Brief Announcement: Demand-Aware Consistent Hashing with Bounded Loads & Greedy Routing

Arash Pourdamghani
TU Berlin, Germany

Chen Avin
Ben-Gurion University of the Negev, Beer Sheva, Israel

Stefan Schmid
TU Berlin, Germany
Weizenbaum Institute, Berlin, Germany
Fraunhofer SIT, Berlin, Germany

Abstract

Workloads of networked and distributed systems are often far from random, but exhibit certain structure: they are skewed and bursty. Networked systems which can adapt to and exploit such spatial and temporal structure, in a demand-aware manner, bear the potential to be more efficient than their demand-oblivious alternatives. This paper studies demand-awareness in the context of consistent hashing, a fundamental component in many modern distributed systems. We present a novel demand-aware consistent hashing method to improve access time and storage utilization in distributed hash tables. In particular, the resulting system supports local (greedy) routing, ensures bounded server loads, and applies to various network topologies. We formally prove that our algorithms achieve a constant approximation ratio in the offline scenario.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Consistent hashing, peer-to-peer networks, greedy routing

Digital Object Identifier 10.4230/LIPIcs.SAND.2026.24

Funding This project has received funding from the European Research Council (ERC), Proof-of-Concept project, FortifyNet (grant 101287293), 2026-2027. Chen Avin received funding from the Israel Science Foundation (ISF) grant no. 2497/23.

1 Introduction

In distributed systems, workload and communication traffic often feature much spatial and temporal locality, i.e., are *structured* [4, 5]. For example, many distributed systems leverage caches (or entire cache hierarchies) to benefit from locality of reference. In this paper, we focus on designing demand-aware algorithms to improve performance of consistent hashing. Consistent hashing was first introduced by [11] to facilitate efficient updates to hash tables, for example in web caching [12]. Supporting such efficient updates and dynamicity in general, is crucial for many distributed systems handling large amounts of data, for example in Amazon DynamoDB [20] and Apache Cassandra [13]. However, consistent hashing methods today are demand-oblivious, and ignore the high locality in access patterns. This can result in an inefficient use of the infrastructure and its resources [6].

In this paper, we explore how to improve the efficiency of consistent hashing, by making it demand-aware, striking an ideal tradeoff between the benefits and costs of adjustments. We show how incorporating the network and bounding server load, can result in low access cost and load variance, and hence in much faster response times. Our solution also supports efficient routing on the network: routing decisions can be local and greedy, which is particularly attractive in large-scale networks. Our approach relies on a set of offline algorithms that provide guarantees in terms of both access cost and storage utilization.

Table 1 Comparison of the consistent hashing methods most closely related to our work. We provide two variants of our system, which can be adopted based on the requirements of each use case.

Method	Storage Utilization	Average Access Cost	Local Search
Traditional [11]	Low	Low	Yes
Bounded loads [15]	Moderate	High	Yes
Revisiting [6]	Moderate	Moderate	No
Hash&Adjust [18]	High	Moderate	Yes
A-DACH [this work]	High	Low	No
L-DACH [this work]	Moderate	Low	Yes

1.1 Overview of Model and Objectives

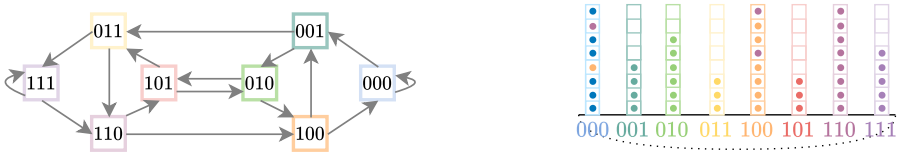
Consistent hashing is a hashing technique that allows for fast dynamic updates in the network. As the input, it considers a sequence of items (e.g., web content) that need to be inserted, accessed, or deleted from a set of servers which are interconnected by a network. Also, we should be able to add or remove servers as well.

Previous work on consistent hashing assumed that servers are placed on a ring, where each server can access only its predecessor and successor. However, in real-world scenarios, each server is usually connected to many other servers, resulting in a more communication efficient network, e.g., a network that has a much lower diameter than a ring. Hence, considering an optimized overlay network is critical in improving performance of the system. In this work, we utilize such connections, by focusing on *constant-degree networks* (e.g., de Bruijn graphs [7]).

An objective of this paper is to provide a faster access time, on average and in expectation. We show that the average cost of our algorithms is on par with an optimal, i.e., it has constant approximation factor. We also aim to improve upon previous works, by showing that our algorithms can provide guarantees on the access time, while they require less *load variance*. Ensuring less load variance results in better utilization of the storage at hand. This is particularly important given that it is unknown beforehand which server is going to have the highest load. In particular, we discuss how to just have an *additive* additional capacity on top of the minimum capacity needed.

1.2 Related Work

Traditional consistent hashing algorithms, like [11, 12]–those which do not have a bounded load per server– might suffer from a big difference between maximum and average load, when the number of items is much larger than the number of servers. To avoid such discrepancy, the work of Mirrokni et al. [15] suggested that bounding loads with a multiplicative additional capacity per server. The later variations of this work [1, 3, 6] have improved on other fronts, but kept the possibility of a multiplicative additional capacity intact. Consistent hashing with bounded loads helped teams at Google [15] and Vimeo [19] improve their content hosting system, and an open-source version of it is part of HAProxy [22]. Incorporating self-adjustments was first suggested by [18], building on the ideas from self-adjusting lists [2]. In terms of overlay networks, some of the initial structures suggest using $O(\log n)$ edges per server to ensure $O(\log n)$ diameter [9, 10, 21]. Later on, it was shown that ensuring logarithmic diameter is possible with constant degree [16]. To the best of our knowledge, previous theoretical work has mainly considered capacitated tree networks [17], rather than capacitated overlay networks. Table 1 provides an overview of key related works.



(a) The network between servers. **(b)** Servers on the virtual ring.

Figure 1 A visualization of our model, considering 8 servers with capacity $c = 8$ to store 45 items, and degree $b = 2$ De Bruijn server graph (hence, $d = 3$ and additive capacity of 2 is considered). Color of items match colors of their sources, and hence there are overflow of items to other servers (from 110 to 100, and then both to server 000).

2 Formal Model

In this section we provide formal definitions and routing mechanisms used in this paper.

Items & servers. We consider a set of n servers S, and a set of m items V. The server that an item v is currently in is its *host* server, $host(v)$. The frequency of an item v, noted by $freq(v)$, is the number of repetitions of an item in the request sequence σ with size $|\sigma|$. We assume that all servers have the same capacity for items. We allow each server to hold up to c items from set of items V. We define the number of items in a server s as its load and also the storage utilization as number of items divided by capacity, in other words $utilization = \frac{m}{c}$.

Server graph. A server graph, is an overlay network that connects servers to each other. For a constant outgoing degree b, we use a b-ary de Bruijn graph of dimension d [7]. We choose b such that b^d is bigger than number of servers . In such a construction, each server is assigned a d-digit sID (server ID) in base b. Each server is connected, through a directed connection, to at most b other servers. It is known that such a graph results also in in-going degree of b (considering self loops) [14]. Figure 1 shows an example of de Bruijn graphs with 8 servers, considering $b = 2$ and $d = 3$. For each item v, we consider an item ID, vID, which is in base b, with length $\gamma = 2 \cdot d$. The first d digits of vID determines source of an item: source of item v is the server that has sID equal to the first d digits of vID an item. The rest of digits helps us in the local routing.

Greedy routing. By considering greedy routing, we choose the next server which minimize the distance on the directed server graph to the destination. It is known that the greedy routing results in distance at most d between any two servers in a de Bruijn graph of dimension d [14]. We note that this *greedy* distance is on the directed server graph.

For a source server s, *GreedyTree* of source s, denoted by GTree(s), is a sub-tree of the server graph, rooted in s. This tree is constructed by taking a union of unique greedy paths from s to all other servers. See an example of GreedyTree in Figure 2b.

Search methods. In *global* search method, all servers maintain a *global* map. This map keeps track of sID of the host of items that are sourced in the server. When we start searching for an item from its source, we consider access to sID when following the greedy path toward the host of the item.

The *local* search builds on top of the greedy routing, and finds the item solely based on the vID of items. The local search works in two phases.

1. In the first phase, we take a walk on the server graph. The steps of this walk are only determined by the vID of the item, without the need to access a global map. For an item v, we determine the i-th step of the walk as follows: we go to the neighboring server that has $(i+d)$-th digit of vID of v as its rightmost digit.
2. After d steps of the walk, when we reach the last digit of vID, we start the second phase, and follow a Hamiltonian path from that server, to find a suitable server. We follow the Hamiltonian cycle that is created by a d digit b-ary Gray code [8]

To find an item, we start from the source of the item, and follow the above-mentioned phases, until we reach the item. We eventually find such a host server, given that we visit all server during the second phase of the local search. Procedure 1 recaps a local search step.

Procedure 1 LocalSearchStep(s, v, i).

Input: Server s, item v, counter i
Output: Next server s'
1.1 **if** $i \leq d$ **then**
1.2 $x =$ the $i+d$-th digit v's vID.
1.3 Get s' by shifting sID of s left and adding x to it.
1.4 **else**
1.5 s' is the next server following b-ary Gray code.

Distance & Access cost. Distance of an item v, $dis_A(v)$, is the number of steps that search algorithm A requires to find item v. For a given search algorithm A we define the offline access cost as $\sum_{v \in V} dis_A(v) \cdot freq(v)$.

Problem definition. Following is the formal problem tackled by this paper:

Offline Minimum Average Access Cost (Off-MAAT).

Input: A set of n servers connected as de Bruijn Graph, m items, and their frequency distribution.

Output: Find an assignment of items to the servers in the server graph such that $\sum_{v \in V} dis_A(v) \cdot freq(v)$ is minimized, while respecting greedy routing.

3 Algorithm Supporting Additive Capacity

A-DACH (as an acronym for additive demand-aware consistent hashing) is an algorithm that aims to reduce the access cost and improve the storage utilization, when global routing is allowed. In doing so, this algorithm considers *free for all* capacity decomposition. This algorithm first sorts items in descending order. Starting with the most frequent item, this algorithm puts items one by one in a server that is closest to the item's source (considering greedy routing) and is not full, and then updates the forwarding table for the source of item.

Offline *A-DACH* algorithm always terminates, as there is available capacity for all items in servers of the network (given the capacity allocation). The running time of this algorithm is $O(m \cdot \log m)$. Sorting the items takes $O(m \cdot \log m)$ time, and each item can then be placed in at most $O(\log n)$ time. The next theorem proves that this algorithm has a constant approximation (all proofs are deferred to the full version of the paper).

▶ **Theorem 1.** *Considering* ***OFF-MAAT*** *problem,* A-DACH *gives 2 approximation compared to the optimal offline algorithm, considering* $m = \omega((n \log n)^{\frac{3}{2}})$.

(a) Capacity allocation inside a source. **(b)** Capacity assigned to a serve in its tree.

Figure 2 Figure 2a shows fractional capacity allocation inside server 100 and its corresponding Gtree. The rounded cross-hatched rectangles depict the percentage of capacity that is allocated to each server. Figure 2b shows capacity reserved for the server 100 in other servers, by colored orange rectangles. These figures are based on the example server graph shown in Figure 1a.

4 Algorithm Supporting Local Search

In order to describe our algorithm we first need to define a few terms:

Fractional capacity allocation. We benefit from an elaborate collection of items in each server by assigning a fixed fraction of the capacity to a certain source. This fraction depends on the distance of the source to the current server.

We define $frac_{s,t}$, as the fraction of server s dedicated to the source t. Consider A as the adjacency matrix of the server graph, and $A[s,t]$ as the value in row s and column t of it. For any given distance d, we allocate $A^d[s,t] \times \frac{c}{b^{2\cdot d-1}}$[1] fraction of server t to s. If there is a leftover capacity in a source, we dedicate it to the same source. See an example of capacity division inside a server in Figure 2a.

A *fractional tree* of a source s, is source tree of s in which we only consider the capacities allocated to source s. See an example of fractional tree in Figure 2b.

Virtual source. The virtual source of an item v is initially its source, $s = source(v)$. The virtual source of an item changes, when that items reach a leaf of the $GTree(s)$. This way, we can ensure that v can always find a spot to placed in, even if its source is heavily loaded. See an example of capacity assigned to a server in its Tree in Figure 2b.

Similar to the *A-DACH*, *L-DACH* (as an acronym for local demand-aware consistent hashing) starts by sorting items based on their frequencies. It then goes over the items of the sorted list, one by one. For each item, it finds the closest server, using the local routing, that has an empty spot in its fraction of capacity for the virtual source of the item.

▶ **Theorem 2.** *The cost of* L-DACH *is constant approximation of the cost of any optimal algorithm, for de Bruijn server graphs, and a constant multiplicative factor, we have a constant approximation of the cost compared to an optimal offline and static algorithm.*

5 Conclusion

This paper introduced new consistent hashing methods that enhance both access time and storage time, while considering the possibility of local search. We discussed *A-DACH* and *L-DACH*, optimized for storage utilization and having local search, respectively. As a future work, we will be looking into the online variant of the problem, in which we aim to provide a dynamic optimal algorithm for our model.

[1] $A^d[s,t]$ determines the number of walks from s to t, and remember that b is the outgoing degree of a server.

References

1 Anders Aamand, Jakob Bæk Tejs Knudsen, and Mikkel Thorup. Load balancing with dynamic set of balls and bins. In *ACM-SIGACT STOC*, 2021.

2 Vamsi Addanki, Maciej Pacut, Arash Pourdamghani, Gabor Retvari, Stefan Schmid, , and Juan Vanerio. Self-adjusting partially ordered lists. In *IEEE INFOCOM*, 2023.

3 Sara Ahmadian, Hossein Esfandiari, Vahab S. Mirrokni, and Binghui Peng. Robust load balancing with machine learned advice. In *ACM-SIAM SODA*, 2022.

4 Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE*, 2012.

5 Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. On the complexity of traffic traces and implications. In *ACM SIGMETRICS*, 2020.

6 John Chen, Benjamin Coleman, and Anshumali Shrivastava. Revisiting consistent hashing with bounded loads. In *AAAI*, 2021.

7 Nicolaas Govert De Bruijn. A combinatorial problem. *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam*, 1946.

8 Dah-Jyh Guan. Generalized gray codes with applications. In *PROC NATL SCI COUNC REPUB CHINA PART A PHYS SCI ENG*. Citeseer, 1998.

9 Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USENIX USITS*, 2003.

10 Qingyun Ji, Darya Melnyk, Arash Pourdamghani, and Stefan Schmid. Towards demand-aware peer selection with xor-based routing. In *SSS*. Springer, 2025.

11 David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM STOC*, 1997.

12 David R. Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Comput. Networks*, 1999. doi:10.1016/S1389-1286(99)00055-9.

13 Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.*, 2010. doi:10.1145/1773912.1773922.

14 F Thomson Leighton. *Introduction to parallel algorithms and architectures: Arrays · trees · hypercubes*. Elsevier, 2014.

15 Vahab S. Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. Consistent hashing with bounded loads. In *ACM-SIAM SODA*, 2018.

16 Moni Naor and Udi Wieder. Novel architectures for P2P applications: the continuous-discrete approach. In *SPAA*. ACM, 2003. doi:10.1145/777412.777421.

17 Arash Pourdamghani, Chen Avin, Robert Sama, and Stefan Schmid. Seedtree: A dynamically optimal and local self-adjusting tree. In *IEEE INFOCOM*, 2023.

18 Arash Pourdamghani, Chen Avin, Robert Sama, Maryam Shiran, and Stefan Schmid. Hash & adjust: Competitive demand-aware consistent hashing. In *OPODIS*, 2024.

19 Andrew Rodland. Improving load balancing with a new consistent-hashing algorithm. *Vimeo Engineering Blog, Medium*, 2016.

20 Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *ACM SIGMOD*, 2012.

21 Ion Stoica, Robert Tappan Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM TON*, 2003.

22 Willy Tarreau et al. Haproxy-the reliable, high-performance tcp/http load balancer, 2012. URL: https://www.haproxy.org.