Caching

Subhash Suri

January 30, 2020

1 Memory Organization

- Computer scientists have spent a lot of time thinking about *memory management*: how to organize content for *efficient* access. In this lecture, we will study the principle of *caching*, which plays an important role in architecture of memory, and in everything from the layout of processor chips at the milli or micrometer scale to the geography of global internet.

- Caching creates *illusion of an infinitely large* fast memory, even though we only have a small finite amount of such memory.

1.1 Origins

- The story begins in 1946 when Arthur Burks, Herman Goldstine, and John von Neumann, working at Institute for Advanced Studies in Princeton, laid out a design manual for what they called an electrical "memory organ." The wrote:

  > *In an ideal world, the machine would have a limitless quantity of lightening fast storage, but in practice that was not possible (it still isn't).* So, instead, they proposed the next best thing: *a hierarchy of memories, each with bigger capacity than one before but less quickly accessible.*

- By building a pyramid of different forms of memory—from very small but very fast ones to larger but slower ones—they hoped to get the best of both. This *intrinsic* tradeoff between *size* and *speed* (and cost) remains true even today.

- *An Analogy.* The intuitive idea between combining small-and-fast with vast-but-slow has an analogy in how patrons of the (physical) library system operate: when working on a research project, we may borrow some books that we expect to reference frequently—they form our small but fast memory—and then periodically go back to

check out other books, while returning the old ones—with the whole library acting as our vast but slower memory.

- Some of these ideas were used in the design of 1962 supercomputer Atlas in Manchester. Later Cambridge mathematician Maurice Wilkes recognized an even more important aspect of memory hierarchy: smaller and faster memory wasn't just a convenient place to *work with data before saving it off* in the slower memory; it could also be used to

  > *deliberately hold onto pieces of information likely to be needed later, anticipating similar future requests, which could dramatically speed up the operation of the machine by avoiding the need to fetch them again from the slow memory.*

- In extending the analogy to the library, suppose we could make just one trip to the library and then spend rest of the week working at home as if every book in the library had already been available at our desk. The more trips we make back to the library, the slower our work and less effective working from home is.

- Wilkes's proposal was implemented in IBM 360/85 supercomputer in late 1960s, where it acquired the name *cache*, and since then caches have appeared everywhere in computer systems: processors, hard drives, operating systems, web browsers, servers.

1.2 Performance Bottleneck: Memory Wall

- One of the reasons caches figure so prominently is that while processor performance has followed the stunning exponential growth curve of Moore's law—number of transistors in CPUs doubling every two years—the memory performance has significantly lagged behind.

- Put differently, it means that relative to processing time, the memory *access cost* has increased exponentially! (It's like a factory whose manufacturing speed continues to double but the the number of parts shipped to it from overseas has stayed the same, creating a bottleneck.) *There is even a name for it: "memory wall."*

- Our best defense against hitting this wall has been an ever more elaborate cache hierarchy: *caches for caches for caches, all the way down!* Modern laptops, tablets, smart phones have upto six layer memory hierarchy.

- *Managing* this memory hierarchy, therefore, is an important problem. This lecture explores one of the most important questions in cache management—what to do when cache is full, namely, *cache replacement policy*, for a simple two-level memory system.

1.3 Algorithmic Decision Making: Eviction Policies

- As long as there is room in the cache, we can add more stuff to it, which does not require much forethought. It is lot trickier when the cache fills up and we need to decide what to evict to make room for the new stuff that needs to be brought in?
- Cache is only a *small fraction* of the size of the main memory, so sooner or later we need to make room to bring new stuff needed by the program.
- Algorithms that decide what to evict are called *replacement policies*, *eviction policies*, or simply as *caching algorithms.*
- IBM, which played an important role in deploying cache systems, also played an important role in early research on caching algorithms. Laszlo Belady, an IBM researcher, wrote one of the early and very influential papers on caching.
- Belady recognized that the goal of cache management is to *minimize* the number of times you can't find what you are looking for in the cache and, therefore, have to go to the slower main memory. These "misses" are called *page faults* or *cache misses.*
- Belady proved that the *optimal* cache eviction policy is this: *when the cache is full, evict the item whose next use is* farthest *in the future.*
- This policy is now called *Belady's Algorithm* as a tribute to his influential work. Of course, it requires *clairvoyance*, full knowledge of the future, which is impractical.
- An important observation is that unlike many problems in CS for which optimal answer is NP-hard to compute, in the case of caching, the optimal solution is easy to find *if we have all the data.*
- Thus, Belady's algorithm also helps distinguish between *online and offline* computational problems. Caching is an *online* problem—the algorithm must process a sequence of requests, which arrive one at a time, and the algorithm must satisfy each request before seeing the next one.
- In contrast, in problems such as shortest paths, spanning tree, or even TSP, the algorithm knows the entire input in advance. The difficulty there is purely computational, while in online the difficulty is both computational and *informational*: algorithm must make choices without knowing what the future requests will be.

- A huge number of *online* cache policies have been proposed and analyzed over the years, including the following:

    1. FIFO
    2. LIFO
    3. LRU
    4. LFU
    5. Randomized
    6. MIN (farthest in future)
    7. Many other variants

- LRU counts on temporal locality, which results in part from the way computers solve problems, for instance, by executing a loop that makes a rapid series of related reads and writes.

- There are many additional eviction algorithms that try to track all sorts of data usage patterns to improve the caching performance, and some of them can indeed do better than LRU under right conditions, LRU, perhaps with some its minor tweaks remains the overwhelming favorite at a variety of scale.

1.4 Caching at Internet Scale

- We tend to associate cache with our local computer or laptop, but the concept is just as important at the Internet scale.

- In designing computer systems, our geographic scale is the microprocessor: faster memory is placed closer to processor, minimizing the length of wires data has to travel. A GHz processor performs each operation in nanoseconds, so at the speed of light the signal can travel a distance of only *few inches* in this time.

- In the global Internet, the *distances are in thousands of miles.* If we create a cache of webpage content that is geographically much closer to the people who want it, those pages can be served a lot faster.

- In the Internet, therefore, physical *proximity*, rather than memory performance, is the scarce resource. But the principle of caching applies just the same!

- In fact, much of the Internet traffic is now handled by CDNs (content distribution networks), which have computers around the world maintaining copies of popular web pages: Internet caches. Remarkably, a quarter of all internet traffic is handled by a single company, called Akamai, founded by a CS professor Tom Leighton.

- Similarly, Amazon's enormous fulfillment centers organize items for fast retrieval: items are not arranged in any human-comprehensible organization, the kind you will find in a library or department store, but rather to make their retrieval by algorithms (robot) efficient.

2 Analyzing Caching Algorithms

- With that general context for caching, let us now study performance of caching algorithms in a more rigorous way. In order to analyze cache algorithms, we will work with the following simple abstract model.

  1. Data is organized in fixed size chunks, called *pages* or *blocks*.
  2. *2 Level Memory Hierarchy.* We have a *cache* (fast memory) capable of holding k pages, and an unlimited amount of slower (main) memory.
  3. *Request Sequence.* The process makes a sequence of page requests; the order of requests is *not known* to the algorithm.
  4. *Cost Model.* Suppose the next request is for page p.
     - If p is already in cache, we call it a *cache hit* and the algorithm incurs 0 cost.
     - If p is not in cache, we call it a *cache miss*, and the algorithm must fetch p from main memory into cache, *at the cost of* 1.
  5. *Cache Policy.* Bringing p into cache may require evicting some page from the cache. The goal of the caching algorithm is to minimize the total cost (number of cache misses), relative to the best caching algorithm, over all request sequences.

- We will compare the cache performance of our *online* algorithm with Belady's *offline* algorithm—that is, given a request sequence S, suppose our online algorithm incurs C_A misses, and the optimal algorithm, which can see the entire sequence S in advance to plan its page evictions, incurs C_{opt} misses, then we want to minimize the ratio

$$\frac{C_A}{C_{\text{opt}}}$$

  which is called the *competitive ratio* of A.

- The algorithm A must attain this competitive ratio over all request sequences S—namely, worst-case analysis.

- Let us begin with some general remarks about caching algorithms and by proving that Belady's algorithm—always evict the page whose next request is furthest in the future—is optimal.

2.1 Optimality of Belady's Algorithm

1. Analysis of Belady's uses a greedy swap argument, but is subtle. We first make an intuitively obvious but important claim: we can assume that a caching algorithm only evicts a page when a new request is made and the cache is full.

2. That is, we can focus on *demand paging* algorithms, which never evict paper preemptively. It is easy to see any preemptive eviction policy can be converted to a demand eviction policy without degrading the cache performance.

3. **Theorem.** LFD (longest future distance) is an optimal offline caching algorithm.

4. **Proof.**

   - We show that any algorithm can be modified to act like LFD without degrading its performance.
   - In particular, let A be a paging algorithm, and let S be a request sequence. Then, for any $i = 1, 2, \ldots, |S|$, we can construct an offline algorithm A_i that satisfies the following three properties:
     - A_i processes the first $i-1$ request exactly as A
     - if the ith request is a page fault, A_i evicts the page with longest forward distance, and
     - $C_{A_i}(S) \leq C_A(S)$.
   - Thus, A_i mimics Belady's strategy for the first i requests. This claim will prove the theorem by applying it $n = |S|$ times, transforming any paging algorithm into Belady's algorithm, with the same cost.
   - By induction, we assume that after processing the first $i-1$ pages, the caches of A_i and A are identical.
   - Suppose that that after processing the ith request, we have

     $$A\text{'s cache} \;=\; X + v \qquad A_i\text{'s cache} \;=\; X + u$$

     In other words, the ith request is a cache miss, and A and A_i evict different pages to fetch the newly requested page p. (Except for the page they evicted, the remaining $k-1$ pages are same for them.)
   - Clearly, if $v = u$, there is nothing to prove, so assume that $v \neq u$.
   - A_i now processes the remaining request sequence as follows:

- Suppose the next request is *not* for v. Then, A_i mimics A. This is feasible since after servicing any requests in this way, the number of common pages remains at least $k-1$. The only *exception* is that if A ever evicts v, then A_i evicts u, and in doing so their caches become identical.
- If the next request is for page v, and the two caches are not identical, then A_i suffers a page fault, while A does not. However, by the time v is requested, since it had the longest forward distance when evicted by A_i, there must have been at least one request for u, after the ith page fault. The first such request causes a fault for A but not A_i, and so the total number of faults for A_i and A after serving v are equal. In order to serve the request for v, A_i evicts u and the two caches become identical.

2.2 Lower Bounds

- We begin by showing that the performance of some of the natural online algorithms can be *unboundedly* bad.
- **Claim.** LIFO and LFU have *unbounded* competitive ratios. That is, $\rho_A \to \infty$, for $A =$ LIFO or LFU.
  1. Let us first consider LIFO.
     - Suppose we have a cache of size k, initially containing pages $1, 2, \ldots, k$.
     - Construct the remainder sequence as: $k+1, k, k+1, k, \ldots, k+1, k$.
     - The last request was for page k, so under LIFO policy the new request $k+1$ evicts k, which gets evicted when k is requested again, and so on.
     - Thus, LIFO incurs a page fault on every single request in this *arbitrarily long* sequence, whereas OPT can process this sequence with just one miss.
  2. Now consider LFU.
     - We start with the cache $\{1, \ldots, k\}$, and first make m requests for each of the pages $1, 2, \ldots, k-1$.
     - Next we make m requests for pages $k+1$ and k, alternating between them.
     - This gives us a total of $2m$ cache misses because each time k is requested, $k+1$ is the least frequently used item, and vice versa.
     - On the same sequence, the OPT will make only one miss. Since m can be made arbitrarily large, LFU has unbounded competitive ratio.
- A very broad worst-case lower bound for any *deterministic* caching policy is the following.

- **Claim.** *No deterministic algorithm can be $< k$-Competitive.*

  1. Consider a malicious request sequence, constructed out of a set of $k+1$ pages.
  2. Given algorithm A, the adversary always requests the one page not in A's cache. Thus, A will fault on each request.
  3. By contrast, once the adversary has fixed its request sequence and the LFD algorithm can inspect it, the offline will always evict the page whose next request is farthest in the future, so it will have one page fault every k requests.
  4. Thus, the competitive ratio of any *online* algorithm relative to the best *offline* algorithm is $\geq k$.

- On the positive side, we now show that LRU can in fact match this lower bound.

2.3 LRU is k-Competitive

- We show that LRU is k-competitive, meaning that the ratio between its number of faults and optimal is at most k.
- Let us implicitly, for the purpose of analysis, divide the request sequence into *blocks.* Each block is defined as the *maximal* sequence containing k distinct page requests.
- Block i is the empty sequence. For each $i \geq 1$, block i is the maximal sequence following block $i-1$ that has at most k distinct page requests.
- The block partition is well defined and independent of how any particular algorithm processes the input.
- As an example, suppose the request sequence is $4, 2, 4, 1, 2, 4, 3, 2, 1, 4, 2, 4, 2, 3, 1, 5, 2, 1$. Its block partition, assuming $k = 3$, is the following:

$$\{4, 2, 4, 1, 2, 4\}, \quad \{3, 2, 1\}, \quad \{4, 2, 4, 2, 3\}, \quad \{1, 5, 2\}$$

  Phase 1 begins with 4 and ends just before 3, which is the 4th distinct page in the block occurs.

- For the analysis, we observe that LRU faults at most k times in any block—this follows because there are k distinct page requests in a block, and once requested a page will not be evicted from the cache under the LRU policy until this block ends. Thus, LRU will not fault twice on the same page within a block.

- To complete the analysis, we now show that any algorithm including OPT must fault at least once during each (slightly modified) block. We argue as follows.

    1. Consider the first block plus the first request of the second block. Altogether they have $k+1$ distinct page requests, and therefore any algorithm with cache size k must fault at least once.
    2. Now suppose p is the first request of the 2nd block. That leaves $k-1$ pages in the cache. Together with the first request of the *next* block, the algorithm has k distinct pages to serve (not counting p), and thus again at least one fault must occur.
    3. Therefore, the OPT must have at least one page fault for each block, with the exception of the last block.

- This proves that the competitive ratio of LRU is at most k.

Remarks

- Is a competitive ratio of k good? Unlike LIFO or LFU, whose competitive ratio can grow without bound, LRU's ratio is bounded by the cache size k, and does not depend on the input sequence length.
- On the other hand, the size of cache in practical systems is usually not a small constant, and so this theoretical guarantee may seem laughable.
- We will come back to a more in-depth discussion about the theoretical guarantees of LRU and how to interpret it later.

Computational Overhead of Caching Algorithms

- While we are mostly focusing on the single criterion of cache misses as our performance measure, it is also important to keep in mind that *implementations* of different cache algorithms may have different overhead costs.
- For instance, LRU must maintain a counter for each of its pages, and so requires $O(k \log k)$ bits. By contrast, FIFO can be maintained by a single mod k counter, and requires $O(\log k)$ bits.

2.4 Randomized 1-Bit LRU Algorithm

1. The previous lower bound rules out the possibility of a deterministic algorithm with better than k competitive ratio. We now show that randomization leads to a significantly better bound.

2. In particular, the algorithm works as follows.

   - Initially, all pages are *unmarked*. For each request, if the requested page is in the cache, the algorithm marks it.
   - On a miss, if there is at least one unmarked page in the cache, uniformly at random evict an unmarked page, fetch the requested page and mark it. Otherwise, unmark all the pages and start a new block.

3. Instead of keeping track of the exact *last use* time of a page, the algorithm uses a 1-bit marker to track whether a page has been used during the current block. As a result, it is often called a 1-Bit LRU algorithm. (It is also called Marking algorithm.)

4. **Randomized 1-Bit LRU Algorithm.**

   - If the requested page is not in the cache, then
   - if all pages are marked, then unmark all the pages
   - uniformly at random choose an unmarked page p
   - evict p, fetch and mark requested page

5. *The Marking algorithm implicitly keeps track of the blocks by unmarking all the pages at the start of a block.* That is, each unmarking of all pages begins a new block, which ends when all pages become marked.

6. The competitive ratio claim below is for the expected number of misses, due to randomization in the algorithm.

7. **Theorem.** The 1-Bit LRU algorithm is $O(\log k)$-competitive.

8. **Proof.**

   - We will analyze the expected number of misses in block i.
   - Clearly, within each block k distinct pages are requested, and all the pages requested during a block are marked, and they stay in the cache until the block ends. That is, at the end of block $i-1$, there are k pages in the cache.

- Let m_i denote the number of "new" pages requested in block i, pages that were not requested during block $i-1$. We will call the remaining requested pages of block i "old," because they were also requested in block $i-1$.
- Requesting a new page necessarily causes an eviction, while requesting an old page may not.
- Without loss of generality, assume that the m_i new page requests arrive *first* during the block i; you can convince yourself that this is indeed the worst-case by thinking about how caches misses are affected by the delay in page requests.
- Once the m_i new pages are requested, each causing a cache miss, they are marked in the cache. The remaining $k-m_i$ pages in the cache are old. We need to analyze how many of them lead to a cache miss.
- Consider the probability of a cache miss when we request an old page for the first time, e.g. when we request the (m_i+1)th distinct page in block i. This old page by definition was in the cache at the end of block $i-1$, but it may have been evicted when one of the m_i new pages was requested.
- Let us calculate the probability that a cache miss occurs on an "old" page We claim that the jth old page *is in the cache*, when it is requested for the first time during block i, with probability

  $$\frac{k-m_i-(j-1)}{k-(j-1)}$$

  To see, note that the numerator is the number of old unmarked pages in the cache, whole denominator is the total number of old unmarked pages.
- The probability of a page fault for this request is, therefore,

  $$1-\frac{k-m_i-(j-1)}{k-(j-1)} \quad = \quad \frac{m_i}{k-j+1}$$

- Thus, the total number of expected cache misses in block i is at most

  $$m_i \; + \; \sum_{j=1}^{k-m_i} \frac{m_i}{k-j+1} \quad \leq \quad m_i H_k \quad = \quad O(m_i \log k)$$

- If the total number of blocks is N, then expected cache miss count for 1-Bit LRU is at most

  $$\text{1-BIT-LRU} \quad \leq \quad H_k \sum_{i=1}^{N} m_i$$

- What remains now is to analyze the number of cache misses for the optimal.
- *While it is still true that OPT must fault at least once per block, as in the analysis of deterministic LRU, that lower bound is too weak. We need to somehow relate the number of faults to* $\sum_i m_i$. So, we need a bit more refined analysis.
- Let n_i be the number of cache misses by OPT in block i. Since in total there are $k + m_i$ distinct pages requested during block $i-1$ and i, at least m_i of them must cause a miss. Thus, we claim that

$$n_{i-1} + n_i \quad \geq \quad m_i$$

and therefore

$$OPT \quad \geq \quad \frac{1}{2} \sum_{i=1}^{N} m_i$$

- Combining with the earlier upper bound on sum of m_is we get that the total number of misses for LRU is at most $2H_k$ times the optimal.

3 List Searching Rules

- The caching problem also arises in seemingly an even simpler setting: *maintaining a linear list.*
- Suppose we want to organize a set of items in a linear list, allowing items to be inserted and deleted, where to search for an item we begin at the head of the list incurring cost proportional to the number of items searched before termination. In particular, we have have the following goal.
- Maintain a set S of items with the following operations:
  1. access(i): locate item i in S
  2. insert(i): insert item i
  3. delete(i): delete item i
- Self-Organizing List: The list S is not sorted, so we can maintain it any order we want to help us minimize the search cost, *but of course we pay for any rearrangements* we perform.
- This basic problem is so ubiquitous in computing applications that many different list-rearrangement rules, as in caching, have been studied and implemented. Some common ones being:

1. Move-to-Front: After access of insert, move item to the head of list.
2. Transpose: After access or insert, move up one position.
3. Frequency: Keep items in frequency of access order.

- A natural question is how well do these heuristics perform?
- In a famous paper "Amortized Efficiency of List Update and Paging Rules" [Comm. of ACM, Feb. 1985], Dan Sleator and Robert Tarjan analyzed them using competitive analysis.
- Consider a sequence of m access, insert, deletes.
- Let n be the maximum number of items in list at any time.
- Let C_{MF}, C_T, C_{FC} be worst-case costs under 3 update rules, and let C_{opt} be the cost of optimal algorithm.
- Sleator-Tarjan prove the following results:

1. For any sequence of updates,
$$\frac{C_{MF}}{C_{opt}} \leq 2$$
2. There is a sequence of updates where
$$\frac{C_T}{C_{opt}} = \Omega(n)$$
3. There is a sequence of updates where
$$\frac{C_{FC}}{C_{opt}} = \Omega(n)$$

4 Human Memory and Forgetting Curve

- Ideas from computer science about organization of memory have even influenced how psychologists think about *human brain and memory*.
- The science of human memory is said to have begin in 1879 with a young psychologist Hermann Ebbinghaus at University of Berlin. He wanted to know how memory worked and to study mind with the mathematical rigor of physical sciences.

- For one year, Ebbinghaus would sit down and memorize a list of nonsense syllables each day. Then he would test himself on lists from previous days.

    1. He established many of the most basic results in human memory research. For instance, he confirmed that practicing a list multiple times makes it persist longer in memory, and the number of items one can accurately recall goes down as time passes.
    2. Produced a *forgetting curve*, a graph of how memory fades over time.
    3. His results established the credibility of a quantitative science of human memory, but left open the mystery: *why this particular curve?* These questions have stimulated psychologists for more than a hundred years.

- CMU psychologist John Anderson writes: for a long time, it was felt that there was something missing in the existing theories of human memory. Basically, all these theories characterize memory as an arbitrary and non-optimal configuration. I had felt that basic memory processes were quite adaptive and perhaps even optimal but I had never been able to see a framework in which to make this point. The information retrieval framework in CS offered a solution.

5 Cache Model of Human Memory

- A natural way to think about forgetting is that our minds simply run out of space. The important idea behind Anderson's new account of human memory is that the problem may not be one of *storage* but one of *organization*.

- According to this theory, the mind has essentially infinite capacity for memories, but we have only a *finite amount of time in which to search for them.*

- In analogy to Noguchi Filing System of LoC, you can place as many books as you want on a shelf, but the closer something is to the front, the faster it can be found.

- The key to good human memory then becomes the same as the key to good computer cache: *predicting which items will be most likely needed in the future.*

- Barring clairvoyance, the best approach to making such predictions requires understanding the world itself. Anderson and Schooler set out to perform Ebbinghaus-like studies not on human minds, *but on human societies.*

- The question was straightforward: what patterns characterize the way the world itself "forgets"–that is in what way do events and references fade over time?

- They analyzed three environments: headlines from NY Times, recordings of parents talking to their children, and Anderson's own email box.
- They found that a word is most likely to appear right after it had just been used, and that the likelihood of seeing it again falls off as time goes on. In other words, the reality itself has a statistical structure that mimics Ebbinghaus curve.
- This suggests the following remarkable conclusion: if the pattern by which things fade from our minds is the same pattern by which things fade from use around us, then Ebbinghaus curve may simply show a perfect tuning of human brain to the world—making available precisely the things most likely to be needed!
- In putting the emphasis on time, caching shows us the memory involves unavoidable tradeoffs, and a certain zero-sumness. You can't have every library book on your desk, or every product on display. In the same way, you can't have every fact or name at the front of your mind.
- Anderson: many people believe human memory is sub-optimal, and point to its many frustrating failings as evidence. However, these criticisms fail to appreciate the task before human memory, which is to try to manage a huge stockpile of memories.
- In any system responsible for managing a vast database there must be failures of retrieval. It is just too expensive to maintain access to an unbounded number of items.
- This understanding has in turn led to a second revelation about human memory. If these tradeoffs are unavoidable, and the brain appears to be optimally tuned to the world around it, then what we refer to as the inevitable "cognitive decline" that comes with age may in fact be something else.

5.1 Cognitive Decay

- Caching is a solution to a seemingly fundamental performance bottleneck: *size alone is enough to impair speed.*

    1. If we make a city bigger, it takes longer to travel between two points.
    2. If you make a library bigger, it takes longer to find a book.
    3. If you have a larger stack of papers on the desk, it takes longer to find one you need.

- In the same vein, the larger memory unavoidably is slower. In fact, the fastest current cache memory SRAM *costs about thousand times more per byte* than the flash memory in solid state drives.

- Can our experiences with computer memory explain why we humans suffer from cognitive decline in our older years? We often find ourselves in situations where the word we are looking for is just "on the tip of the tongue." We forget our phone numbers, passwords, various dates, or episodes in life.
- Can this be a result of *simply too much data to store and recall?*
    1. A typical two year old knows a few hundred words. A typical adult knows about $30,000$!
    2. In terms of episodic memory, every year adds about a $1/3$ of *a million waking minutes* to lived experience.
- Considered this way, the memory's slowdown does not seem surprising, rather the fact the mind can possibly stay afloat and responsive as so much data accumulates.
- If the fundamental challenge of memory is one of organization rather than storage, perhaps it should change how we think about the impact of aging on our mental abilities.
- Cognitive decline—lags and retrieval errors—may not be about the *search process* slowing or deteriorating, but (at least partly) an unavoidable consequence of the fact that the amount of information we have to navigate keeps getting bigger and bigger.
- Linguist Ramsacar's research showed that simply knowing more things makes it harder when it comes to recognizing words, or names.
- Caching gives us the language to understand what's happening. We say "brain fart" when we should really say "cache miss". The occasional lags in retrieval should remind us how great the cache performance has been in all other cases!