Cheaper Attention on KV Cache

1. Introduction

Attention is the core operation of every large language model. At each step a token looks back at all earlier tokens and decides which ones matter. This flexibility has a price: to look back, the model must store a key and a value vector for every past token in every layer (the *KV cache*) and read it again for each new token. When context windows were 2K–4K tokens this cost was small next to the rest of the network. Today models read 128K tokens or more and serve many users at once. At that scale attention is the main bottleneck: it dominates the compute of long prompts and the memory traffic of generation.

Since 2019 dozens of designs have tried to make attention cheaper. They look unrelated at first: shared heads, latent caches, sliding windows, sparse selectors, cache eviction, low-bit storage. This essay argues that most of them answer one question: **which factor of the attention cost do they shrink?** We first write the cost down. Then we go through the methods that shrink the key–value side, one factor at a time, and for each ask what problem it solves and what it leaves behind.

2. The cost of attention

2.1 The formula

For one head, the output at position t is a weighted average of past values:

$$o_t = \sum_{j \le t} \mathrm{softmax}_j \left(\frac{q_t^\top k_j}{\sqrt{d_h}} \right) v_j$$

A layer has H_q query heads and H_{kv} key–value heads, each of width d_h. A model has L layers. Everything below follows from one fact: to compute o_t, the model needs k_j and v_j for every earlier token j.

2.2 Prefill: quadratic compute

When the model reads a prompt of N tokens, all queries are processed together. Every query meets every earlier key, so the score matrix is $N \times N$:

$$\mathrm{FLOP}_{\mathrm{prefill}} \approx 2N^2 H_q d_h \cdot L$$

2.3 Decode: linear memory traffic for every token

During generation the model produces one token at a time. The new query must read every stored key and value:

$$\text{bytes per step} \approx N \cdot L \cdot 2H_{kv}d_h \cdot s + \text{weights}$$

Here s is bytes per number.

This cost is only linear in N, but it is paid again for every generated token, and every request has its own cache. For Llama-3-8B at 128K tokens one decode step reads 16 GiB of KV cache, more than the 15 GiB of model weights.

Besides, the formula above assumes the query reads every record. Some methods about "sparse read" will let each query read only part of the cache. This lowers the bytes *read* per step, but not the bytes *stored*: a record skipped by this query may be needed by a later one, so it must stay in memory.

So, in my point of view, decoding therefore has two costs that should be counted separately:

$$\text{stored} = N \cdot L \cdot 2H_{kv}d_h \cdot s \qquad \text{read per step} = N_{\text{read}} \cdot L \cdot 2H_{kv}d_h \cdot s, \quad N_{\text{read}} \le N$$

Stored bytes decide how long a context and how many requests fit in GPU memory. Read bytes decide how long each step takes. A method that only lowers N_{read} makes each step faster but leaves the memory footprint unchanged; only removing or merging records like methods about "compress sequence kV cache" shrinks what is stored.

2.4 Roadmap

Both decode costs are products of the same kind of factors. They differ only in the first one:

$$\text{stored} \approx \underbrace{N}_{\text{records kept}} \times \underbrace{L_{\text{cache}}}_{\text{layers with a cache}} \times \underbrace{w}_{\text{record width}} \times \underbrace{s}_{\text{bytes per number}}$$

$$\text{read per step} \approx \underbrace{N_{\text{read}}}_{\text{records read}} \times L_{\text{cache}} \times w \times s$$

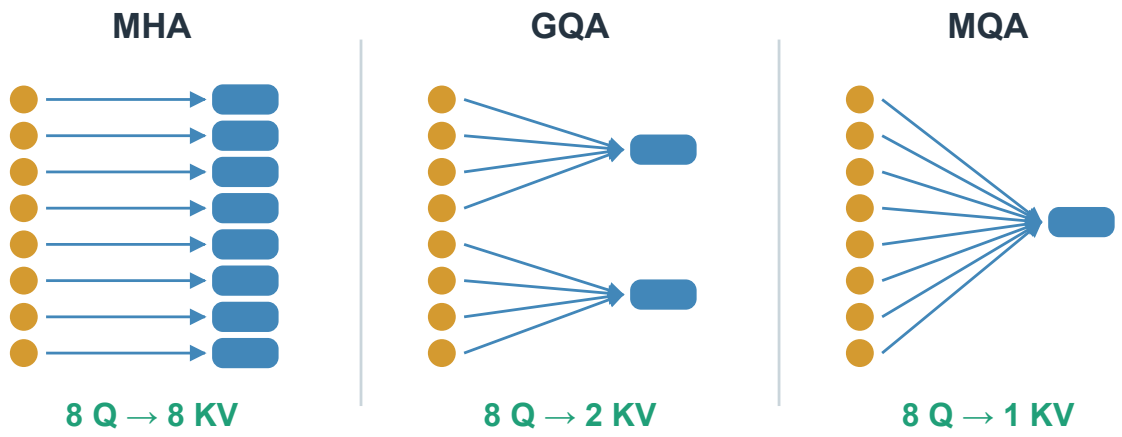
Each section of Part 3 makes one factor smaller. A method that shrinks w, L_{cache} or s lowers both costs at once. A method that only shrinks N_{read} makes steps faster but leaves memory unchanged.

Factor	Section	Main methods
w (record width)	3.1	MQA, GQA, MLA, K=V
L_{cache}	3.2	cross-layer sharing
N_{read}	3.3	sliding windows, sparse reads
N (records kept)	3.4	eviction, compression
s	3.5	FP8 / FP4 KV cache

3. Shrinking the key–value side

3.1 Thinner records: w

This part talks about length of KV cache for every **token**. Since there are many heads inside attention, it's very intuitive to make heads of query to share some heads in Key and Value.



In standard multi-head attention, every query head has its own key and value head.

Multi-query attention (MQA, Shazeer 2019) lets all query heads share a single key–value head. Each head still computes its own attention pattern, but the cache shrinks by a factor of H_q.

The price is quality, because one shared key–value view is too narrow for many heads.

Grouped-query attention (GQA, 2023) is not so aggressive: groups of G query heads share one key–value head. For example, Llama-3-70B uses 64 query heads and 8 KV heads, an 8× saving with quality close to MHA. An existing MHA model can be converted by averaging its key–value heads and training for some steps more.

Note that GQA still stores full-width vectors and sharing heads will definitely more or less affect the quality.

Multi-head latent attention

Then deepseek makes more insightful optimizations following above methods.

1. The key and value are all about the two aspects of one token. So perhaps they can be combined to one representation and converted to the correct form when used.
2. Sharing heads will more or less lose something. Therefore, for each head, specific representation can be computed at runtime.
3. Inside KV cache, only the compressed latent should be stored, in the size of Nd_l.

With above methods, some memory costs are reduced at the price of more computation.

Multi-head latent attention (MLA, DeepSeek-V2, 2024) stores one compressed vector per token instead: a 512-number latent plus a small 64-number positional key, 576 numbers per layer. An equivalent full MHA cache would hold 40,960. At decode time the up-projection matrices are folded into the query and output, so the model attends directly over the latents.

Here's an interesting thing about the RoPE in MLA.

Only Q and K need rotation in RoPE, but we compress K and V inside one latent space now. Since the K should be positional embedded first and then goes into KV cache. So confliction comes. There's no way embedding positions in advance into the latent and query after rotation can do the GEMM directly because that will affect V at the same time.

However, doing the rotation every time when the K is computed is too expensive since it will be $O(N^2)$.

Deepseek solves the problem by:

1. Let latent works without positional embedding. But split the a dimension of Q and K to do the rotation. The K here is also stored inside one separate cache.
2. Once a two token comes, its query will splitted into two parts and compute the score with two different K. Finally, sum them up to get the final score in softmax.

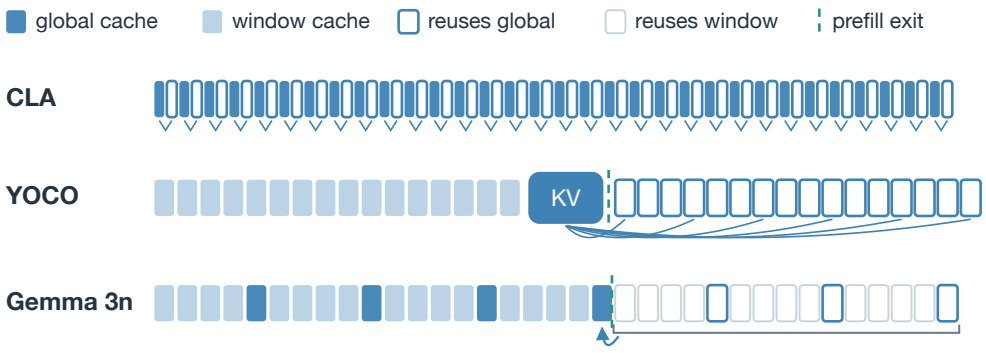
Analyzing the problem, one would find the skill for doing RoPE for part of Key is not unique in this scenario. Actually, one could always choose part of K for rotation and left remain unchanged. Then splitting the query into two parts, repeating the process above. Surprisingly, that's the basic idea of partial RoPE.

Then in some models, **K=V tying** (Gemma 4, DeepSeek-V4), uses one stored vector as both key and value, reducing the size of KV cache further.

Problems left over: the records are thinner, but there is still one record per token, per layer.

3.2 Fewer layers own a cache: L_{cache}

This part talks about the number of KV caches along the **depth** of the model. Section 3.1 let query heads share key–value heads inside one layer. The same idea can be turned sideways: let several layers share one cache, which is also quite intuitive.



In MLA, every layer computes and stores its own keys and values, so the cache is multiplied by L. But neighbouring layers see almost the same token representations, and their keys and values are often similar during observations.

Cross-layer attention (CLA, 2024) lets a layer reuse the cache of the previous layers instead of computing its own. With one owner for every two layers, storage halves. It also combines with GQA, since one factor shrinks the heads and the other shrinks the layers.

YOCO ("You Only Cache Once", 2024) pushes this to the extreme:

1. The lower half of the network uses cheap layers with small caches. In the paper, it provides two options: sliding-window attention or the fixed-size running memories with the gated retention mechanism.
2. At the middle position, which is exactly the boundary of two splitted parts about whether to compute K and V, the model produces one global key–value cache, for all previous tokens.
3. Every layer in the upper half reads that same cache, so the upper half stores almost nothing.

Side effect for prefill: The layers that only reuse a cache produce no keys and values of their own. So when the model reads a prompt for profill, it only needs to run the prompt up to the last layer that owns a cache. The upper layers are needed only for the tokens that are actually generated. That is why cross-layer sharing speeds up prefill as well, not only decode.

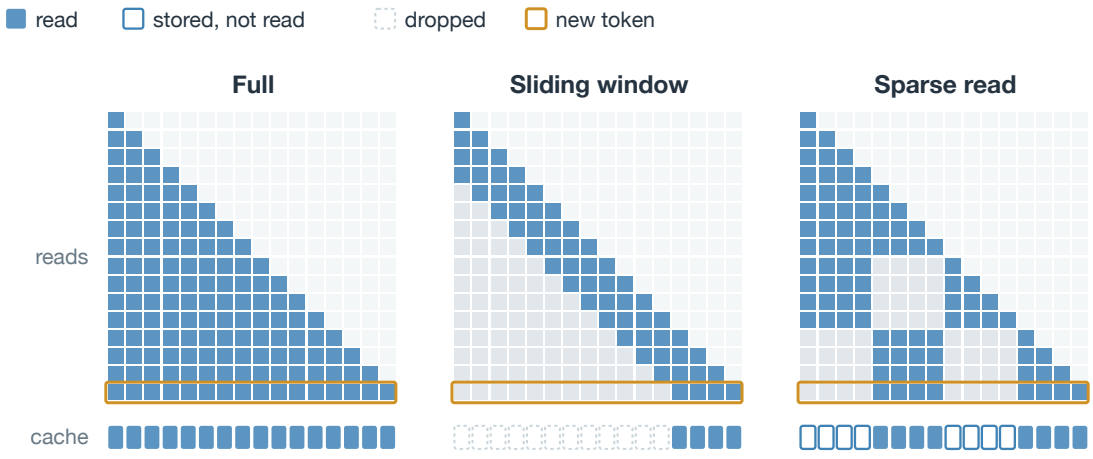
The same idea now appears in production: in Gemma 3n, only 20 of its 35 layers own a cache, and the rest reuse one of them.

Note that sharing must be built in during training. The reusing layers have no key and value projections at all, so an existing model cannot simply drop its upper caches.

Problems left over: storage shrinks, but reads do not: every layer that shares a cache still reads all of it at each step. And there is still one record per token, so the cache keeps growing with N.

3.3 Read fewer records: N_{read}

This part talks about how many records each new token **reads** along the **sequence**. Sections 3.1 and 3.2 made every record cheaper, but a new token still reads all N of them at every step. Yet most tokens only need a small part of the history, so it's quite intuitive to let each query read only that part.



There are two ways to choose that part: fix it in advance by **position**, or let each query choose by **content**.

Sliding window attention (SWA) is the simplest choice by position: a layer only looks at the last W tokens. A record older than W will never be read by this layer again, so it can be thrown away. In practice the cache is a ring buffer: token t is written into slot $t \bmod W$, over the oldest

record. So the layer's cache stops growing at W, and this is the one method in this part that also shrinks what is stored.

Consider the implementation of its KV cache. Orders won't matter if the positional embeddings are done to the right position of K and KV positions match, so order shift caused by modulo operation.

The price is clear: a window layer cannot look further back than W tokens. Therefore,modern models always mix many window layers with a few full layers.

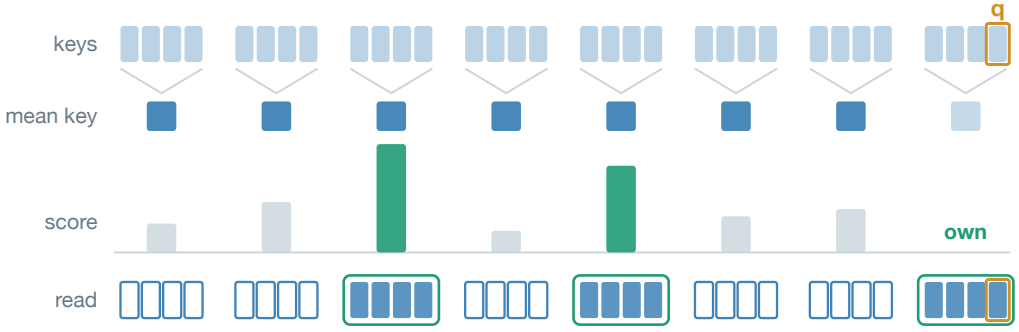
About the reach of windows: Information also flows upward through the layers. The output of a window layer at token t already mixes the W tokens before it. When the next window layer reads token $t - W + 1$, it also gets a summary of the W tokens before that one. So two window layers reach about $2W$ back, and L layers about $L \cdot W$: with 32 layers and $W = 4096$ that is about 131K tokens (Mistral 7B). But this reach is indirect, squeezed through the hidden states, so exact recall of a far-away detail is weak. That is why the few full layers are still needed, and why they now set the cache bill: only they keep growing with N.

Sparse reads choose by content instead. They keep the whole cache, and they all follow the workflow:

1. A cheap scorer estimates which past tokens matter for the current query.
2. Only the top few are picked.
3. Exact attention runs on those few.

The methods differ mainly in the scorer.

MoBA (Mixture of Block Attention, Moonshot, 2025) borrows the idea of mixture-of-experts: the blocks of the cache play the role of experts, and each query is routed to only a few of them.



1. Split the past tokens into blocks of B tokens, for example $B = 512$.
2. Give each block one summary key: the mean of its keys, $\bar{k}_b$.
3. Score each block with a single dot product, $s_b = q_t^\top \bar{k}_b$.
4. Keep the best few blocks, for example the top 3. The query's own block is always one of them, and inside it the query only sees earlier tokens. Blocks in the future are never candidates.
5. Run normal attention over the tokens of the chosen blocks only.

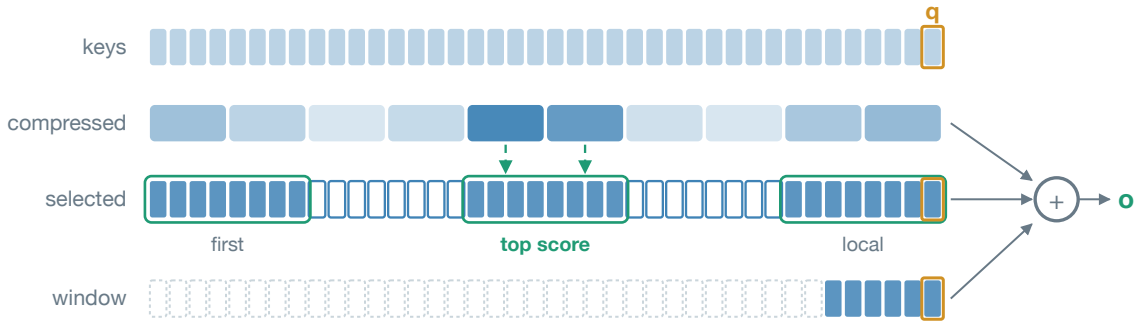
About the mean key: A dot product is linear, so comparing the query with the mean key is the same as averaging its scores with every key in the block:

$$q_t^\top \bar{k}_b = q_t^\top \frac{1}{B} \sum_{j \in b} k_j = \frac{1}{B} \sum_{j \in b} q_t^\top k_j$$

So one dot product tells the query how relevant a block is *on average*, without looking at its keys one by one. The scorer also has no weights of its own. The model is simply trained with MoBA, and its queries and keys learn to work with the mean.

The price comes from the same average: a block with one very relevant token among many irrelevant ones gets a low mean and may not be picked. The larger the block, the easier one sharp key is hidden. Also, full memory has to be stored though.

NSA (Native Sparse Attention, DeepSeek, 2025) reads the cache the way like one reads a long book: skim a summary of every chapter, read a few chapters in full, and always reread the last page. Each query gets these three views as three branches.



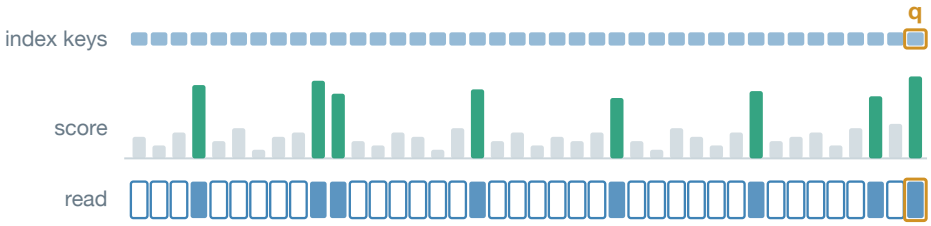
1. **Compressed:** cut the keys into blocks of 32 tokens (sliding by 16, so the blocks overlap) and squeeze each block into one key and one value with a small learned network. The query attends over all of these coarse records.
2. **Selected:** cut the keys again into larger blocks of 64 tokens. Keep 16 of them: the first block and the 2 most recent blocks always, plus the 13 that scored highest in the compressed branch. The query attends over their tokens in full detail.
3. **Window:** the query also attends over the last 512 tokens.
4. A small gate, computed from the current token, weights the three results and adds them up.

NSA has no separate scorer at all. To attend over the compressed records, branch 1 already computes how much the query cares about each block. Those attention weights are added up per 64-token block and reused as the scores for branch 2, so choosing costs almost nothing extra. The coarse view is both part of the answer and the map of where to look closely. Compared with MoBA, the block summary is also learned instead of a plain mean.

Also, all the query heads in one GQA group choose the same blocks. So each chosen block is loaded from memory once and used by all 16 heads of the group, which keeps the GPU busy. Because it is sparse from the start of training ("native"), it speeds up training too: 9× faster forward at 64K than FlashAttention-2.

The price is complexity: each branch has its own key and value projections, so there are more weights to learn and a little more to store.

DSA (DeepSeek Sparse Attention, DeepSeek-V3.2, 2025) drops blocks and picks single tokens. It adds a small **lightning indexer** to every layer.



1. Every token stores one extra small key for the indexer, k_s^I: 128 numbers in FP8.
2. The current token makes 64 small index queries $q_{t,j}^I$, each with a weight $w_{t,j}$.
3. Score every past token s:

$$I_{t,s} = \sum_{j=1}^{64} w_{t,j} \ \mathrm{ReLU}\left(q_{t,j}^I \cdot k_s^I\right)$$

4. Keep the 2,048 tokens with the highest scores. All 128 attention heads share this one set.
5. Run the main attention (MLA) over those 2,048 tokens only.

Why DSA can pick single tokens while MoBA and NSA pick blocks? GPUs like to load memory in contiguous blocks, and scattered tokens are normally slow to gather. But with MLA, all 128 heads share one 576-number latent per token. So one gather of a token serves every head, and one set of 2,048 tokens for all heads is enough. Token-level picking fits MLA.

Like most sparse methods, DSA is trained with the model: the indexer is first trained alone to copy the full attention pattern, and only then does the model switch to sparse reads.

The price is the indexer itself: it is cheap per token, but it still scores every token, which comes back in the problems left over below.

MSA (MiniMax Sparse Attention, MiniMax-M3, 2026) combines the two ideas above. Like MoBA, it reads whole blocks: each query picks 16 blocks of 128 tokens, again 2,048 tokens in total. Like DSA, a small indexer scores every token, but with only one index head for each GQA group, so it does 16× fewer index dot products than DSA. A block's score is the **maximum** of its token scores instead of the mean, so one sharp key can no longer hide inside a block. MiniMax reports 28.4× less attention compute at 1M tokens.

Problems left over:

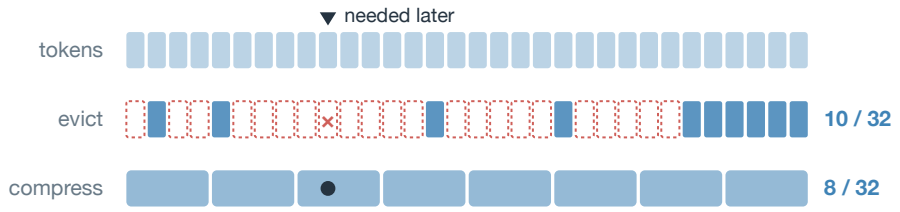
- the scorer still has to look at every key, so its cost grows with N. At longer context, DSA's indexer already costs more than the sparse attention it feeds.

- Sparse reads lower the bytes read, but not the bytes stored. A block skipped by this query may be the one the next query needs, so every record must stay in memory. DSA even adds a little to the cache, because the indexer keeps its own small keys.

To save memory, records must really be removed or merged.

3.4 Keep fewer records: N

This part talks about how many records are **kept / stored** in the cache along the **sequence**. Section 3.3 reads fewer records, but every record still stays in memory. Yet most records are rarely read again, so it's intuitive to either delete them (**eviction**) or merge several of them into one (**compression**).



The two differ in what is lost. Eviction keeps the surviving records exact, but the deleted ones are gone completely. Compression keeps something of every token, but only as a blurred summary.

Eviction has to guess, from what it has seen so far, which records future queries will need. The methods differ mainly in who makes that guess, and when.

H2O (Heavy-Hitter Oracle, 2023) guesses at every decode step, from the attention itself.

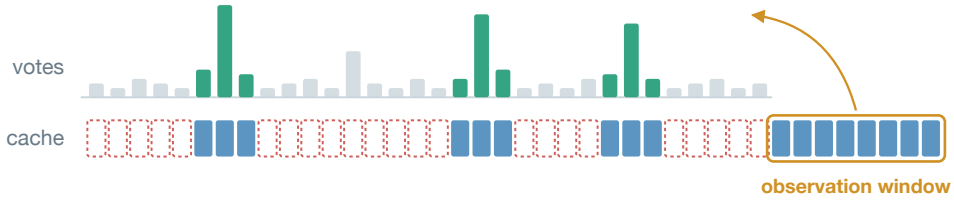


1. Give every record a running score, and after each step add the attention weight it just received: $S_j \leftarrow S_j + a_{t,j}$.
2. Fix a budget of records. Half of it goes to the records with the highest score (the "heavy hitters"), half to the most recent tokens.
3. When a new token arrives, the record that falls out of both groups is evicted, for good.

Attention is very uneven: a few tokens collect most of the attention, and they keep collecting it step after step. So the running sum is a good guess of what will matter next.

The price is that H2O needs the attention weights $a_{t,j}$ at every step, but fast fused kernels such as FlashAttention never write them out. Also, early tokens have been summed over the most steps, so the score is biased toward the start of the context.

SnapKV (2024) makes the guess only once, right after reading the prompt.



1. Take the last 32 tokens of the prompt as an "observation window". They usually contain the question.
2. Let them vote: for every earlier token, add up the attention it gets from the window.
3. Max-pool the votes over neighbouring positions.
4. Each head keeps its top positions plus the window, and the rest of the prompt's cache is dropped.

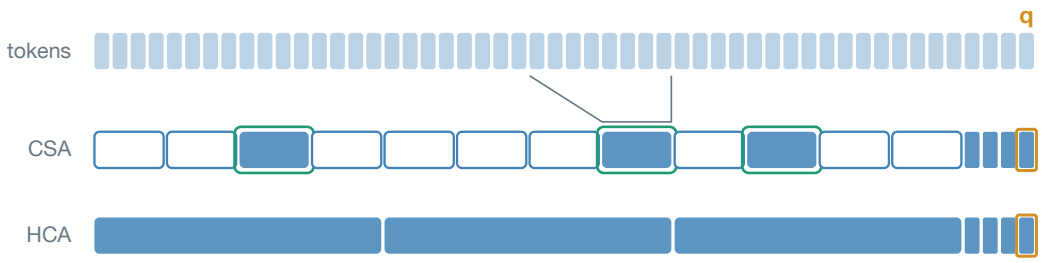
A fact is often several tokens long, like an order number "7314" that is split into a few tokens. Without pooling, only the token with the highest vote would survive. Pooling also keeps its neighbours, so the whole string survives together.

The price is that the voters are the current question. If a later question needs something else, it has already been deleted. Also, the tokens generated after the prompt are never pruned, so a long answer still grows the cache.

KVzip (2025) removes that bias toward one question. Instead of asking "what does this question need?", it asks "what would the model need to **repeat the whole context**?" It runs the prompt "Repeat the previous context:" once, scores every record by the most attention it gets during the repetition, and keeps the top records. Since no particular question voted, one pruned cache can serve many later questions. The authors report a 3–4× smaller cache and about 2× faster decoding. The price is one extra pass over the context.

Note that eviction cannot be undone. Unlike the sparse reads of 3.3, a deleted record is gone for every later query, so a wrong guess stays wrong.

Compression merges several neighbouring tokens into one learned record instead of deleting any. With a ratio m, the cache grows as N/m instead of N. The compressed branch of NSA in 3.3 already did this, but NSA kept the full cache for its other branches, so memory did not shrink. DeepSeek-V4 keeps only the compressed records, in two forms.



CSA (Compressed Sparse Attention, DeepSeek-V4, 2026) compresses lightly, then selects:

1. Every 4 tokens, emit one new entry. It pools 8 tokens, its own block of 4 and the previous one, so neighbouring entries overlap.
2. The pooling is learned: a softmax per channel, so each channel decides which of the 8 tokens to weigh most, instead of a plain mean.
3. A DSA-style indexer (3.3) picks the top 1,024 entries (in V4-Pro).
4. Attention reads those entries plus the last 128 raw tokens.

So the stored records drop to $N/4$, and each query reads about 1,152 entries however long the context is.

HCA (Heavily Compressed Attention) compresses heavily and reads everything:

1. Every 128 tokens become one entry, with no overlap.
2. Attention reads all entries plus the last 128 raw tokens. There is no scorer, so nothing can be missed by a wrong pick.

CSA keeps finer records but must choose among them. HCA sees the whole history, but only coarsely. DeepSeek-V4 alternates the two layer by layer, so some layers look closely at a few places and others glance at everything: the same "skim, then read closely" idea as NSA, now split across layers. Each entry also serves as both key and value (the K=V tying of 3.1).

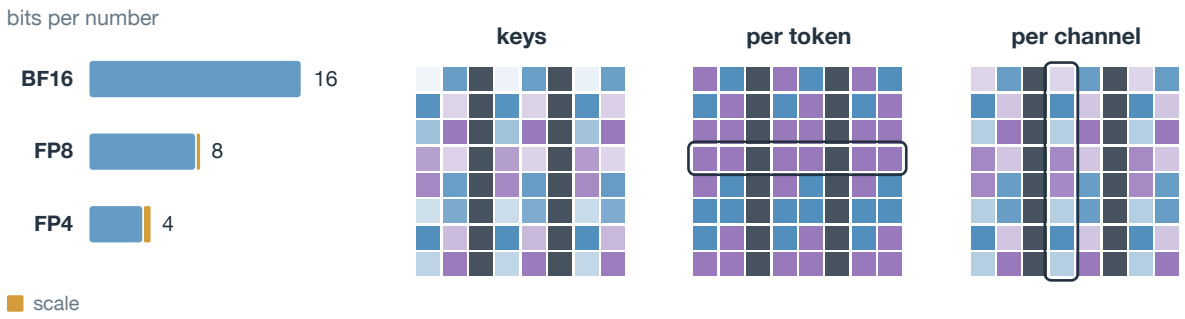
The price is detail: a fact that is one token long gets blended with its neighbours inside an entry. And the pooling is learned, so the model has to be pretrained with it.

Problems left over:

- Evicted records are gone for good, so a wrong guess can never be fixed later.
- Merged records lose detail.
- The number of records still grows with N. $N/4$ or $N/128$ is much smaller, but it is not fixed.

3.5 Fewer bytes per number: s

This part talks about how many **bytes** each number in the cache takes. Sections 3.1–3.4 cut how many numbers are stored and read, and s is the last factor in the formula. A model usually computes in BF16, 2 bytes per number. But a cached key or value does not need all of that precision to stay useful, so it's intuitive to store it with fewer bits as well.



1. Cut the cache into small groups, for example 16 to 128 numbers each.
2. For each group, store a step Δ and an offset z at higher precision, taken from the group's range: $\Delta = (\max - \min)/(2^b - 1)$ and $z = \min$.
3. Store each number as a b-bit code, round $\big((x - z)/\Delta\big)$.
4. When attention reads the cache, decode every number back: $\hat{x} = z + \Delta \cdot \text{code}$.

FP8 and **FP4** follow the same idea, but each code is a tiny floating-point number instead of an integer, so a group only needs a scale. FP8 halves the cache and FP4 quarters it. The real saving is a little smaller, because the scales take space too.

Note that quantization saves memory and bandwidth, not compute: the numbers are decoded back to higher precision before the matrix products. But decode is limited by the bytes it moves, so fewer bytes per number already means faster steps.

Below 8 bits, how the numbers are grouped starts to matter. **KIVI** (2024) goes down to 2 bits by grouping keys **per channel** and values **per token**, and it keeps the most recent tokens in full precision.

Why should keys and values be grouped differently? Keys have a few fixed channels with very large values (outliers), and values do not. In a group that runs along one token, an outlier stretches the range, so the step Δ becomes huge, and all the ordinary numbers of that token fall onto one or two codes. Their small differences, which are exactly what the attention scores depend on, are lost. Grouping per channel puts each outlier channel in a group of its own with its own step, so the ordinary channels keep a fine step. Values have no such channel pattern, so grouping them per token is fine. The recent tokens stay in full precision because a per-channel group needs a full set of tokens before it can be quantized.

There is one more catch for keys. The rotation that encodes position mixes pairs of channels differently at every position, which smears the outlier channels. **KVQuant** (2024) therefore quantizes the keys before the rotation and applies the rotation after decoding.

The price is error: every bit removed adds some, and the scales, the outlier handling and the decoding all cost extra space and work.

Problems left over: quantization shrinks every record by a constant factor, but it does not change how many records there are. The cache still grows with N.

3.6 What is still left

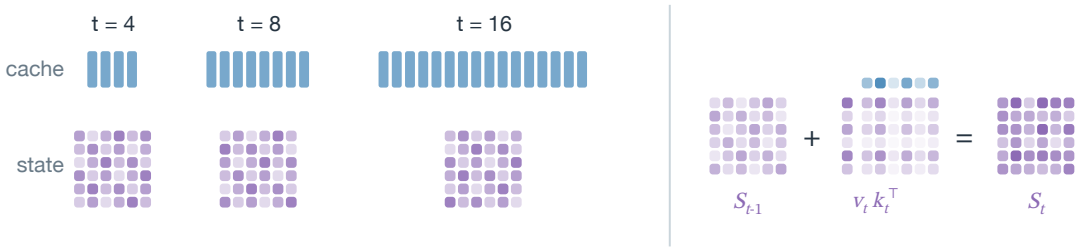
Combining these factors changes the bill enormously. Across 17 production models, the growing KV cache ranges from 516,096 bytes per token (Llama 3.1-405B, plain GQA) to 6,144 (Nemotron 3 Nano), and DeepSeek-V4.1-Flash reports 890. But every method in this part keeps a record for each past token. The cache still grows with N, however slowly. The next step is more radical: replace the growing cache with a **fixed-size state**. That is the subject of Part 4.

4. Replacing the cache with a fixed-size state

4.1 Linear attention: the history becomes one matrix

Records of methodx in Part 3 still grows with scale of prompts. The softmax forces this, because it needs every past score before it can weigh any value. **Linear attention** (2020) drops the softmax, and then the output can be regrouped:

$$o_t = \sum_{j \le t} v_j \, (k_j^\top q_t) = \Big(\sum_{j \le t} v_j k_j^\top \Big) q_t = S_t \, q_t, \qquad S_t = S_{t-1} + v_t k_t^\top$$



The whole history folds into one $d_v \times d_k$ **state** per head. Each new token is added to it as an outer product, and then its key and value can be thrown away. With a head width of 128, the state holds 16,384 numbers per head at any length, while a KV cache passes that size after only 64 tokens. For training and prefill, the sequence is cut into chunks: plain attention inside each chunk and the state carried between chunks, so the cost grows linearly with N.

Left over: the state only adds. Nothing is ever forgotten or overwritten, so memories pile up and blur.

4.2 Forget: decay the state

Let the state fade a little before each write:

$$S_t = \alpha_t \, S_{t-1} + v_t k_t^\top, \qquad 0 < \alpha_t < 1$$

RetNet (2023) uses one fixed decay per head, so old tokens fade at a set rate. **GLA** (2023) and **Mamba-2** (2024) make the decay depend on the input, so the model decides at each token how much to forget. GLA goes further and decays each key channel separately.

Left over: decay fades everything a little. It cannot remove one wrong memory and keep the rest.

4.3 Correct: the delta rule

Before writing, read what the state already returns for the new key, and write only the error:

$$S_t = S_{t-1} + \beta_t \big(v_t - S_{t-1} k_t \big) k_t^\top = S_{t-1} \big(I - \beta_t k_t k_t^\top \big) + \beta_t v_t k_t^\top$$

If the state already holds the right value for k_t, nothing changes. If it holds a stale one, that value is overwritten. This is **DeltaNet** (2021). **Gated DeltaNet** (2024) puts the decay of 4.2 in front, so the state forgets a little and then corrects. **KDA** (2025) makes that decay per channel.

Left over: the state still has a fixed size. It can hold only about d_k key–value pairs exactly, against N for a cache.

4.4 One view of all: the transition matrix

All these updates share one form:

$$S_t = S_{t-1} A_t + W_t$$

They differ only in the matrix A_t applied to the old state: I for linear attention, αI for RetNet (fixed) and $\alpha_t I$ for Mamba-2, $\mathrm{Diag}(\alpha_t)$ for GLA, and $\alpha_t(I - \beta_t k_t k_t^\top)$ for Gated DeltaNet.

Each update can also be read as one learning step. The linear-attention write is one gradient step on $-\langle Sk_t, v_t\rangle$. The delta rule is one gradient step on $\frac{1}{2}\|Sk_t - v_t\|^2$, so the state is trained during inference to map keys to values.

4.5 What is still left

A fixed state compresses the whole history, but it cannot look things up. A $d_v \times d_k$ state stores only about d_k pairs exactly, while a cache keeps all N. Tasks that need one exact detail from far back, such as a number to repeat or a passage to copy, still need attention over real records. So models mix the two: many cheap state layers and a few full-attention layers. These hybrids are the subject of Part 5.

5. Hybrids: mixing states and records

5.1 Why mix

A state is cheap but cannot look things up. A cache can look things up but grows with N. A **hybrid** stack keeps both: most layers carry a fixed state, and a few full-attention layers keep every record for exact lookup. The memory then has a fixed part and a growing part:

$$M(N) = \underbrace{\sum_{\ell \in \text{state}} |S_\ell|}_{\text{fixed}} + N \underbrace{\sum_{\ell \in \text{attention}} 2 H_{kv} d_h \, s}_{\text{per token}}$$

The fixed part matters only for short contexts. With one attention layer in four, the cache of the attention layers already outweighs all the states after a few thousand tokens. So at long context, the few attention layers set almost the whole bill.

5.2 How to mix

The common way is to **interleave** layers: state layers and full-attention layers alternate in a fixed ratio, spread evenly through the depth. Ratios range from one attention layer in 2 to one in 10,

and one in 4 is common with delta-rule states. Another way is **parallel heads**: inside each layer, some heads are attention and some are states, and their outputs are added. That keeps a cache in every layer, so it saves less memory.

Some examples:

- **Window + full attention:** Gemma 3 (2025, one full layer in 6), gpt-oss (2025, one in 2).
- **Mamba + attention:** Jamba (2024, one attention layer in 8), Nemotron-H (2025, about 8% attention).
- **Linear attention + softmax:** MiniMax-01 (2025, one softmax layer in 8).
- **Delta rule + attention:** Qwen3-Next and Kimi Linear (2025, both one attention layer in 4).
- **Parallel heads:** Hymba (2024), Falcon-H1 (2025).

Not every model goes this way: MiniMax-M2 (2025) went back to full attention in every layer.

The cheap layers can be any bounded kind: a sliding window (3.3), a decayed state (4.2) or a delta-rule state (4.3). And the few attention layers can still use the tricks of Part 3, such as latent records, sparse reads and low-bit storage.

However, which layers must stay full attention is still found mostly by trial, and it depends on the task.

5.3 Putting it together

This essay started from one cost: every new token reads a key and a value for every past token, in every layer. Written as a product, stored $\approx N \times L_{\text{cache}} \times w \times s$ and read per step $\approx N_{\text{read}} \times L_{\text{cache}} \times w \times s$, and almost every method shrinks one factor.

Fixed-size states remove the growth with N altogether, at the price of exact recall. Hybrids put a few full-attention layers back to recover it. New models combine several of these levers at once.

6. What's left over?

The essay focuses on how much attention should remember. However, how attention is computed or kept stable, like positions, softmax stability, kernels designs, remains in the future.