CS168: The Modern Algorithmic Toolbox
Lecture #1: Introduction and Consistent Hashing

Tim Roughgarden & Gregory Valiant*

April 2, 2024

1 Consistent Hashing

1.1 Meta-Discussion

We'll talk about the course in general in Section 2, but first let's discuss a representative technical topic: *consistent hashing.* This topic is representative in the following respects:

1. As you could guess by the word "hashing," the topic builds on central algorithmic ideas that you've already learned (e.g., in CS161) and adapts them to some very real-world applications.

2. The topic is "modern," in the sense that it is motivated by issues in present-day systems that were not present in the applications of yore — consistent hashing is not in your parents' algorithms textbook, because back then it wasn't needed. The original idea isn't that new anymore (from 1997), but it has been repurposed for new technologies several times since then.

3. Consistent hashing is a "tool" in the sense that it is a non-obvious idea but, once you know it, it's general and flexible enough to potentially prove useful for other problems. In this course, we'll be looking for the following trifecta: (i) ideas that are *non-obvious*, even to the well-trained computer scientist, so that we're not wasting your time; (ii) *conceptually simple* — realistically, these are the only ideas that you might remember a year or more from now, when you're a start-up founder, senior software engineer, PhD student, etc. (iii) *fundamental*, meaning that there is some chance that the idea will prove useful to you in the future.

4. The idea has real applications. Consistent hashing gave birth to Akamai, which to this day is a major player in the Internet, managing the Web presence of tons of major

---

companies. (Quantitatively, Akamai serves 10-30% of all internet traffic, and has a market cap >15B.) More recently, consistent hashing has been repurposed to solve basic problems in peer-to-peer networks (initially in [4]), including parts of BitTorrent. These days, all the cool kids are using consistent hashing for distributed storage — made popular by Amazon's Dynamo [1], the idea is to have a lightweight alternative to a database where all the data resides in main memory across multiple machines, rather than on disk.

1.2   Web Caching

The original motivation for consistent hashing (in 1997) was Web caching. You're familiar with the concept and benefits of caching. In the context of the Web, imagine a browser requesting a URL, like `amazon.com`. Of course, one could request the page from the appropriate Web server. But if the page is being requested over and over again, it's wasteful to repeatedly download it from the server. An obvious idea is to use a Web cache, which stores a local copy of recently visited pages. When a URL is requested, one can first check the local cache for the page. If the page is in the cache, one can send it directly to the browser — no need to contact the original server. If the page is not in the cache, then the page is downloaded from a suitable server as before, and the result is both sent to the browser and also stored in the local cache for future re-use.

Caching is good. The most obvious benefit is that the end user experiences a much faster response time. But caches also improve the Internet as a whole: fewer requests to far away servers means less network traffic, less congestion in the queues at network switches and Web servers, fewer dropped packets, etc.

So clearly we want Web caches. Where should they go? A first idea is to give each end user their own cache, maintained on their own machine or device. So if you request `amazon.com`, and you also requested it in the recent past, then the page can be served from your local cache. If not, you incur a cache miss, and the page is downloaded and stored in your local cache.

However, we could take the benefit of caching to the next level if we could implement a Web cache that is *shared by many users*, for example, all users of Stanford's network. For example, if you haven't accessed `amazon.com` recently but someone "nearby" has (e.g., someone else in the Stanford network), wouldn't it be cool if you could just use their local copy? The benefits should be clear: by aggregating the recent page requests of a large number of users, these users will enjoy many more cache hits and consequently less latency. Akamai's goal was to take the daydream of a single logical Web cache shared by tons of users and turn it into a viable technology.

Why isn't this an easy problem? We focus on one of several obstacles. Namely, remembering the recently accessed Web pages of a large number of users might take a lot of storage. Unless we want to resort to a big, slow, and special-purpose machine for this purpose, this means that the aggregated cache might not fit on a single machine. Thus, implementing a shared cache at a large scale requires *spreading the cache over multiple machines*.

Now suppose a browser requests `amazon.com` and you want to know if the Web page has
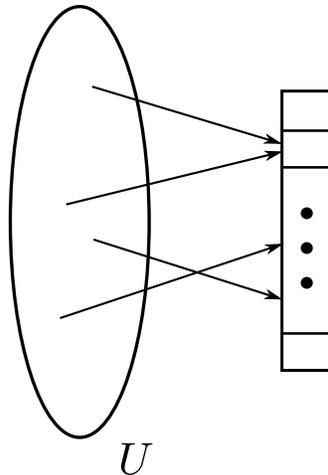
2

Figure 1: A hash function maps elements from a (generally large) universe U to a list of "buckets," such as 32-bit values.

been cached locally. Suppose the shared cache is spread over 100 machines. Where should you look for a cached copy of the Web page? You could poll all 100 caches for a copy, but that feels pretty dumb. And with lots of users and caches, this solution crosses the line from dumb to infeasible. Wouldn't it be nice if, instead, given a URL (like `amazon.com`) we magically knew which cache (like #23) we should look to for a copy?

1.3   A Simple Solution Using Hashing

Formally, we want a mapping from URLs to caches. The first thought of a well-trained computer scientist might be to use a hash function for this purpose.[1] Recall that a *hash function* maps elements of a (usually super-big) universe U, like URLs, to "buckets," such as 32-bit values (Figure 1). A "good" hash function h satisfies two properties:

1. It is easy to remember and evaluate. Ideally, computing the function involves just a few arithmetic operations, and maybe a "mod" operation.

2. For all practical purposes, h behaves like a totally random function, spreading data out evenly and without noticeable correlation across the possible buckets.

Designing good hash functions is not easy — hopefully you won't need to do it yourself — but you can regard it as a solved problem. A common approach in practice is to use a well-known and well-crafted hash function like MD5[2] — it's overwhelmingly likely that this

---

[1]We'll assume that you've seen hashing before, probably multiple times. See the course site for review videos on the topic.

[2]This is built in to most programming languages, or you can just copy the code for it from the Web. Or you might want to use something faster and more lightweight (but still well tested), like from the FarmHash family.

function will "behave randomly" for whatever data set you're working with. Theoretical guarantees are possible only for *families* of hash functions,[3] which motivates picking a hash function at random from a "universal" family (see CS161 for details).

Taking the existence of a good hash function h for granted, we can solve the problem of mapping URLs to caches. Say there are n caches, named $\{0, 1, 2, \ldots, n-1\}$. Then we can just store the Web page with URL x at the cache server named

$$h(x) \bmod n. \tag{1}$$

Note that $h(x)$ is probably something like a 32-bit value, representing an integer that is way way bigger than n — this is the reason we apply the "$\bmod n$" operation to recover the name of one of the caches.

The solution (1) of mapping URLs to caches is an excellent first cut, and it works great in many cases. To motivate why we might need a different solution, suppose the number n of caches is not static, but rather is changing over time. For example, in Akamai's early days, they were focused on adding as many caches as possible all over the Internet, so n was constantly increasing. Web caches can also fail or lose connection to the network, which causes n to decrease. In a peer-to-peer context (see Section 1.5), n corresponds to the number of nodes of the network, which is constantly changing as nodes join and depart.

Suppose we add a new cache and thereby bump up n from 100 to 101. For an object x, it is very unlikely that $h(x) \bmod 100$ and $h(x) \bmod 101$ are the same number, Thus, *changing n forces almost all objects to relocate.* This is a disaster for applications where n is constantly changing.[4]

1.4   Consistent Hashing

Our criticism of the solution (1) for mapping URLs to caches motivates the goal of *consistent hashing*: we want hash table-type functionality (we can store stuff and retrieve it later) with the additional property that almost all objects stay assigned to the same cache even as the number n of caches changes. We next give the most popular implementation of this functionality [2].[5]

---

[3]Assuming that the number of buckets n is significantly smaller than the universe size U, every fixed hash function has a large pathological data set $S \subseteq U$ for it, with all elements of S colliding and hashing to the same bucket. (Hint: Pigeonhole Principle.)

[4]These relocations should remind you of the "rehash" operation in most hash table implementations. Recall that as the load factor (# of elements/# buckets) of a hash table increases, the search time degrades. So when the load factor gets too big one usually increases the number of buckets (by 2x, say), which requires rehashing all of the elements. This is an expensive operation, traditionally justified by arguing it will be invoked infrequently. In the Web caching context, the number of buckets is changing all the time, rather than infrequently. Also, rehashing an object now involves moving data between machines across a network, which is much more expensive than rehashing elements of a hash table stored on a single machine.

[5]You should never conflate two fundamentally different things: (i) what an algorithm or data structure is responsible for doing (i.e., its specification/API); and (ii) the implementation (i.e., how the desired functionality is actually achieved). There can of course be multiple implementations with exactly the same functionality. For example, the functionality of consistent hashing can also be achieved by other implementations [5, 3].

4

Figure 2: Each element of the array above is a bucket of the hash table. Each object x is assigned to the first cache server s on its right.
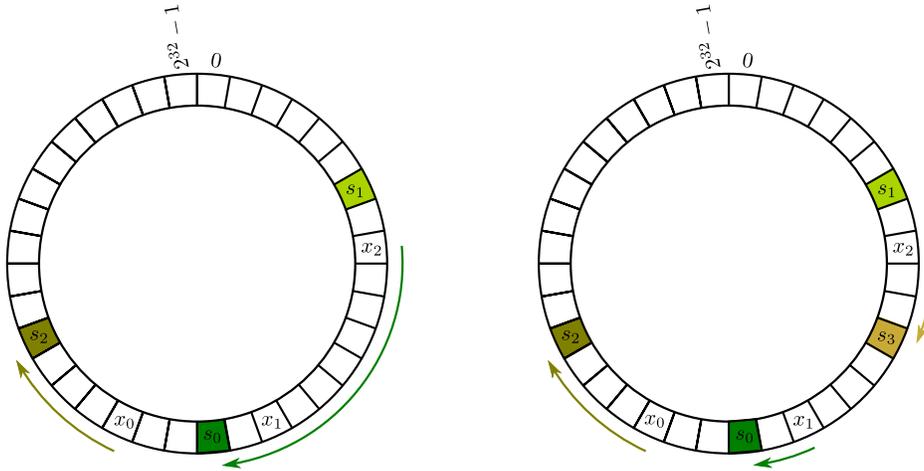

Figure 3: (Left) We glue 0 and $2^{32} - 1$ together, so that objects are instead assigned to the cache server that is closest in the clockwise direction. This solves the problem of the last object being to the right of the last cache. (Right) Adding a new cache server s_3. Object x_2 moves from s_0 to s_3.

The key idea is: in addition to hashing the names of all objects (URLs) x, like before, *we also hash the names of all the cache servers s*. The object and cache names need to be hashed to the same range, such as 32-bit values.

To understand which objects are assigned to which caches, consider the array shown in Figure 2, indexed by the possible hash values. (This array might be very big, e.g. 2^{32}, and it exists only in our minds; we'll discuss the actual implementation shortly.) Imagine that we've already hashed all the cache server names and made a note of them in the corresponding buckets. Given an object x that hashes to the bucket $h(x)$, we scan buckets to the right of $h(x)$ until we find a bucket $h(s)$ to which the name of some cache s hashes. (We wrap around the array, if need be.) We then designate s as the cache responsible for the object x.

This approach to consistent hashing can also be visualized on a circle, with points on the circle corresponding to the possible hash values (Figure 3(left)). Caches and objects both hash to points on this circle; an object is stored on the cache server that is closest in the clockwise direction. Thus n caches partition the circle into n segments, with each cache responsible for all objects in one of these segments.

This simple idea leads to some nice properties. First, assuming reasonable hash functions,

by symmetry, the expected load on each of the n cache servers is exactly a $\frac{1}{n}$ fraction of the objects. (There is non-trivial variance; below we explain how to reduce it via replication.) Second, and more importantly, suppose we add a new cache server s — which objects have to move? *Only the objects stored at s.* See Figure 3(right). Combined, these two observations imply that, in expectation, adding the nth cache causes only a $\frac{1}{n}$ fraction of the objects to relocate. This is the best-case scenario if we want the load to be distributed evenly — clearly the objects on the new cache have to move from where they were before. By contrast, with the naive solution (1), on average only a $\frac{1}{n}$ fraction of the objects *don't* move when the nth cache is added![6]

So how do we actually implement the standard hash table operations Lookup and Insert? Given an object x, both operations boil down to the problem of efficiently implementing the rightward/clockwise scan for the cache server s that minimizes $h(s)$ subject to $h(s) \geq h(x)$.[7] Thus, we want a data structure for storing the cache names, with the corresponding hash values as keys, that supports a fast *Successor* operation. A hash table isn't good enough (it doesn't maintain any order information at all); a heap isn't good enough (it only maintains a partial order so that identifying the minimum is fast); but recall that *binary search trees*, which maintain a total ordering of the stored elements, do export a Successor function.[8] Since the running time of this operation is linear in the depth of the tree, it's a good idea to use a balanced binary search tree, such as a Red-Black tree. Finding the cache responsible for storing a given object x then takes $O(\log n)$ time, where n is the number of caches.[9]

Reducing the variance: While the expected load of each cache server is a $\frac{1}{n}$ fraction of the objects, the realized load of each cache will vary. Pictorially, if you pick n random points on the circle, you're very unlikely to get a perfect partition of the circle into equal-sized segments.

An easy way to decrease this variance is to make k "virtual copies" of each cache s, implemented by hashing its name with k different hash functions to get $h_1(s), \ldots, h_k(s)$. (More on using multiple hash functions next lecture.) For example, with cache servers $\{0, 1, 2\}$ and $k = 4$, we choose 12 points on the circle — 4 labeled "0", 4 labeled "1", and 4 labeled "2". (See Figure 4.) Objects are assigned as before — from $h(x)$, we scan

---

[6]You might wonder how the objects actually get moved. There are several ways to do this, and the best one depends on the context. For example, the new cache server could identify its "successor" and send a request for the objects that hash to the relevant range. In the original Web caching context, one can get away with doing nothing: a request for a Web page that was re-assigned from an original cache s to the new cache s' will initially result in a cache miss, causing s' to download the page from the appropriate Web server and cache it locally to service future requests. The copies of these Web pages that are at s will never be used again (requests for these pages now go to s' instead), so they will eventually time out and be deleted from s.

[7]We ignore the "wraparound" case, which can be handled separately as an edge case.

[8]This operation is usually given short shrift in lectures on search trees, but it's exactly what we want here!

[9]Our description, and the course in general, emphasizes fundamental concepts rather than the details of an implementation. Our assumption is that you're perfectly capable of translating high-level ideas into working code. A quick Web search for "consistent hashing python" or "consistent hashing java" yields some example implementations.
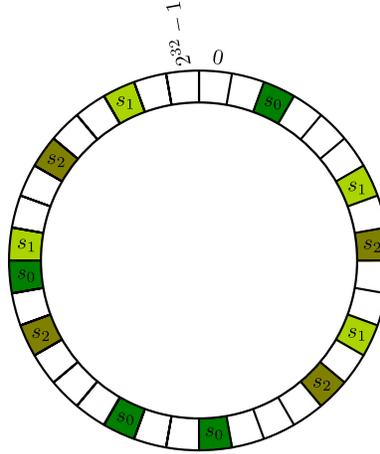
Figure 4: Decreasing the variance by assigning each cache server multiple hash values.

rightward/clockwise until we encounter one of the hash values of some cache s, and s is responsible for storing x. By symmetry, each cache still expects to get a $\frac{1}{n}$ fraction of the objects. This replication increases the number of keys stored in the balanced binary search by a factor of k, but it reduces the variance in load across cache servers significantly. Intuitively, some copies of a cache will get more objects than expected (more than a $\frac{1}{kn}$ fraction), but this will be largely canceled out by other copies that get fewer objects than expected. Choosing $k \approx \log_2 n$ is large enough to obtain reasonably balanced loads. We'll teach you some methods for reasoning mathematically about such replication vs. variance trade-offs later in the course.

Virtual copies are also useful for dealing with heterogeneous caches that have different capacities. The sensible approach is to make the number of virtual copies of a cache server proportional to the server's capacity; for example, if one cache is twice as big as another, it should have twice as many virtual copies.

1.5   Some History of Consistent Hashing

1. 1997: The implementation of consistent hashing given in this lecture first appeared in a research paper in STOC ("Symposium on the Theory of Computing") [2] — this is one of the main conferences in theoretical computer science.[10]   Ironically, the paper had previously been rejected from a theoretical computer science conference because at least one reviewer felt that "it had no hope of being practical."

2. 1998: Akamai is founded.

3. March 31, 1999: A trailer for "Star Wars: The Phantom Menace" is released online,

---

[10]The concept of consistent hashing was also invented, more or less simultaneously, in [5]. The implementation in [5] is different from and incomparable to the one in [2].

with Apple the exclusive official distributor. `apple.com` goes down almost immediately due to the overwhelming number of download requests. For a good part of the day, the only place to watch (an unauthorized copy?) of the trailer is via Akamai's Web caches. This put Akamai on the map.

4. April 1, 1999: Steve Jobs, having noticed Akamai's performance the day before, calls Akamai's President Paul Sagan to talk. Sagan hangs up on Jobs, thinking it's an April Fool's prank by one of the co-founders, Danny Lewin or Tom Leighton.

5. September 11, 2001: Tragically, co-founder Danny Lewin is killed aboard the first airplane that crashes into the World Trade Center. (Akamai remains highly relevant to this day, however.)

6. 2001: Consistent hashing is re-purposed in [4] to address technical challenges that arise in peer-to-peer (P2P) networks. A key issue in P2P networks is how to keep track of where to look for a file, such as an mp3. This functionality is often called a "distributed hash table (DHT)." DHTs were a very hot topic of research in the early years of the 21st century.

   First-generation P2P networks (like Napster) solved this problem by having a centralized server keep track of where everything is. Such a network has a single point of failure, and thus is also easy to shut down. Second-generation P2P networks (like Gnutella) used broadcasting protocols so that everyone could keep track of where everything is. This is an expensive solution that does not scale well with the number of nodes. Third-generation P2P networks, like Chord [4], use consistent hashing to keep track of what's where. The key challenge is to implement the successor operation discussed in Section 1.4 even though nobody is keeping track of the full set of cache servers. The high-level idea in [4], which has been copied or refined in several subsequent P2P networks, is that each machine should be responsible for keeping track of a small number of other machines in the network. An object search is then sent to the appropriate machine using a clever routing protocol.

   Consistent hashing remains in use in modern P2P networks, including for some features of the BitTorrent protocol.

7. 2006: Amazon implements its internal Dynamo system using consistent hashing [1]. The goal of this system is to store tons of stuff using commodity hardware while maintaining a very fast response time. As much data as possible is stored in main memory, and consistent hashing is used to keep track of what's where.

   This idea is now widely copied in modern lightweight alternatives to traditional databases (the latter of which tend to reside on disk). Such alternatives generally support few operations (e.g., no secondary keys) and relax traditional consistency requirements in exchange for speed. As you can imagine, this is a big win for lots of modern Internet companies.

2   About CS168

For the nuts and bolts of coursework, grading, etc., see the course Web site. Read on for an overview of the course and our overarching goals and philosophy.

2.1   Intended Audience

We welcome all comers — there's a zillion courses you could be taking, and we'll be happy and flattered if you decide to take this one.

That said, to prepare a coherent lecture, it's helpful to have a target audience in mind. We view the canonical student in the class as a senior-year computer science major. As you can see in this lecture, we assume a certain degree of "computer science maturity," taking for granted that you know and care about concepts like caching, hashing, balanced search trees, and so on. We also assume sufficient programming maturity to translate the high-level descriptions given in lecture to working implementations. If you didn't have any trouble understanding this lecture, you should be fine.

2.2   Course Topics

The plan is to cover the following topics. No worries if some of this doesn't make sense now; the list is to give you a sense of what this course is going to be about.

1. *Modern hashing.* This lecture's topic of consistent hashing is one example. Next lecture we'll discuss how hash functions can be used to perform "lossy compression" through data structures like bloom filters and count-min sketches. The goal is to compress a data set while approximately preserving properties such as set membership or frequency counts.

2. *The nearest neighbor problem and dimension reduction.* Dimension reduction continues the theme of lossy compression: it's about compressing data while approximately preserving similarity information (represented using distances). In the nearest neighbor problem, you are given a point set (e.g., representing documents) and want to preprocess it so that, given a query (e.g., representing a keyword search query), you can quickly determine which point is closest to the query. This problem offers our first method of understanding and exploring a data set.

3. *Generalization, regularization, and all that.* The next topic is motivated by issues in machine learning. You're probably at least vaguely familiar with the idea of using labeled training data to learn a prediction function. (E.g., from a collection of past emails labeled as spam or not, to learn a classifier that predicts whether or not a previously unseen email is spam.) The goal is to learn a prediction function that "generalizes," meaning one that is accurate on examples that have not been previously seen (as opposed to being accurate merely on the training samples). In particular, one

9

wants to avoid learning a prediction function that effectively memorizes the training sample — this would be "overfitting" to the particular data at hand.

Intuitively, the more training data you have, the better your chance of learning a prediction function that generalizes well. We'll introduce you to a clean and useful conceptual framework, rooted in statistical learning theory, for reasoning about these issues. We'll also cover useful techniques for avoid overfitting in the practically common case where the amount of training data is relatively small. One such technique is "regularization," whereby you penalize "more complex" prediction functions in favor of "simpler" ones. This is a version of Occam's Razor, which advocates accepting the simplest theory that explains the known observations.

4. *Linear algebra and spectral techniques.* One could also call this topic "the unreasonable effectiveness of sophomore-level linear algebra." This is a major topic, and it will occupy us for three weeks. Many data sets are usefully interpreted as points in space (and hence matrices, with the vectors forming the rows or the columns of a matrix). For example, a document can be mapped to a vector of word frequencies. Graphs (social networks, etc.) can also usefully be viewed as matrices in various ways. We'll see that linear algebraic methods are incredibly useful for exposing the "geometry" of a data set, and this allows one to see patterns in the data that would be otherwise undetectable. Exhibit A is principle component analysis (PCA), which identifies the "most meaningful" dimensions of a data set. We'll also cover the singular value decomposition (SVD), which identifies low-rank structure and is useful for denoising data and recovering missing data. Finally, we'll see some *spectral graph theory*, a suite of techniques where linear algebra is applied to reveal information about networks and graphs. Linear algebra is a much-maligned topic in computer science circles, but hopefully the geometric intuition and real-world applications we provide will bring the subject to life.

5. *Sampling and estimation.* It's often useful to view a data set as a sample from some distribution or population. How many samples are necessary and sufficient before you can make accurate inferences about the population? How can you estimate what you don't know? We'll also study the Markov Chain Monte Carlo method, by which you can estimate what you cannot compute.

6. *Alternative bases and the Fourier perspective.* This topic continues the theme of how a shift in perspective can illuminate otherwise undetectable patterns in data. For example, some data has a temporal component (like audio or time-series data). Other data has locality (nearby pixels of an image are often similar, same for measurements by sensors). A naive representation of such data might have one point per moment in time or per point in space. It can be far more informative to transform the data into a "dual" representation, which rephrases the data in terms of patterns that occur across time or across space. This is the point of the Fourier transform and other similar-in-spirit transforms.

7. *Mathematical programming.* Many optimization and data analysis problems can be solved using linear, integer, or convex programming. These days, there are powerful solvers at your disposal that can be used to attack such problems. We'll study the representative application of compressive sensing, which allows you to recover a sparse signal from amazingly few "linear measurements."

8. *Privacy Preserving Computation.* Given a dataset of sensitive information, how can one produce a set of statistics characterizing the overall dataset (e.g. the mean, variance, etc. over the entire set), or train a predictive model over the dataset, in a way that *guarantees that the published statistics or model does not reveal too much information about any specific datapoint*? We will discuss how to formalize this goal, via the notion of "differential privacy", and will see several algorithms for privacy preserving data analysis. These techniques are widely implemented by tech companies (including Apple and Google), and were also implemented in the 2020 US Census.

2.3 Course Goals and Themes

1. Our ambition is for this to be the coolest computer science course you've ever taken. Seriously!

2. We also think of CS168 as a "capstone" course, meaning a course you take at the conclusion of your major, after which the seemingly disparate skills and ideas learned in previous courses can be recognized as a coherent and powerful whole. Even before taking CS168 your computer science toolbox is rich enough to tackle timely and challenging problems, with consistent hashing being a fine example. After the course, your toolbox will be richer still. Capstone courses in computer science are traditionally software engineering courses; in comparison, CS168 will have a much stronger algorithmic and conceptual bent.

3. We focus on general-purpose ideas that are not overly wedded to a particular application domain, and are therefore potentially useful to as many of you as possible, whatever your future trajectory (software engineer, data scientist, start-up founder, PhD student, etc.).

4. If you forced us to pick the most prominent theme of the course, it would probably be *how to be smart with your data.* This has several aspects: how to be smart about storing it (like in this lecture), about what to throw out and what to retain, about how to transform it, visualize it, analyze it, etc. After completing CS168, you will be an educated client of the modern tools for performing all of these tasks.[11]

---

[11]We often receive emails from students mere weeks after the end of the course, with stories about how they already applied the tools acquired in this course at their new job or summer internship!

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[2] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663, 1997.

[3] J. Lamping and E. Veach. A fast, minimal memory, consistent hash algorithm. arXiv:1406.2294, 2014.

[4] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[5] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, 1998.