overlapped instead

Probing before the build completes is only correct if probe rows are retained for late-arriving matches; retaining both sides is a symmetric hash join — strictly worse memory (DataFusion ships `SymmetricHashJoin` for streams, where range guarantees bound its state). Build-first is also a feature: the completed build populates dynamic filters that shrink the probe scan — the 25× mechanism. What is overlapped: cleanup prefetch of spilled pair i+1 while joining pair i (T9); a future “Grace-parallel” mode (partition both sides concurrently when heavy spill is predictable) is rejected for now — it materializes the probe side even when the build would have fit, and forfeits dynamic filters.

6.13 Probe spill vs. probe re-scan

The alternative to spilling deferred probe rows is re-executing the probe child once per restore group. Rejected as default: the probe child is an arbitrary subtree (unbounded, run repeatedly), and re-execution is only sound if deterministic — the same robustness argument behind NLJ's replay-from-spill. The exception worth a follow-up: when the probe child is a bare re-scannable leaf (Parquet scan), a re-scan can push a per-restore-group bucket-membership dynamic filter (hash-window predicate + Bloom, building on #19858) and read only relevant row groups — an opt-in `probe_side_strategy` later, not phase 1.

7. What it costs (rule of thumb)

- **Nothing spilled, nothing paid.** Same code path as today; with no memory limit configured, the spill machinery is structurally unreachable (Phase A never transitions).
- **Spilling: pay only for what didn't fit.** Every spilled build byte is written once and read back once (about 2× the spilled bytes); probe-side spill is only rows that might match a spilled bucket — with Bloom, close to the true match volume. CPU stays linear; peak memory stays under the budget by construction.
- **Capacity: ×16 per recursion level.** A 75 MB share covers a multi-TB build side within 3–4 levels. All-equal-key skew falls back to chunked build + probe replay — slower but unbounded (an equal-key join's output is inherently quadratic anyway).
- **Versus today's workaround:** SMJ sorts and spills both complete sides no matter how close the build was to fitting — and §3 measures its 3.5× tax on fitting joins. Hybrid hash pays proportionally to the miss: in the §3 experiment (~3/4 of the build missing) that is ~3 GB of sequential, compressible spill traffic instead of query failure — and zero when it fits.
- **One accounting decision, made:** spilled rows carry their 8-byte join hash (~10–15% extra spill volume) so restore, routing, and Bloom construction never re-hash.

8. Prior work

The chronological table of prior asks and landings (#1599 '22 → #12952 '24 → SMJ spill landed → #17267 '25 unimplemented → spill infra landed → NLJ spill landed in DF 54) lives in the GitHub epic, which is the tracking document; this design doc does not duplicate it.

9. Implementation plan — one reviewable PR per task

Ordered so each PR is individually mergeable and leaves main releasable. Sizes: S < ~300, M ≈ 300–800, L ≈ 800–1500 net LoC.

ID	Deliverable	Size	Scope & acceptance
T0	Evidence + harness	S	Memory-limited join benchmark + budgeted-RuntimeEnv test utility; encodes §3's failure matrix (incl. the 3.5× SMJ-tax row) as the baseline
T1	Refactor collect_left_input → BuildSideBuffer	M	Pure mechanical extraction; acceptance: benches within noise of main
T2	Bucket-splitting kernel	S/M	bucket_indices(hashes, bit_window) + take()-based splitter with criterion benchmarks — puts a number on the “dual take” concern before the core lands

ID	Deliverable	Size	Scope & acceptance
T3	BucketedBuildBuffer (build-side spill)	M	16 buckets, Phase-A→B transition, destage-largest, SpillManager files, with_can_spill(true); flag-gated; unit tests with a tiny FairSpillPool
T4	Core: INNER equi-join end-to-end, single level	L	Probe routing + probe spill + RestoreSpilledBucket/ProcessSpilledProbe states. Acceptance: §3's failing join completes under 300 MB in ≤ 2× the SMJ-workaround time (≤ ~7.5 s); no-spill join benches within 2% of main; sqllogictests under forced tiny budgets
T5	Full join-type matrix	L	Per-bucket bitmaps; Right/Full early emission; null-aware anti; join fuzzer gains a spill-forcing mode (DuckDB force_external trick)
T6	Recursion + skew fallback	M	Bit-window recursion, depth cap, no-progress detection, chunked build + ReplayableStreamSource probe replay
T7	Bloom-negative early resolution + role reversal	M	Per-spilled-bucket Blooms from retained hashes; pass-1 resolution for all join types; role reversal for Inner/Semi; probe-spill-volume benchmarks
T8	Planner + property integration	S/M	CollectLeft-steering rule when flag on (§5.1); drop probe-side maintains_input_order under the flag; membership→bounds/InList downgrade; TPC-H plan-diff audit with the flag on
T9	Memory-model polish	M	hash_join_spill_reservation_bytes; victim policy; cancellation-safe temp-file cleanup; prefetch of spilled pair i+1
T10	Observability + docs	S	Metrics in EXPLAIN ANALYZE; user-guide memory-limited-execution section; degradation-curve write-up
T11	CI + the path to default-on	M	TPC-H SF1 under stepped budgets (4 GB → 256 MB) in extended tests. Flip criteria, stated up front: force-spill suite green two consecutive releases; fuzzer clean; spilled hybrid ≤ SMJ workaround on the benchmark matrix; no-spill regression ≤ 2% — proposed and decided on the tracking issue

Dependency order: T0–T2 parallelizable; T3 → T4 → {T5, T6, T7} in parallel → T8–T11. T4 is the review-heavy centerpiece; everything around it is shaped to keep T4 as small as an end-to-end change can be.

10. Testing and verification

- **Correctness matrix:** sqllogictest parameterized over {JoinType} × {filter} × {null_equality} × {spill forced / not}, with the in-memory path as oracle.
- **Force-spill mode:** a test-only knob (à la DuckDB force_external) sets the destage watermark to ~0 so every existing join test also exercises the external path — correctness for free from the whole current suite.
- **Fuzzing:** join fuzzer gains random budgets + spill forcing; differential-check spilled vs unspilled results.

- **Plan audit:** TPC-H plan-diff with the flag on (T8) to surface any sort insertions from the dropped ordering property before users do.
- **Memory-budget CI:** TPC-H SF1 under stepped budgets; assert zero ResourcesExhausted at every step; track wall-clock so the degradation curve is visible over time.

11. Risks and mitigations

Risk	Mitigation
Probe-side routing (take) regresses the fast path	Routing exists only in Phase B (§6.2); T2 benchmarks the kernel; T4 gates merge on ≤2% end-to-end
Join-type semantics bugs on the spilled path	T5 matrix + force-spill over the whole suite + fuzzer differential checks; ClickHouse's INNER/LEFT-only history is the cautionary tale
Dropped ordering property causes silent plan regressions when the flag is on	Property change tied to the flag; T8 TPC-H plan-diff audit; documented in the config
Identical-key skew defeats bucketing	No-progress detection + chunked-build/replay fallback (T6); NLJ precedent in-tree
FairSpillPool starvation between partition streams	with_can_spill(true) + watermark relative to the consumer's share (T9)
Spill file count (2 files × spilled buckets × partitions)	Only spilled buckets create files; pairs deleted when processed; bytes bounded by disk quota; SpillPool rotation available if needed (Q3)
Review bandwidth (the #21036 concern)	Twelve small PRs with benchmarks attached; T1/T2 mechanical; a supervising-maintainer ask accompanies the epic

12. Open questions

- Q1 — Bucket count: 16 per level vs larger fan-out; routing cost and file count vs recursion depth (T2/T4 benchmarks decide).
- Q2 — When to start spilling: only when an allocation is actually refused (try_grow fails), or slightly earlier — e.g. once the join has used ~80% of the memory it can expect to get — so there is still headroom for the split work and the hash-table build? (The early trigger is only meaningful under FairSpillPool, where each consumer has a defined share; under GreedyMemoryPool the failure reaction is the trigger either way.)
- Q3 — Probe spill files: plain per-bucket InProgressSpillFile (phase 1 proposal) vs SpillPool with rotation for very large probes.

13. References

- DataFusion: #17267 (adopted and extended) · #12952 · #1599 · #9359 (SMJ spill) · #14078 · #15512/#19858 · DF 54 release notes (spilling NLJ) · source: `datafusion/physical-plan/src/{joins/hash_join, joins/nested_loop_join.rs, spill}`
- DuckDB: PR duckdb/duckdb#4189; Kuiper, Groß, Boncz, Mühleisen, “Saving Private Hash Join”, VLDB 2025.

- Velox: “Spilling”, facebookincubator.github.io/velox/develop/spilling.html · ClickHouse: “Joins Under the Hood” pt. 2 · Spark: SPARK-32634 · Trino: “Spill to disk” docs.
- Literature: Kitsuregawa et al. 1983 (GRACE); DeWitt et al. SIGMOD 1984, Shapiro 1986 (hybrid hash); Graefe et al. VLDB 1998 (dynamic destaging, role reversal, hash teams).

A. Appendix — experiment script (datafusion-python 54.0.0)

```
from datafusion import SessionContext, SessionConfig, RuntimeEnvBuilder
MB = 1024*1024
rt  = RuntimeEnvBuilder().with_fair_spill_pool(300*MB).with_disk_manager_os()
cfg = (SessionConfig().with_target_partitions(4)
       .set('datafusion.optimizer.prefer_hash_join', 'true'))  # 'false' → SMJ
ctx = SessionContext(cfg, rt)
# one-time: COPY (SELECT v AS k, concat('payload-', v, '-', repeat('x', 24)) AS payload
#            FROM (SELECT unnest(generate_series(1, 20000000)) AS v)) TO 't.parquet'
ctx.register_parquet('t_build', 't.parquet'); ctx.register_parquet('t_probe', 't.parquet')
ctx.sql('SELECT count(*) FROM t_probe p JOIN t_build b ON p.k = b.k').collect()
# → Resources exhausted: Failed to allocate additional 95.4 MB for HashJoinInput[3] ...
# Fitting case (build k<=10,000,000): hash join 0.70 s; prefer_hash_join=false: 2.48 s.
# EXPLAIN ANALYZE on the SMJ plan reports SortExec spill_count=16 / 155.3 MB.
```

B. Appendix — stream state machine delta

```
WaitBuildSide
  -> FetchProbeBatch <-> ProcessProbeBatch        (loop; today's states)
  -> ExhaustedProbeSide
       -> Completed                               (no bucket spilled: today's path)
       -> RestoreSpilledBucket(i)                 (new states; only when the
            -> ProcessSpilledProbe(i)              spilled set is non-empty)
            -> RestoreSpilledBucket(i+1) ...
       -> Completed

RestoreSpilledBucket: read build.Bi; fits -> build map; else re-split with the
next 4-bit hash window (max 4 levels), or chunked-build + probe replay when the
bucket stops shrinking (all-equal keys).
```