Access Strategies for Network Caching

Itamar Cohen*, Gil Einziger*, Roy Friedman†, and Gabriel Scalosub*

*Ben-Gurion University of the Negev, Beer Sheva, Israel
†Technion, Haifa, Israel

Email: itamarq@post.bgu.ac.il, gilein@bgu.ac.il, roy@cs.technion.ac.il, sgabriel@bgu.ac.il

Abstract—Having multiple data stores that can potentially serve content is common in modern networked applications. Data stores often publish approximate summaries of their content to enable effective utilization. Since these summaries are not entirely accurate, forming an efficient access strategy to multiple data stores becomes a complex risk management problem.

This paper formally models this problem, and introduces practical algorithms with guaranteed approximation ratios, and in particular we show that our algorithms are optimal in a variety of settings. We also perform an extensive simulation study based on real data, and show that our algorithms are more robust than existing heuristics. That is, they exhibit near optimal performance in various settings whereas the efficiency of existing approaches depends upon system parameters that may change over time, or be otherwise unknown.

I. INTRODUCTION

Having access to multiple network connected data stores is common in modern network settings such as 5G in-network caching [1], [2], content delivery networks (CDN) [3], [4], information centric networking [5], [6], wide-area networks [7], as well as in any multi data center Internet company. Data stores can be cache enabled network devices, memory layers within a server, virtual machines, physical hosts, remote data centers or any combination of the above examples. In such settings, each data store acts as a network cache by holding a potentially overlapping fraction of the entire data that may be accessed by applications and services hosted in the network.

Accessing a data store incurs a certain cost in terms of latency, bandwidth, and energy. Hence, smart utilization of data stores may reduce the operational costs of such systems and improve their users’ experience. Naturally, knowing which item is stored in each data store at any given moment is a key enabler for efficient utilization, but maintaining such knowledge may not be feasible. Instead, it is more practical to occasionally exchange space efficient indicators for the content of the data stores [7]. Bloom filters [8] are a common implementation for such indicators, but many other space-efficient approximate membership representations can also be used [3], [9]–[14].

The shortcoming of relying on such indicators is that they may exhibit false positives, meaning that they may indicate that a given item is held by a certain data store while it is actually not there. Indeed, the work of [13] formally showed that naively relying on indicators for accessing even a single data store may do more harm than good. In this work, we are interested in the general case of accessing multiple data stores. The difference is that we require an access strategy that selects a subset of the data stores to access per request. Existing strategies for this problem include: (i) the Cheapest Positive Indication (CPI) [10], [15] strategy that accesses the cheapest data store with a positive indication for the requested item, and (ii) the Every Positive Indication (EPI) [7] strategy that accesses every data store with a positive indication. The access is considered successful if the item is stored in one of the accessed data stores, and incurs no further cost. Otherwise, we pay a miss penalty for retrieving the requested item, e.g., due to the need to fetch it from an external remote site.

In the example of Figure 1a, CPI accesses only data store 1, which is the cheapest with a positive indication (captured by \( I(x) = \text{Yes} \)), and incurs a cost of 1 for this. However, since \( x \) is not in data store 1 (captured by \( C(x) = \text{No} \)), this indication is a false positive, and an additional miss penalty of 100 is incurred for the request, for a total cost of 101 imposed on CPI. Alternatively, the EPI policy accesses every data store with a positive indication (data stores 1, 2, and 3). This implies an

![Fig. 1. Motivation for the access strategy problem.](image-url)
space can be conserved by allowing a small number of false positives. Bloom filters [8] offer space-efficient encoding but do not support the removal of items. Other works [7], [11], [12], [14], [16] improve on them in various aspects, such as support for removals [12], [17], [18], a more efficient access pattern [11], [14], and lower transmission overheads [19].

### B. Applicability Examples

Bloom filter variants are extensively used in multiple domains [9], [10]. Most notable is their use in front of a cache or a slow memory hierarchy. Such usage leverages that Bloom filters do not exhibit false negatives. Thus, there is no need to access the data store on a negative indication.

The work of [7] suggests an architecture for distributed caching on wide area networks. In this solution, caches share an approximation of their content. Clients use this information to only contact the caches with positive indications (EPI). A similar architecture is also considered in [10], [15]. There, clients access the cheapest cache with a positive indication (CPI). Let us note that the impact of the access strategy and its optimization is overlooked in previous works.

The work of [13] considers the special case of a single data store, equipped with a Standard Bloom Filter [8] or a Counting Bloom Filter [20]. They identify cases where following a positive indication may increase the overall cost. Thus, they suggest that in those cases the data store should be ignored, regardless of its indicator value. We, on the other hand, address the more general problem, which involves any number of data stores, equipped with any kind of indicators.

### III. System Model and Preliminaries

This section formally defines our system model and notations. For ease of reference, our notation is summarized in Table I. We consider a set \( N \) of \( n \) data stores, containing possibly overlapping subsets of items. We denote by \( S_j \) the set of items stored at data store \( j \). Given a sequence of requests for items \( \sigma \) (with possible repetitions), the hit ratio of a data store \( j \) is

---

**Table I: List of Symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>All data stores</td>
</tr>
<tr>
<td>( n )</td>
<td>Number of data stores, ( n =</td>
</tr>
<tr>
<td>( N_k )</td>
<td>Data stores with positive indications for requested datum ( x )</td>
</tr>
<tr>
<td>( n_k )</td>
<td>Number of positive indications for requested datum ( x ) in (</td>
</tr>
<tr>
<td>( S_j )</td>
<td>The set of data items in data store ( j )</td>
</tr>
<tr>
<td>( p_j^h )</td>
<td>Hit ratio of data store ( j )</td>
</tr>
<tr>
<td>( I_j(x) )</td>
<td>Indication of data store ( j ) for datum ( x )</td>
</tr>
<tr>
<td>( q_j )</td>
<td>Probability of positive indication by ( I_j ); ( \text{Pr}(I_j(x) = 1) )</td>
</tr>
<tr>
<td>( FP_j )</td>
<td>False positive ratio for ( I_j ); ( \text{Pr}(I_j(x) = 1</td>
</tr>
<tr>
<td>( \rho_j )</td>
<td>Missindication ratio for a data store ( j )</td>
</tr>
<tr>
<td>( \phi )</td>
<td>Cost function; ( \phi(D) = \sum c_i + \beta \prod D \rho_i )</td>
</tr>
<tr>
<td>( \beta )</td>
<td>Miss penalty</td>
</tr>
<tr>
<td>( M )</td>
<td>( M = \min { \sum_{j \in N_k} c_j, \beta } )</td>
</tr>
<tr>
<td>( H_k(L_k) )</td>
<td>Access cost for the ( k ) highest (lowest) data stores in ( N_k ).</td>
</tr>
</tbody>
</table>
the fraction of requests in $\sigma$ that were available in data store $j$ (when requested). Our work assumes that past hit ratio is a good indication for the near future [21], [22]. We denote by $p_h^j$ the hit ratio of data store $j$, i.e., the probability that the next accessed item $x$ is stored in $S_j$.

Each data store $j$ maintains an indicator $I_j$, which approximates $S_j$: given an item $x$, $I_j(x) = 1$ indicates that $x$ is likely to be in $S_j$ while $I_j(x) = 0$ indicates that it is surely not in $S_j$. These are referred to as a positive indication and a negative indication, respectively. Our model assumes indicators that may exhibit only one-sided errors, i.e., they never err when providing a negative indication\(^1\). In practice, most implementations satisfy this assumption [7], [11], [12], [14]. The false positive ratio of $I_j$ is defined by $FP_j = Pr(I_j(x) = 1|x \notin S_j)$. It captures the probability that given a uniformly random item that is not in $S_j$, the indicator would mistakenly indicate that it is in $S_j$. For every data store $j$, given its indicator $I_j$, we let $\rho_j \in [0, 1]$ denote its misindication ratio $\rho_j = Pr(x \notin S_j|I_j(x) = 1)$, where $x$ is uniformly selected from the entire domain $S_j$.

Given an item $x$ within sequence $\sigma$, a query for $x$ triggers a data access which consists of selecting a subset of the data stores $D$ and accessing this subset in parallel. The data access is considered successful, or a hit, if the item $x$ is found in at least one of the data stores being accessed and is considered unsuccessful, or a miss, otherwise. Since by our assumption all indicators might have a one-sided error, we focus our attention only on subsets of data stores which all provide a positive indication. Given such a subset of the data stores $D$ all providing a positive indication, we denote by $\rho_D$ the misindication ratio of $D$, i.e., the probability that an item is not available in any of the data stores in $D$, in spite of their positive indications. Note, that if $D = \emptyset$, then $\rho_D = 1$. We make no assumptions on the sharing policy among the data stores. Yet, in the analysis sections we assume that the misindication ratios are mutually independent, that is, $\rho_D = \prod_{j \in D} \rho_j$. Under this assumption our analysis provides a baseline for understanding the performance of such systems.

Each data store has some predefined access cost, $c_j$, which is incurred whenever data store $j$ is being accessed. These access costs induce the overall cost for accessing a set $D$ of data stores, defined by $c_D = \sum_{j \in D} c_j$. We assume without loss of generality that $\min_j c_j = 1$. In case the data access results in a miss, it incurs a miss penalty of $\beta$, for some $\beta \geq 1$. For a subset of data stores $D$, which all provide a positive indication, we define its (expected) miss cost by $\beta \cdot \rho_D$.

For any query item $x$, let $N_x \subseteq N$ denote the subset of data stores with a positive indication, i.e., $N_x = \{j \in N|I_j(x) = 1\}$, and denote the size of this set by $n_x = |N_x|$. The expected cost of accessing any $D \subseteq N_x$ is defined to be the sum of its access cost and its expected miss cost, i.e.,

$$\phi(D) = c_D + \beta \cdot \rho_D.$$  

When misindication ratios are mutually independent we have

$$\phi(D) = c_D + \beta \cdot \rho_D = \sum_{j \in D} c_j + \beta \prod_{j \in D} \rho_j.$$

\(^1\)This means having no false negatives, i.e., $Pr(I_j(x) = 0|x \in S_j) = 0$.

The Data Store Selection (DSS) problem is to find a subset of data stores $D \subseteq N_x$ that minimizes the expected cost $\phi(D)$.

We denote by $q_j$ the probability that indicator $j$ positively replies to a query for an item $x$. This happens when either $x \in S_j$; or $x \notin S_j$, and a false positive occurs. Therefore,

$$q_j = Pr(I_j(x) = 1) = p_h^j + (1 - p_h^j)FP_j.$$

Using Bayes’ theorem and Eq. 3, the misindication ratio $\rho_j$ is

$$\rho_j = Pr(x \notin S_j|I_j(x) = 1) = FP_j(1 - p_h^j)/[p_h^j + (1 - p_h^j)FP_j].$$

IV. The Fully Homogeneous Case

To gain some insight about the challenges in developing an access strategy, we start with a simplified fully-homogeneous case. In this setting, the cost of accessing each data store is the same ($c = 1$). The per data store hit ratios and false positive ratios are uniform, i.e., for each $j$, $p_h^j = p_h^0$ and $FP_j = FP$, for some constants $p_h^0$, $FP \in [0, 1]$. Consequently, the per data store misindication ratios, captured by Eq. 4, are also uniform, i.e., for each $j$, $\rho_j = \rho$ for some constant $\rho \in [0, 1]$. Recall that our objective is to pick a subset of data stores with positive indications, $D \subseteq N_x$, so as to minimize the overall expected cost of a query, $\phi(D) = \sum_{j \in D} c_j + \beta \prod_{j \in D} \rho_j$. In the fully-homogeneous case considered here, the expected cost reduces to $\phi(D) = |D| + \beta \rho^{|D|}$, which merely depends on the size of the chosen set $D$ of data stores to be accessed. The task of choosing which subset of data stores to access is reduced to deciding on the number $0 \leq k \leq n_x$ of data stores one should access. For any such potential number $k$, we denote the expected cost of accessing $k$ data stores by

$$\hat{\phi}(k) = k + \beta \rho^k,$$

and focus our attention on studying the cost $\hat{\phi}(\cdot)$ incurred by different data store selection schemes.

The size of the selected subset is clearly upper-bounded by the number of positive indications, $n_x$. So we start by calculating the distribution of $n_x$. Ideally, one can interpret each positive indication as a result of an independent Bernoulli trial with success probability $q$. By Eq. 3, $q = p_h^0 + (1 - p_h^0)$ FP. Hence, $n_x$ is binomially distributed such that

$$Pr(n_x = k) = \binom{n}{k} q^k (1 - q)^{n-k}.$$  

Using equations 5 and 6 we now derive the expected costs of several selection schemes, where we let $D_X$ denote the set of data stores selected by selection scheme $X$.

The EPI policy accesses all the data stores with positive indications, and therefore its expected overall cost is

$$\phi(D_{EPI}) = \sum_{k=0}^{n} \left[ Pr(n_x = k) \cdot \hat{\phi}(k) \right] = \sum_{k=0}^{n} \left[ Pr(n_x = k) \cdot k \right] + \beta \cdot \sum_{k=0}^{n} \left[ Pr(n_x = k) \cdot \rho^k \right] = E[n_x] + \beta \cdot \text{PGF}_{n_x} (\rho).$$

$$= n \cdot q + \beta (1 - q + q \cdot \rho)^n.$$

1
where PGF \(X(t)\) denotes the probability generating function for random variable \(X\) at point \(t\).

CPI accesses either a single data store with a positive indication, if one exists, or no data store if there are no positive indications. The expected overall cost of CPI is therefore

\[
\phi(D_{\text{CPI}}) = \Pr(n_x = 0) \cdot \hat{\phi}(0) + \Pr(n_x > 0) \cdot \hat{\phi}(1) = (1 - q)^n \beta + [1 - (1 - q)^n] (1 + \beta \rho).
\]  

(8)

We now turn to analyze the false-positive-aware optimal policy, FPO, which minimizes the expected overall cost, given the false positive ratio, \(FP\). In the fully homogeneous case, this translates to finding \(\arg \min_k \phi(k)\). Consider \(\phi(y)\) defined in Eq. 5 as a function defined over the reals. This function is convex since its second derivatives is non-negative, and it obtains its minimum at \(y^* = -\ln(-\beta \ln(\rho))/\ln(\rho)\) for \(0 < \rho < 1\). In practice, the number of data stores accessed must be an integer between 0 and \(n_x\). The optimal number \(m^*(k)\) of data stores to access given that there are \(k\) positive indications satisfies \(m^*(k) \in \{0, k, \lfloor y^* \rfloor, \lceil y^* \rceil\}\), where \(\lfloor y^* \rfloor\) and \(\lceil y^* \rceil\) should be considered only if \(y^* \in [0, k]\). Hence, The expected overall cost of FPO is

\[
\phi(D_{\text{FPO}}) = \sum_{k=0}^{n} \binom{n}{k} q^k (1-q)^{n-k} m^*(k).
\]  

(9)

Having studied the overall cost of the above policies, we may revisit Figure 1b. The expected costs of each of the policies are presented as a function of \(\rho\), using Equations 7-9. In particular, in the special case where \(FP = 0\), the expected overall costs of CPI, FPO and the perfect indicators benchmark are identical. This fits our intuition that when there are no false indications, the optimal policy is to access a single data store among those with positive indications if such a data store exists. At the other extreme, we have the case where \(FP = 1\), in which we always have \(n_x = n\), i.e., all the indicators are positive. This extreme case renders the indicators useless and is thus equivalent to not having indicators at all. In particular, note that depending on the values of \(n\) and \(\beta\), EPI might end up being worse than not having any indicators at all.

V. THE HETEROGENEOUS CASE

In the previous section, we addressed the fully homogeneous case, in which minimizing our objective function \(\phi(D)\) was made tractable due to the uniformity of the settings. In general, many systems are heterogeneous, making the minimization of \(\phi(D)\) a much more challenging task.

Recall that our goal is to select a subset \(D \subseteq N_x\) of data stores with positive indications minimizing the expected cost

\[
\phi(D) = c_D + \beta \rho_D = \sum_{i \in D} c_j + \beta \prod_{j \in D} \rho_j,
\]

as defined in Eq. 2. This can be viewed as a combined bicriteria optimization problem, of minimizing two objectives simultaneously: (i) \(c_D\), which is monotone non-decreasing as we pick more data stores to include in \(D\), and (ii) \(\rho_D\), which is monotone non-increasing as we pick more data stores to include in \(D\), where the latter objective is “regularized” by \(\beta\).

In this section, we describe several algorithms for solving the DSS problem in fully heterogeneous settings and provide a rigorous analysis of their performance. In particular, we also study trade-offs between the time complexity and the performance guarantees of our proposed solutions.

A. A Potential-based Algorithm

In the special case where the non-decreasing orderings of data stores by access costs and by misindication ratios are the same, a simple substitution argument shows that a greedy approach will yield an optimal solution \(D\) which consists of a prefix of this ordering.

In what follows we generalize the above observation and suggest an algorithm for the general case based on the special case described above. We denote by \(L_k\) and \(H_k\) the sum of the \(k\) smallest access costs of data stores in \(N_x\) and the \(k\) largest access costs of data stores in \(N_x\), respectively. Our algorithm, \(\text{DSPot}\), described in Algorithm 1, considers the data stores ordered in non-decreasing order of miss-ratio, \(\ell_1, \ldots, \ell_{n_x}\), such that \(\rho_{\ell_j} \leq \rho_{\ell_{j+1}}\) for all \(j = 1, \ldots, n_x - 1\). The algorithm iterates over all prefixes of indices in this order, and picks a subset of data stores corresponding to a prefix which minimizes the potential function \(P(k) = L_k + \beta \prod_{j=1}^{k} \rho_{\ell_j}\).

We now turn to analyze the performance of our proposed algorithm \(\text{DSPot}\). In particular, we show the following theorem:

Theorem 1. Let \(D^*\) be an optimal set of data stores for the DSS problem, and let \(D\) be the solution found by \(\text{DSPot}\). Then

\[
\phi(D) \leq \frac{H_k}{L_k} \phi(D^*).
\]

Proof. Let \(k = |D|\). We therefore have

\[
\phi(D) = \sum_{j=1}^{k} c_{\ell_j} + \beta \prod_{j=1}^{k} \rho_{\ell_j} \leq \frac{H_k}{L_k} \left( L_k + \beta \prod_{j=1}^{k} \rho_{\ell_j} \right) = \frac{H_k}{L_k} P(k),
\]

(10)

where the penultimate inequality follows from the definitions of \(L_k\) and \(H_k\), and the last equality follows from the definition of the potential function \(P(k)\). Let \(k^* = |D^*|\). Since data stores are ordered in non-decreasing order of misindication ratio, it follows that \(\prod_{j=1}^{k^*} \rho_{\ell_j} \leq \prod_{j \in D^*} \rho_{j}\), and by the definition of \(L_{k^*}\) as the sum of the \(k^*\) smallest access costs of data stores in \(N_x\), it follows that

\[
P(k^*) = L_{k^*} + \beta \prod_{j=1}^{k^*} \rho_{\ell_j} \leq \sum_{j \in D^*} c_j + \beta \prod_{j \in D^*} \rho_j = \phi(D^*).
\]

(11)

Since \(D\) is chosen to be the set of data stores that minimizes \(P(k)\), where \(k\) is the length of the prefix \(N_x\) considered in non-decreasing order of miss-ratio, we have \(P(k) \leq P(k^*)\). Combining this with Eqs. 10 and 11, the result follows. □
Since for every $k$ we have $H_n \leq \max_j \{c_j\}$ and the running time of $\text{DS}_{\text{Pot}}$ is dominated by the time required to sort the data stores, we obtain the following corollary:

**Corollary 2.** $\text{DS}_{\text{Pot}}$ is a $(\max_j \{c_j\})$-approximation algorithm, running in time $O(n_x \log n_x)$.

In particular, Corollary 2 implies that for the case where all accesses costs are equal, $\text{DS}_{\text{Pot}}$ yields an optimal solution to the DSS problem.

### B. A Knapsack-based Algorithmic Framework

In this section, we develop an alternative algorithm for the DSS problem and provide guarantees on its performance. We begin by recalling that the main difficulty in solving the DSS problem stems from the fact that our objective function is composed of a linear component (the access cost) and a multiplicative component (the miss cost). The algorithmic framework we propose in the sequel is based on carefully linearizing the multiplicative component, and defining a collection of knapsack problems for which their solution space contains a good approximate solution to the DSS problem.

We associate each data store $j$ with its log-hit weight, defined by $w_j = -\log(\rho_j)$. We therefore have for every subset of data stores $D \subseteq N_x$: $-\log(\rho_D) = \sum_{j \in D} w_j$. Therefore, any set of data stores has a minimal miss cost if and only if it has a maximal log-hit weight. In what follows we define a collection of Knapsack problems, where the Knapsack problem is defined as follows: Given a budget $B$, and collection of items $U$, such that each item $j \in U$ has some profit $\pi_j$ and cost $\gamma_j$, the goal is to find a subset of items $S \subseteq U$ such that $\sum_{j \in S} \gamma_j \leq B$ and $\sum_{j \in S} \pi_j$ is maximized. We refer to such an instance as the $(B, U, \pi, \gamma)$-Knapsack problem, and denote by $\text{A}_{\text{Knap}}(B, U, \pi, \gamma)$ the set of items produced as output by an algorithm $\text{A}_{\text{Knap}}$ for the Knapsack problem. The Knapsack problem is known to be NP-hard, but it can be solved exactly by dynamic programming in pseudo-polynomial time, and can be approximated to within a $(1+\epsilon)$ factor in polynomial time by an FPTAS [23].

We now turn to define our collection of knapsack problems, to be used by our algorithm for solving the DSS problem. We recall that given a query $x$, $N_x \subseteq N$ denotes the subset of data stores for which their indicator is positive. In the following we let $M = \min \{\sum_{j \in N_x} c_j, \beta\}$. Clearly, $M$ is an upper bound on the access cost of any optimal solution for the DSS problem. For any $B \in \{0, 1, \ldots, M\}$, consider the $(B, N_x, w, c)$-Knapsack problem, i.e., the Knapsack problem with budget $B$ over a collection of items $N_x$, such that each item $j \in N_x$ has profit $w_j$ (the log-hit weight of data store $j$) and cost $c_j$ (the access cost of data store $j$).

Our algorithm named $\text{DS}_{\text{PP}}$, formally defined in Algorithm 2, makes use of a $(1+\epsilon)$-approximation algorithm $\text{A}_{\text{Knap}}$ for the knapsack problem, for some $\epsilon \geq 0$. The complexity and performance guarantee depends upon the value of $\epsilon$. $\text{DS}_{\text{PP}}$ essentially iterates over all possible values for the access cost, and solves the associated Knapsack problem using the algorithm $\text{A}_{\text{Knap}}$ as a subroutine for each such value. $\text{DS}_{\text{PP}}$ then selects the subset of data stores $D \subseteq N_x$ which minimizes $\phi(D)$ over all Knapsack solutions calculated by $\text{A}_{\text{Knap}}$ in all iterations.

We first show that if $\text{A}_{\text{Knap}}$ finds an optimal solution to the Knapsack problem in each iteration, then our algorithm finds an optimal solution to the DSS problem. In terms of running time, since the best exact algorithm for the Knapsack problem over $n$ items with budget $B$ runs in pseudo-polynomial time of $O(nB)$ [23], our algorithm also runs in pseudo-polynomial time. These properties are formalized in the following theorem:

**Theorem 3.** When using the pseudo-polynomial algorithm $\text{A}_{\text{Knap}}$ which finds an optimal solution to the Knapsack problem over $n$ items with budget $B$ in time $O(nB)$, $\text{DS}_{\text{PP}}$ is a pseudo-polynomial algorithm that finds an optimal solution to the DSS problem in time $O(n_x M^2)$.

**Proof.** We first show that $\text{DS}_{\text{PP}}$, defined in Algorithm 2, finds an optimal solution to the DSS problem. Consider an optimal solution $D^* \subseteq N_x$ for the DSS problem, and let $B^* = c_{D^*}$. By optimality $B^* \leq M$, we are guaranteed that $\text{DS}_{\text{PP}}$ finds $B = B^*$ in one of the iterations of the for-loop in lines 2-5. Let $D_B$ denote the solution of the knapsack problem being solved in that iteration, where the knapsack budget is $B$. Since algorithm $\text{A}_{\text{Knap}}$ finds an optimal solution for the knapsack problem in this iteration

$$D_B = \arg \max_{D \subseteq N_x} \left\{ \sum_{j \in D} w_j \right\}.$$  

By the definition of $w_j$ and the monotonicity of the log function, such a $D_B$ also satisfies

$$D_B = \arg \min_{D \subseteq N_x \mid D \leq B} \left\{ \rho_D \right\}. \tag{12}$$

Assume by contradiction that $D_B$ is not optimal for the DSS problem, i.e., that $\phi(D_B) = c_{D_B} + \beta \rho_{D_B} > c_{D^*} + \beta \rho_{D^*} = \phi(D^*)$. Since $c_{D^*} = B^* = B \geq c_{D_B}$, it must follow that $\rho_{D_B} > \rho_{D^*}$, for $c_{D^*} \leq B$, which contradicts Eq. 12.

**Running time:** $\text{DS}_{\text{PP}}$ performs $M$ iterations, where in each iteration it solves a knapsack problem using an algorithm which runs in $O(n_M)$ time. It follows that the running time of $\text{DS}_{\text{PP}}$ in this case, is $O(n_x M^2)$, as required. \hfill $\square$

In many cases, the value of $M = \min \{\sum_{j \in N_x} c_j, \beta\}$ is polynomially bounded by $n_x$. The following is an immediate corollary of Theorem 3 in such cases:

**Corollary 4.** If $M = \min \{\sum_{j \in N_x} c_j, \beta\}$ is polynomially bounded by $n_x$, then $\text{DS}_{\text{PP}}$ solves the DSS problem in polynomial time.
We now turn to study the tradeoff between the running time of DSpP and its performance guarantee, when using a polynomial time approximation algorithm for Knapsack instead of the pseudo-polynomial time exact algorithm. We first show in Theorem 5 how the approximation guarantee of an algorithm for Knapsack translates to an approximation guarantee for the DSS problem, while still in pseudo-polynomial time.

**Theorem 5.** If there exists some constant \( \delta \) such that \( \rho_j \leq \delta \) for all \( j \in N_x \) and algorithm \( A_{\text{Knapsack}} \) is a \((1 + \epsilon)\)-polynomial time approximation algorithm for Knapsack running in time \( O(f(n_x, \epsilon)) \), then DSPP is a pseudo-polynomial algorithm that finds an \( O(\beta^{1+\epsilon}) \)-approximate solution for the DSS problem in time \( O(f(n_x, \epsilon) \cdot M) \).

**Proof.** First, note that by its definition, the running time of DSPP is as required since it makes \( M \) iterations, and in every iteration solves an instance of Knapsack in time \( O(f(n_x, \epsilon)) \). It remains to bound the approximation ratio of DSPP.

Consider an optimal solution \( D^* \subseteq N_x \) to the DSS problem, and let \( B^* = c_{D^*} \) and \( \ell \) be an integer such that
\[
2^{-\ell(1+\epsilon)} \leq \rho_{D^*} \leq 2^{-\ell}. \tag{13}
\]
By our assumption there exists some constant \( \delta \) such that for all \( j \in N_x \) we have \( \rho_j \leq \delta \). We are therefore guaranteed to have \( \ell = O(\log \beta) \), since for \( \ell > \log_1 \beta \) we have \( \rho_{D^*} < 1 \), in which case the optimal solution would not benefit from accessing more data stores than it currently does. By the definition of the log-hit weight, we therefore have \( \ell \leq \sum_{j \in D^*} w_j \leq \ell + 1 \).

Consider the iteration of DSPP where \( B = B^* \), and let \( D_B \) denote the solution obtained by algorithm \( A_{\text{Knapsack}} \) for solving the Knapsack problem in this iteration. Since \( A_{\text{Knapsack}} \) is a \((1 + \epsilon)\)-approximation algorithm we are guaranteed to have \( \sum_{j \in D_B} w_j \geq \frac{1}{1+\epsilon} \sum_{j \in D^*} w_j \) since \( D^* \) is an optimal solution with an access cost of \( B^* \), and therefore maximizes the objective function in the Knapsack problem being solved in this iteration. It follows that
\[
\prod_{j \in D_B} \rho_j \leq \prod_{j \in D^*} \rho_j^{1+\epsilon} \leq 2^{\frac{\ell}{1+\epsilon}} \leq 2^{1+\frac{\ell}{1+\epsilon} \log (\beta)} \prod_{j \in D^*} \rho_j, \tag{14}
\]
where the first inequality follows from our Knapsack approximation guarantee, the following two inequalities follow from Eq. 13, and the last inequality follows from the fact that \( \ell = O(\log \beta) \). For \( B = B^* \) we are guaranteed to have \( \sum_{j \in D_B} c_j \leq B^* \). Hence,
\[
\phi(D_B) = \sum_{j \in D_B} c_j + \beta \prod_{j \in D_B} \rho_j \\
\leq B^* + O(\beta^{1+\epsilon}) \left( \beta \prod_{j \in D^*} \rho_j \right) \\
= \sum_{j \in D^*} c_j + O(\beta^{1+\epsilon}) \left( \beta \prod_{j \in D^*} \rho_j \right) \\
\leq O(\beta^{1+\epsilon}) \left( \sum_{j \in D^*} c_j + \beta \prod_{j \in D^*} \rho_j \right) \\
= O(\beta^{1+\epsilon}) \phi(D^*) \tag{15}
\]
which completes the proof.

---

**Algorithm 3 DSKnapsack(\( N_x, c, \rho, \beta \))**

1: \( w_j \leftarrow \log(\rho_j) \) for all \( j \in N_x \)
2: for \( u \in \{c_j | j \in N_x \} \) do
3: \( N_u^x \leftarrow \{j \in N_x | c_j \leq u\} \), let \( n_u^x = |N_u^x| \)
4: \( k_1, \ldots, k_{n_u^x} \leftarrow N_u^x \) in non-increasing order of \( w_j/c_j \)
5: for all \( 1 \leq t \leq n_u^x \) do
6: \( D^x_t \leftarrow \{k_1, \ldots, k_t\} \)
7: \( D^x_{t+1} \leftarrow \{k_{t+1}\} \)
8: end for
9: end for
10: return \( D = \arg \min_{D \in \{D^x_t \cup \{D^x_{t+1}\} \cup \emptyset \}} \{ \phi(D) \} \)

In what follows, we present a polynomial-time approximation algorithm, DSKnapsack, for the problem, formally defined in Algorithm 3. The algorithm is based on DSPP but avoids the need to iterate over all possible budgets. In particular, DSKnapsack does not make use of a general \((1 + \epsilon)\)-approximation algorithm for solving the Knapsack problem. Instead, DSKnapsack incorporates within its design the specifics of a 2-approximation algorithm for the Knapsack problem, the details of which are presented and discussed in the proof of Theorem 6.

**Theorem 6.** If there exists some constant \( \delta \) such that \( \rho_j \leq \delta \) for all \( j \in N_x \), then Algorithm DSKnapsack is a polynomial \( O(\sqrt{\beta}) \)-approximation algorithm running in time \( O(n_x^2 \log n_x) \).

**Proof.** The algorithm is based on the 2-approximation algorithm for Knapsack [23], which works as follows: given budget \( B \), prune all elements with a cost greater than \( B \). Order all elements in non-increasing order of their profitability, captured by their profit-to-cost ratio. Greedily add elements to the solution, starting from the most profitable one, as long as their overall cost does not exceed the given budget. Once adding an additional element causes a violation of the budget constraint, pick the best out of two candidate solutions: the set of elements accumulated which satisfy the budget constraint, and the first element that caused the violation of the constraint.\(^2\)

The remainder of the proof draws its intuition from the proof of Theorem 5, combined with the properties of the 2-approximation algorithm for Knapsack.

Given some budget constraint \( B \) on the access cost of a solution, consider the 2-approximation algorithm for knapsack when given \( B \) as its budget constraint.

The algorithm first prunes all elements with cost greater than the budget. In particular, there exists some element \( j \) such that \( c_j \) is the maximal cost of an element not violating the budget. DSKnapsack simulates the same pruning by iterating over all potential values for this maximal cost, and maintaining only the data stores with cost not exceeding this maximal cost (lines 2-3). It follows that there is a \( u \in \{c_j | j \in N_x \} \) for which
\[
N_u^x = \{j \in N_x | c_j \leq B\}. \tag{16}
\]

\(^2\)Most common implementations consider the element with maximum profit instead of the first element causing the violation of the budget constraint. However, such an amended choice has no effect on the analysis of the algorithm’s performance.
where \(\text{dist}(i)\) will choose either \(\sum_{j=1}^{t_B} c_{i,j} \leq B\), but \(\sum_{j=1}^{t_B+1} c_{i,j} > B\). The algorithm then picks the best between two possible candidate solutions: the set \(\{1, \ldots, k_{t_B}\}\), and the set \(\{k_{t_B+1}\}\).

Our algorithm iterates over all potential candidates of this form, namely, all sets of data stores \(\{1, \ldots, k_i\}\), and all sets of data stores \(\{k_i\}\). Consider an optimal solution \(D^* \subseteq N_x\) to the DSS problem, and denote by \(B^*\) the access cost contributing to the overall cost of \(D^*\). Consider the iteration of DS\(_{\text{Knap}}\) where \(N^u_x = \{j \in N_x | c_j \leq B^*\}\) (as shown in the argument leading to Eq. 16 such a cost \(u\) necessarily exists).

Consider the items in \(N^u_x\) ordered in non-increasing order of \(w_j/c_j\), and let \(t_{B^*}\) be the first item in the order for which \(\sum_{j=1}^{t_{B^*}} c_{j} \leq B^*\), but \(\sum_{j=1}^{t_{B^*}+1} c_{j} > B^*\). The algorithm will choose either \(\{1, \ldots, k_{t_{B^*}}\}\), which is candidate \(D^*_t\) in the iteration where \(t = t_{B^*}\), of lines 5-5; or it will choose \(\{k_{t_B}+1\}\), which is candidate \(D^*_t\) in the iteration where \(t = t_{B^*} + 1\) of lines 5-8. By the proof of Theorem 5, the best of these two candidate solutions is an \(O(\sqrt{B})\)-approximate solution for the DSS problem, since we are using a 2-approximation algorithm for knapsack, implying \(\epsilon = 1\).

Since DS\(_{\text{Knap}}\) picks the candidate solution with minimal overall cost, the solution returned by the algorithm is itself an \(O(\sqrt{B})\)-approximate solution for the DSS problem. The running time of the algorithm is dominated by the outer for-loop in lines 2-9 which has \(n_x\) iterations, wherein each iteration we order all elements in \(N^u_x\), which takes \(O(n_x \log n_x)\) time. Hence, the overall running time of the algorithm is \(O(n_x^2 \log n_x)\), which completes the proof. \(\square\)

VI. SIMULATION STUDY

This section uses a real access trace and a real content distribution network topology to provide insights into the performance of various access strategies in versatile settings.

A. System Topology and Costs

We use the topology of the OVH [24] content distribution network. The OVH network [24] includes 19 Points of Presence (PoPs) in Europe and North America along with the available bandwidth between PoPs. We interpret each PoP as containing both a data store and a co-located client. Queries are generated at clients and each such query triggers an access to a subset of the data stores according to the prescribed policy.

We assume that clients use the shortest hop-count path between their location and the data store they access. Ties are broken by picking the path with maximal bottleneck link bandwidth. The cost for a client located at node \(i\) to access a data store at node \(j\) is:

\[
c_{i,j} = \left[1 + \alpha \cdot \text{dist}(i,j) + (1 - \alpha) \cdot \frac{T}{\text{BW}(i,j)} \right],
\]

where (i) \(\text{dist}(i,j)\) is the hop-count between node \(i\) and node \(j\), where \(\text{dist}(i,i) = 0\), (ii) \(\text{BW}(i,j)\) is the maximum bottleneck bandwidth of a minimum length path from node \(i\) to node \(j\), where \(\text{BW}(i,i) = \infty\), (iii) \(T\) is a design parameter satisfying \(T \geq \max_{i,j} \text{BW}(i,j)\), that relates the increased cost of having a smaller bandwidth with the increased cost due to having a higher hop-count. Lastly, (iv) \(\alpha\) is a design parameter that helps balance the effects of hop-count distance and bottleneck bandwidth on the cost. In particular, for \(\alpha = 1\) the cost is fully dominated by the hop-count distance and for \(\alpha = 0\) it is fully dominated by the bottleneck bandwidth, regularized by the parameter \(T\). Unless stated otherwise, throughout our simulations we set \(T = \max_{i,j} \text{BW}(i,j)\). Specifically, \(T = 500\) for the OVH network.

Figure 2 presents the histogram of the default access cost used in our evaluation between all pairs of clients and data stores in the OVH network.

B. Data Store Characteristics

Data stores are initially empty, and each can contain a maximum of \(S\) data elements. Once an item is added to a full data store, it evicts an item according to the Least Recently Used (LRU) policy. The indicators are implemented using Counting Bloom Filters [20], each consisting of \(B(S) 8\)-bit counters and 5 hash functions, where \(B(S)\) is chosen as the number of counters required to obtain a target false positive ratio of 0.02 [9]. For example, in most of our simulations we set \(S = 1000\), which implies \(B(S) = 8181\). We assume that up-to-date indicators are available at all time as can be efficiently realized by compressed Bloom filters [19], or by only transmitting the changes as in [3].

Each data store estimates its own misindication ratio by evaluating an exponential moving average over epochs of \(R\) requests made to the data store. Formally, let \(m_j(s,t)\) denote the number of misses occurring at data store \(j\) during the requests \(s+1, \ldots, t\) made to data store \(j\). For any \(t \leq R\) we let the estimated misindication ratio after handling request \(t\) be \(\rho_j(t) = \frac{m_j(0,t)}{t-R}\). For \(t > R\), we let \(\rho_j(t)\) be the most recent estimate over epochs of \(R\) requests, \(\rho_j(t/R \cdot R)\), where for every non-negative integer \(k\) this estimate is updated after handling request \((k+1)R\) such that \(\rho_j((k+1)R) = \delta \cdot m_j(kR, (k+1)R)/R + (1 - \delta) \cdot \rho_j(kR)\). In our simulations, we take \(\delta = 0.1\) and \(R = 100\), as we found this configuration

\[\text{Fig. 2. Histogram of } c_{i,j} \text{ values for the OVH network, based on Eq. 17, using } \alpha = 0.5 \text{ and } T = 500.\]
TABLE II

<table>
<thead>
<tr>
<th>$\beta$</th>
<th>Policy</th>
<th>1 location</th>
<th>3 locations</th>
<th>5 locations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TC</td>
<td>AC</td>
<td>TC</td>
<td>AC</td>
</tr>
<tr>
<td>10^2</td>
<td>CPI</td>
<td>1.32</td>
<td>1.08</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td>EPI</td>
<td>0.00</td>
<td>1.58</td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>DSKnap</td>
<td>0.35</td>
<td>1.31</td>
<td>1.10</td>
</tr>
<tr>
<td></td>
<td>DSPot</td>
<td>0.02</td>
<td>1.56</td>
<td>1.09</td>
</tr>
<tr>
<td>10^3</td>
<td>CPI</td>
<td>13.20</td>
<td>1.08</td>
<td>1.23</td>
</tr>
<tr>
<td></td>
<td>EPI</td>
<td>0.00</td>
<td>1.58</td>
<td>1.02</td>
</tr>
<tr>
<td></td>
<td>DSKnap</td>
<td>0.01</td>
<td>1.58</td>
<td>1.02</td>
</tr>
<tr>
<td></td>
<td>DSPot</td>
<td>0.00</td>
<td>1.58</td>
<td>1.02</td>
</tr>
<tr>
<td>10^4</td>
<td>CPI</td>
<td>132.00</td>
<td>1.08</td>
<td>1.24</td>
</tr>
<tr>
<td></td>
<td>EPI</td>
<td>0.00</td>
<td>1.58</td>
<td>1.01</td>
</tr>
<tr>
<td></td>
<td>DSKnap</td>
<td>0.00</td>
<td>1.58</td>
<td>1.01</td>
</tr>
<tr>
<td></td>
<td>DSPot</td>
<td>0.00</td>
<td>1.58</td>
<td>1.01</td>
</tr>
</tbody>
</table>

C. Traffic Trace, Metrics, and Simulated Scenarios

We used a publicly available Wikipedia trace [25] consisting of 357K read requests to Wikipedia pages during a 5 minute period. Each request in this trace is assigned to a random client issuing the request, and requests appear according to their order in the trace. For handling the requests, we consider the following access policies applied by the clients for choosing the set of data stores to access: (i) CPI, (ii) EPI, (iii) DS_Knap, and (iv) DS_Pot. The evaluation factors the total cost, where all clients are running the same algorithms. We also considered the benchmark performance provided by using perfect indicators (PI). This benchmark is used to normalize the costs of the various policies considered.

In terms of metrics, we measure the following three metrics: First and foremost, the total cost (TC) incurred by each access strategy for serving the entire trace, normalized by the total cost of PI. This is further refined into the total access cost (AC) and the Non Compulsory Miss Penalty (NCMP). These two measures are normalized by the total access cost of PI. The former (AC) captures the cost of accessing the data stores and is likely to be higher for access strategies that access multiple data stores for each item request. The latter (NCMP) accounts for miss penalties incurred by an access strategy despite the fact that the item already exists in one of the data stores. This can happen due to the combined effect of false positives and strategies that do not access all data stores whose indicator is positive, such as CPI. We note that normalizing these performance measures by the total access cost of PI allows us to compare the performance in various settings, while alleviating some of the exogenous effects specific to the scenario being evaluated.

D. Heterogeneous Case (OVH network)

Our first experiment considers a system-wide request distribution policy where an item can only be placed in $k$ data stores that are chosen by a hash function based on the requests’ content. Such a policy is inspired by ideas such as replication and partitioning to increase the hit ratio [26]. The outcome of this evaluation is provided in Table II, where we present the PI normalized results for various $\beta$ and $k$ values. We increased $k$ up to 25% of the 19 data stores in the system. Notice that CPI has the minimal AC in all scenarios, as could be expected by its definition. However, CPI is extremely sensitive to false positives, which are translated to a high NCMP value. EPI, on the other hand, is very effective for $k = 1$ but becomes less attractive as we increase $k$, due to the fact it ends up accessing too many data stores. It has the minimal NCMP but pays too much for access costs. Note that for $\beta = 100$ and $k = 5$ EPI has NCMP > 0 as the total access cost of all positive data stores is often larger than $\beta$ and thus EPI would rather avoid accessing any of the data stores and reverts to paying the miss-penalty $\beta$. The DS_Pot strategy is the most efficient (by a very small margin) for $\beta = 100$ and $k = 1$, and is equal or inferior to DS_Knap in all other cases. Intuitively, DS_Pot optimizes for reducing the miss penalty which in turn results in an increased AC. It always outperforms EPI but is inefficient in cases where the access cost is the dominant part of the cost (similarly to EPI). In contrast, DS_Knap exhibits the all-around best performance. It is the best strategy in most scenarios, but most importantly it is never a bad strategy. Thus, even when it underperforms compared to some other strategy, the differences tend to be marginal. Furthermore, when considering the costs incurred by its potential errors (i.e., its AC and NCMP costs), it demonstrates the best performance compared to the other policies in almost all scenarios, and by a significant margin.

E. Homogeneous Case: Varying Data Store Size

Figure 3 shows an experiment with 19 locations, where we vary the data store sizes. The results are shown for $k = 1$, $k = 3$ and $k = 5$ data stores, with homogeneous access costs throughout the network. In these homogeneous cost settings DS_Knap and DS_Pot are equivalent to the scheme which minimizes the expected overall cost, FPO. Furthermore, their performance is very close to the one achieved with perfect

---

4The trace includes requests made on Sep. 22, 2007, from 06:12 to 06:17
indicators. When $k = 1$, DS$_{Knap}$ and DS$_{Pot}$ behave like EPI, while CPI is inefficient. When $k$ increases, CPI improves (due to fewer non-compulsory misses) while EPI worsens (due to higher access cost). This shows that the existing heuristics are too simplistic to fit all system configurations, thus motivating the need for our algorithms.

VII. DISCUSSION

Our work closes an important knowledge gap concerning indicator based caching in network systems. Namely, it answers the fundamental question of providing a stable access strategy that achieves near-optimal results in a wide variety of scenarios.

Our work starts by showing that the access strategy problem was roughly ignored until now and that the existing solutions are only attractive for some system parameters. That is, their effectiveness is determined by uncontrolled variables that may change throughout the system’s life, and may not be known in advance. In contrast, the algorithms suggested in this work provide provable approximation ratios to the optimal solution and are shown to be near optimal in a variety of system settings.

ACKNOWLEDGEMENTS

The work was supported by the Israel Science Foundation (grants No. 1036/14 and 1505/16).

REFERENCES