RadixAttention and Prefix Reuse

Deep Dive into SGLang — Chapter 8

Yin Yang

Foundation Books Project

Where this chapter sits

This is the chapter that turns "the same prompt prefix" into "reusable runtime state."

- Chapter 7 gave us the request row and the KV slot map.
- Chapter 2 gave us the prefix hit as a serving idea.
- Here we make it executable: when may a second request *skip* prefill over a shared prefix?

Goal: reuse is legal only when a matched token prefix still names live, protected KV slots — logical identity and physical lifetime must agree.

Roadmap

The reuse problem

Identity, geometry, and the radix tree

Match, project, execute

Publishing, locks, and eviction

Honesty: invariants and measurement

The reuse problem

Textual sameness is not executable reuse

Two requests can start with exactly the same tokens and still fail to reuse work.

- The earlier request's KV pages may have been *evicted*, moved, or released.
- A prefix can match perfectly while its KV state is gone or owned by someone else.

The running prompt pair (compressed)
$A = r_{\text{pan}} = [11, 12, 13, 91], \quad B = r_{\text{waf}} = [11, 12, 13, 77], \quad C = r^{+}_{\text{pan}} = [11, 12, 13, 91, 5].$

Prefix reuse is a runtime proof about identity *and* lifetime — not a string comparison.

Identity, geometry, and the radix tree

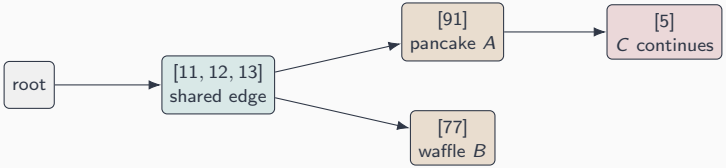
Identity is more than token equality

Definition (Prefix identity)
The cache namespace of a request prefix: ordered tokens, logical positions, and namespace metadata that decide whether two prefixes may share KV state:

$$I(r, h) = (x_{r,0:h},\ 0{:}h,\ \eta_r).$$

- Same token span, different namespace η (LoRA id, cache salt) — must miss.
- RoPE positions matter: a shifted prefix rotates keys differently.
- The key $K(I(r, h))$ is the lookup object; the tree node holds the KV indices.

The radix tree: one edge per shared run

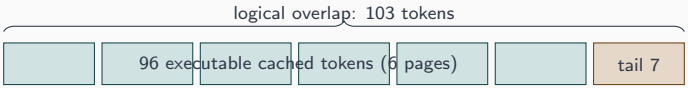


A compressed radix tree stores maximal one-child runs as one edge; branching happens only where cached sequences diverge. 1,000 requests sharing a 2,000-token prompt store it *once*, not a thousand times.

Page geometry decides the executable hit

Definition (Page-aligned reusable prefix)
For page size P, a logical hit of length h has executable reusable length

$$h^{\text{page}} = P\lfloor h/P \rfloor.$$



A 103-token textual match at $P{=}16$ is a 96-token tree hit; the 7-token tail stays request-local until a later page-compatible transition.

Match, project, execute

One prefix hit, several representations

Stage	What the hit becomes
Tree match	longest legal key match $\rightarrow$ KV indices V, owner ν
Request state	cached-prefix vector, protected length, suffix length u_r
Batch	suffix-only input tensor + output cache locations
Attention	a forward batch whose slot map already holds the prefix

For prompt length L_r and page-aligned hit h_r^{page}, the prefill step accounts for $u_r = L_r - h_r^{\mathrm{page}}$ new prompt tokens. Reuse is decided at scheduling time; attention only consumes it.

From concept to code: match becomes request state

The scheduler turns one match result into request-row fields.

from `python/sglang/srt/managers/schedule_policy.py`

```
match_result = tree_cache.match_prefix(
    MatchPrefixParams(
        key=RadixKey(token_ids=token_ids, extra_key=req.extra_key), ...))
(req.prefix_indices, req.last_node,
 req.last_host_node, req.host_hit_length) = (
    match_result.device_indices, match_result.last_device_node,
    match_result.last_host_node, match_result.host_hit_length)
```

The projection tuple P_r is literal: matched `device_indices` become `prefix_indices`; the terminal node becomes `last_node`; the namespace tag is `extra_key`.

Publishing, locks, and eviction

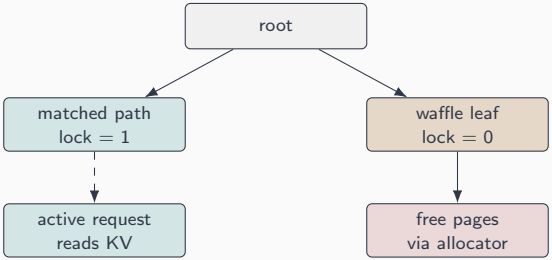
Publication is an allocator operation, too

Lookup asks "may I reuse?" Publication asks "which new KV is stable enough to become tree-owned?"

- A finished request inserts its page-aligned committed prefix.
- It frees duplicate slots the tree already owns, frees the unaligned tail, and releases its lock.
- An unfinished chunk inserts, *rematches*, and rewrites its row with the tree's authoritative indices.

The rematch is the non-obvious round trip: after a split or deduplication, the producer's local slots are no longer authoritative.

Locks decide safety; policy decides priority



Eviction may free only *unlocked* leaves: $\{\nu : \mathrm{ref}(\nu) = 0\}$. A positive `lock_ref` on a node or ancestor protects its prefix. Policy (LRU, LFU, FIFO, priority) ranks candidates; the lock rule is policy-independent.

Honesty: invariants and measurement

The one invariant the chapter defends

Prefix-cache alignment
A reuse decision is valid only when prefix identity, scheduler mode, logical position, page geometry, lifetime, batch row, mutation event, and suffix accounting all describe *the same logical prefix and the same physical KV state*.

Why failures show up far from the cache: a position swap or a freed slot can keep the tensor *shape* valid while corrupting the causal history the model reads.

Reuse is measured, not asserted

- **Cache events first:** stored/removed block events with parent-hash lineage make a hit auditable.
- **Direct trace, not aggregates:** join request ids with cache events to report hit geometry, protected tokens, and eviction outcome.
- **Thor/Qwen evidence:** long shared prefix $\rightarrow$ large hit; tighter pressure $\rightarrow$ measured evictions.

A positive result is lower realized suffix work; a negative result is saved work disappearing through eviction or recomputation — not just latency.

What to carry forward

1. Prefix reuse is a **runtime proof** about identity and KV lifetime, not a string match.
2. The **radix tree** owns matching, splitting, locks, insertion, and eviction; the attention layer only consumes the decision.
3. **Page geometry** sets the executable hit: $h^{\text{page}} = P\lfloor h/P \rfloor$.
4. **Locks** protect live prefixes; eviction policy ranks only unlocked leaves.
5. Every concept has a **home in the SGLang source** — `radix_cache.py`, `base_prefix_cache.py`, `schedule_policy.py`.

Next: Chapter 9 lets KV pages leave the device — hierarchical caching keeps the same identity/lifetime contract across storage tiers.