Query-Adaptive Online Partitioning of Associated Data for Efficient Retrieval

Ting Zhang Duc A. Tran
Computer Science Department,
University of Massachusetts, Boston
Email: {ting.zhang001,duc.tran}@umb.edu

***Abstract*—Data partitioning is a crucial component of any distributed storage system that wants to scale. For retrieval efficiency, data frequently requested together in the same query should be placed on the same server as much as possible. Although intuitive, this is not easy to be implemented if constrained by load balancing; computationally, it is an NP hard problem. Existing research has offered approximate solutions optimized for a given workload of queries, in which the order as to when each query is received is not considered. This paper initiates a new study on online partitioning algorithms that are *sequentially* optimized for a query sequence. In the new problem, the queries arrive in a stream manner, unknown, and given the option to revise the partition after each query, the objective is to minimize the total query processing cost and data migration cost. We formulate this problem formally, investigate several online heuristics, and evaluate them using simulation.**

I. Introduction

We consider a distributed storage system where the data items need to be partitioned across a number of servers. For optimal server performance, a rule of thumb is to preserve data locality; that is, those data items that are often retrieved for the same query should be placed on the same server. This rule helps avoid the Multiget Hole problem, a well-known phenomenon originally experienced in Facebook's network of memcached servers [1].

The conventional approach to preserving data locality is to store "similar" data on the same server. Here, "similar" refers to some attribute space, such as similar content, similar location, similar time, and same ownership. However, it is difficult to quantify data similarity effectively. If the data partition is optimized to preserve similarity defined on certain data attributes, this is unfair to queries looking for similar data in other attribute spaces. On the other hand, the quality of similarity will degrade if we include all the data attributes to define similarity. This is not to mention the existence of time-varying attributes, e.g., location data of mobile users, that result in expensive frequent updating of the similarity scores (two users are geographically close now, hence their data collocated on the same server, but they may be far away from one another soon).

This paper presents a study on data partitioning without any presumption about similarity. To preserve locality, "associatedness" among the data items is based merely on their co-appearance in the same query. Associated data can be data that belong to the same social community in an online social network, data that are near a given location in location-based services, or data that are about the items frequently purchased together in e-commerce transactions. In general, this study can support both similarity (range/knn) and non-similarity (skyline/itemset) queries.

Research on associated data placement in distributed servers started gaining traction only recently. For example, Curino et al. [2] and Yu and Pan [3] consider this problem assuming known request patterns (how often certain items are requested in the same query). In contrast, we initiate a novel study without this assumption. Furthermore, while earlier research attempts to optimize for a "batch" workload of queries regardless of the order they are issued, ours is focused on "sequential" optimization: place the data items on the servers adaptively such that it is most efficient for evaluating the *next* query. A partitioning solution optimal for a batch of queries may not offer the best sequential costs incurred when processing the queries sequentially in a certain order. Vice versa, a partitioning solution optimal in these sequential costs for a given order may not be optimal in the batch cost.

We call the proposed problem "Query-Adaptive Online Partitioning". Taking as input a stream of queries, each of which can be an arbitrary itemset, we want to devise an efficient online partitioning scheme that determines a balanced partition most beneficial for each next query in the sequence. This decision has to be made on the fly, irrevocably, unknown of future queries, yet aiming to minimize the total sequential costs which consist of the number of servers read and number of items migrated during the entire process.

The contributions made in this paper are as follows. First, the proposed problem is a novel problem, for which we are aware of no earlier research. Unfortunately, it has no optimal solution due to its online nature. There is simply no way to compute a partition that is optimal for the next queries since we do not know what group of data items will be requested next. Even in the offline setup where the entire query sequence is known in advance the problem is already NP-hard. Second, we show that the proposed problem can fit partially into some existing frameworks but there are fundamental challenges that are not even explored in the literature of these frameworks. Third, we present and compare several heuristic online algorithms to assess the benefit of item migration during the query sequence in order to reduce the read cost. We are interested in the following questions:

- Since the future queries are unknown and so is any data association pattern, is there really a benefit of revising the current partition in hopes of better serving future queries? Should we keep the same partition as it is at the beginning? (At least this incurs zero moving cost.)
- If it is worth revising the current partition upon each query, which items should be migrated and whereto among the existing servers? Should the moved items be among those items requested in the current query or can they be arbitrary?
- Do the answers to the above questions apply to different query sequences? Should we disregard any heuristic that is consistently ineffective or is there one that always works better than the others?

The remainder of the paper is structured as follows. Related work is discussed in Section II. The problem's formulation and its correspondence to two well-known frameworks are presented in Section III. Approximate online algorithms are proposed in detail in Section IV. The results of our evaluation study using real-world datasets are analyzed in Section V. The paper concludes in Section VI with pointers to our future work.

II. Related Work

On top of a distributed infrastructure of commodity servers, the common practice for partitioning is based on hashing, e.g., range-based hashing used in Twitter's Gizzard [4] or consistent hashing in Facebook's Cassandra [5] and Amazon's Dynamo [6]. Hashing is random and blind to the desired locality of associated data. As such, the system may suffer from an overwhelming CPU bottleneck on the server side that cannot be resolved just by adding more servers; this is known as the Multiget Hole problem [1]. It has been shown that we can avoid this problem by keeping all of the relevant data of each query local to the same server [7].

Hence arose a series of works aimed to preserve locality in the data partition. SCHISM [2] is a workload-driven partitioning scheme for transactional databases that partitions the data based on transaction patterns such as how often two data items are retrieved together. Partitioning schemes such as SPAR [7], S-PUT [8], and DAROS [9] are aimed to preserve social locality in the data storage of online social networks, so that the data of two socially connected users are located on the same server. Locality preservation that takes into account both social and geographic properties is considered in COSPLAY [10]. For non-social data, as an example, that of a large RDF (Resource Description Framework) graph, subgraph search can be made efficient if the graph is distributed across a number storage nodes, e.g., pages on a disk [11] or compute servers in a cluster [12], such that cross-reference between different storage nodes is minimized.

The underlying approach in these works is to preserve locality based on a preset similarity measure and, consequently, the data partitioning problem can be seen as a graph partitioning problem: model the set of data items as a similarity graph G where each item is a vertex and a weighted edge exists between two vertices to reflect their similarity, and apply a graph partitioning technique to partition G into balanced min-cut clusters, each assigned to a unique server.

Recently, as in [3], hypergraph has been adopted to model data associated-ness, thus allowing associated data to involve any arbitrary number of items, not just two as in earlier works. A hypergraph partitioner [13], [14] can then be used to assign the data to their corresponding servers. In [15], a re-partitioning hypergraph model is introduced for cases where the hypergraph needs to be re-partitioned over the time to adapt to workload changes. Although incorporating both read cost and migration cost in each partition adjustment, this model is not truly "online" and "query-adaptive". Each repartitioning requires a wait window long enough to form a quality hypergraph to represent the query workload during this epoch. In contrast, we focus on minimizing the sequential cost - the immediate cost - to process each query, not a window of queries. Thus, the arrival order of the queries matters.

Our problem can be seen as a constrained version of online hypergraph partitioning where the set of vertices is known but the hyperedges (representing queries) are inserted in a stream manner. Partitioning algorithms exist for streaming standard graphs [16] and streaming hypergraphs [17]. In contrast, ours is the first not only to process a stream of hyperedges as input, but also to cope with constraints (w.r.t cost to change each partition) that make the problem even more challenging.

III. Problem Formulation

Suppose that we have N data items, $\mathcal{O} = [N]$ that need to be distributed among M servers, $\mathcal{S} = [M]$; here, $[z]$ denotes the set $\{1, 2, ..., z\}$. Each item is placed on a server according to an initial partition. Over the time, queries are submitted to the system one by one, each asking for an itemset (a subset of items). Due to changing query workload, the initial partition may no longer be efficient. After each query is processed, we can keep the same partition, or revise it in hopes of reducing the future read costs. On the other hand, this should be done without having to move too many items from one server to another. Consequently, we propose the following data partitioning problem.

Definition 1 (Query-Adaptive Online Partitioning)**.** *Denote the query sequence (unknown in advance) by $q_1 q_2 ... q_T$, where $q_t \subset [N]$ is the query at time t to retrieve a subset of items from the servers; for example, query $\{3, 5, 10\}$ is for retrieving items 3, 5, and 10 from their respective servers. Start with a given initial partition at time $t = 0$, $f_0 : [N] \to [M]$, assigning each item i to some server $j = f_0(i)$. At each subsequent time $t \geq 1$ once query q_t is received, knowing only queries received thus far, $q_1 q_2 ... q_t$, we need to compute a new partition, $f_t : [N] \to [M]$, to assign each item i to some server $j = f_t(i)$.*

Let $r(t) = \left|\bigcup_{i \in q_t} \{f_{t-1}(i)\}\right|$ be the read cost (number of different servers to read) of query q_t and $m(t) = \sum_{i=1}^{N} [f_t(i) \neq f_{t-1}(i)]$ the move cost (number of items migrated to a different server due to the adjustment) to obtain partition f_t from partition f_{t-1}; notation $[.]$ is the Iverson

bracket. Over the entire query sequence, the objective is to minimize the total sequential read cost and the total sequential move cost while keeping the partition balanced:

$$\min_{\{f_t\}_{t=1}^T} \left\{ \Lambda = \sum_{t=1}^{T} \underbrace{\left| \bigcup_{i \in q_t} \{f_{t-1}(i)\} \right|}_{r(t)} \right\} \tag{1}$$

$$\min_{\{f_t\}_{t=1}^T} \left\{ \Gamma = \sum_{t=1}^{T} \underbrace{\sum_{i=1}^{N} [f_t(i) \neq f_{t-1}(i)]}_{m(t)} \right\} \tag{2}$$

$$\text{s. t.} \quad \sum_{i=1}^{N} [f_t(i) = j] \leq C \ \forall j \in [M], t \in [T]. \tag{3}$$

Here, C is the maximum storage capacity allowed for each server. We assume that the set of data items and the set of servers do not change during the query sequence.

It is easy to see that Objective (1) and Objective (2) cannot concurrently be achieved. The move cost is minimum (zero) if the initial partition is never changed during the entire query sequence. The read cost of the initial partition, however, is not optimal. We show below how this problem can be cast as extended versions of the Online Hypergraph Partitioning problem and the Metrical Task Systems problem.

A. Correspondence to Hypergraph Partitioning

Consider a simplified case where we know the entire query sequence in advance and stick with the initial partition f_0 for the entire query sequence ($f_t = f_0 \ \forall t$); hence, no migration allowed ($\Gamma = 0$). The total read cost (see Eq. (1)) becomes:

$$\Lambda_0 = \sum_{t=1}^{T} \left| \bigcup_{i \in q_t} \{f_0(i)\} \right|. \tag{4}$$

We will show that the best f_0 minimizing Λ_0 is one that is a solution to a min-cut hypergraph partitioning problem.

A hypergraph is a generalized graph where an edge, called a hyperedge, can consist of any arbitrary non-empty subset of vertices, not necessarily a pair of vertices as in standard graphs. Our hypergraph to be partitioned is $G = (V, E)$ where $V = [N]$ represents the set of items and $E = \{q_1, q_2, ..., q_T\}$ represents the set of queries. In other words, each item is a vertex and each query is an hyperedge consisting of all the items of this query. Given a partition, a hyperedge of connectivity k (i.e., spanning k parts) is said to be cut if $k \geq 2$ and the weight of this cut is $(k-1)$. A min-cut partition is one that minimizes the total cut weight.

Consider a partition $f : [N] \rightarrow [M]$. Using this partition, each query q_t requires reading $\left| \bigcup_{i \in q_t} \{f(i)\} \right|$ servers and, accordingly, the cut weight of hyperedge q_t is $(\left| \bigcup_{i \in q_t} \{f(i)\} \right| - 1)$. Therefore, minimizing the total read cost is equivalent to minimizing the total cut weight in the hypergraph. To find a partition f_0 that is balanced with minimal total read cost is equivalent to finding a balanced min-cut partition for hypergraph G. The latter problem in general is known to be NP-hard [18], but effective heuristic algorithms have been developed; e.g., hMetis [13] and PaToH [14]. We can use one such algorithm to obtain f_0.

Now, consider the original scenario where queries arrive sequentially, unknown in advance, and migration is allowed so that we can adapt the partition upon receipt of each query. The corresponding hypergraph G is therefore a streaming hypergraph where the set of vertices is known but the set of hyperedges not; instead, the hyperedges are inserted to the hypergraph one at a time. Not only that we need to compute an online balanced min-cut partitioner for this streaming hypergraph, but also this partitioner should incur the least move cost. In the literature of online hypergraph partitioning, this problem is not yet explored.

B. Correspondence to Metrical Task Systems

As an online problem seeking an optimal solution for sequential input, our proposed problem can be viewed in the framework of metrical task systems (MTS). An MTS is a multi-state system for processing sequential tasks, in which the cost to process a task depends on the state of the system and the system can change its state anytime subject to a transition cost metric. The MTS problem, introduced in [19], is to compute an efficient schedule $\boldsymbol{\psi} = \psi_1\psi_2...\psi_T$ for a task sequence $\boldsymbol{\sigma} = \sigma_1\sigma_2...\sigma_T$, where ψ_t is the system state in which σ_t will be processed, such that the total processing and transition cost is minimized,

$$\min \left\{ C_{transition}(\boldsymbol{\psi}) + C_{processing}(\boldsymbol{\psi}, \boldsymbol{\sigma}) \right\},$$

where $C_{transition}(\boldsymbol{\psi}) = \sum_{i=1}^{T} cost_{transition}(\psi_{t-1}, \psi_t)$ and $C_{processing}(\boldsymbol{\psi}, \boldsymbol{\sigma}) = \sum_{i=1}^{T} cost_{processing}(\psi_t, \sigma_t)$. Here, ψ_0 is the initial state (given).

An online scheduling algorithm must compute ψ_t knowing only $\sigma_1\sigma_2...\sigma_{t-1}$. In competitive analysis, an online algorithm is said to be k-competitive iff, given any task sequence input, its cost is at most k times that of an optimal offline algorithm (plus a constant depending only on k). It is known [19] for any MTS with n states that a deterministic online algorithm can be constructed with ($2n$-1) competitive ratio, which is optimal among all deterministic algorithms.

Our partitioning problem can be cast into a MTS. We combine Objective (1) and Objective (2) into a single objective $(\alpha\Lambda + (1-\alpha)\Gamma)$, where weight $\alpha \in [0, 1]$ indicates the priority between read cost versus move cost. Then, the MTS to be optimized is as follows:

- States: The set of states is the set of all partitions $f : [N] \rightarrow [M]$ that satisfy Ineq. (3). The initial state is $\psi_0 = f_0$ (the initial partition in our problem).
- State transition cost: Cost function $cost_{transition}(f, f')$ is defined to be $(1-\alpha)$ times the move cost to change from partition f to partition f'.

- Task processing cost: Task σ_t at time t is the query q_t. Cost function $cost_{processing}(f, q_t)$ is defined to be α times the read cost for query q_t using partition f.

By solving this MTS, we can derive an optimally-competitive online algorithm for our partitioning problem. Unfortunately, the number of states is roughly $n \approx \frac{N!}{(N/M)!^M M!}$ (number of partitions when the capacity C is precisely N/M), too large to be computationally practical. In the literature of metrical task systems, we are aware of no research aimed to substantially downsize the state set to obtain a more efficient algorithm that remains as competitive.

IV. Approximate Algorithms

As discussed in the previous section, the proposed online partitioning problem poses fresh challenges. To investigate this problem, our focus is on approximate greedy algorithms. The greedy heuristic applies when each query arrives and we need to decide where to move which data items. In practice, if two items are included together in a query, it is more likely than not that they will be requested together again. Also, the 80/20 rule suggests that the total set of requested items should be a small portion of the total data set. As we do not know which items will be requested in the future, we conjecture that if some items are to be moved in hopes of reducing read cost for future queries, the best candidates should be the items that are just requested in the current query. Consequently, we investigate several greedy algorithms that either are based on the above intuition or borrow ideas from greedy algorithms for standard streaming graphs.

A. Greedy Heuristics

We define the following quantities:

- Association Score $a^t(i, j)$: the total number of times item i and item j have been co-requested up to time t (after receipt of q_t). This count is incremented each time these two items appear in a new query.
- Demand Score $b^t(i, s)$: the sum of association scores of item i with the items hosted in server s at time t (after receipt of q_t), representing how much demand server s has for item i. Hence, it is always true that $b^t(i, s) = \sum_{j:f_{t-1}(j)=s} a^t(i, j)$.

Four heuristics are considered in our study:

- All-To-Same-Least-Move (ALL-LM): After receipt of query q_t, move all the requested items to the same server with the least move cost. This server is the one hosting the most among the requested items:

$$s^* = \arg\max_{s\in[M]} |q_t \cap \{i \in [N] : f_{t-1}(i) = s\}|$$

- All-To-Same-Highest-Demand (ALL-HD): After receipt of query q_t, move all the requested items to the same server with the highest demand total for the requested items. This server is the below:

$$s^* = \arg\max_{s\in[M]} \left(\sum_{i\in q_t} b^t(i, s) \right)$$

- Individual-Highest-Demand (IND-HD): After receipt of query q_t, move each individual requested item i to the server having the highest demand for i:

$$s^* = \arg\max_{s\in[M]} b^t(i, s)$$

- Individual-Most-Associated (IND-MA): After receipt of query q_t, move each individual requested item i to the server hosting i's most associated item:

$$j^* = \arg\max_{j\in[N]} a^t(i, j);\ s^* = f_{t-1}(j^*)$$

The last two heuristics borrow the ideas from partitioning of standard streaming graphs where we try to put each vertex in the same part with the most neighbors (IND-HD) or with the nearest neighbor (IND-MA).

When applying the above heuristics, to satisfy the balancing constraint (Constraint (3)) at any time, candidate servers can only be among those that will not exceed the capacity after the migration.

B. Implementation Efficiency

In terms of computation, the above heuristic algorithms can be implemented in an efficient manner. The association and demand scores are incrementally updated over the time after each query q_t is received. This is done as follows in $O(|q_t|^2)$ time: (|.| denotes the cardinality)

$$\begin{aligned} &\forall i, j \in q_t \wedge i \neq j : \\ &\qquad a^t(i, j) \;=\; a^{t-1}(i, j) + 1 \\ &\qquad b^t(i, f_{t-1}(j)) \;=\; b^{t-1}(i, f_{t-1}(j)) + 1 \end{aligned}$$

With the above information already computed, to find the desired server(s) takes no worse than $O(|q_t| \times M)$ time for all the heuristic algorithms.

V. Evaluation

We evaluate the heuristic algorithms using four real-world datasets, arXiv, github, retail, and actor-movie, obtained from the dataset repositories at http://konect.uni-koblenz.de and http://fimi.ua.ac.be/data. We transform these datasets into a diverse set of "transaction" hypergraphs so that we can use each vertex to represent a data item and each hyperedge a query. A query sequence in our simulation is a random permutation of the hyperedge set. Some statistics of these datasets are summarized in Table I and Table II.

We compare ALL-LM, ALL-HD, IND-HD, and IND-MA using the benchmark where the initial partition f_0 is obtained by a random partitioning of the items among the servers and is used unchanged for the entire query sequence. This benchmark, referred to as RAND-NoMove hereafter, represents a hash-based partition assignment that does not adapt to the queries. We are interested in how ALL-LM, ALL-HD, IND-HD, and IND-MA compare to each other and how much of an improvement they each offer compared to RAND-NoMove.

The comparison is in terms of read cost and move cost. In the plots discussed below (Figures 1, 2, 3, and 4), the x-axis is the read cost as a percentage of the read cost incurred by

TABLE I
SUMMARY OF DATASETS USED IN EVALUATION

Dataset	No. Items (N)	Query Sequence Length (T)	Min Query Size	Max Query Size	Avg. Query Size	Notes
arXiv	16, 264	17,837	2	18	3	small N, $N \sim T$
Github	42,444	37,837	2	3675	9	medium N, $N \sim T$
Retail	16,407	85,146	2	76	10	small N, $N << T$
Actor-movie	382,218	118,476	2	294	12	large N, $N >> T$

TABLE II
NUMBER OF ITEMS FOR EACH RANGE OF REQUEST FREQUENCIES

Dataset	Total	% of Total for Each Request Frequency				
		1	2	3	4	5+
arXiv	16264	47.4%	17.3%	9.6%	5.8%	19.8%
Github	42444	45%	13%	7.1%	4.8%	30.2%
Retail	16407	13.4%	8.3%	6.4%	5.1%	66.8%
Actor-movie	382218	63.8%	12.2%	5.6%	3.4%	15%

the RAND-NoMove benchmark and the y-axis is the average move cost as a percentage of the average query size in the dataset. For each dataset, five permutations of the set of queries are randomly generated to serve as five query sequences; the results are averaged over these sequences.

Given the sizes of the datasets, two reasonable cases for the number of servers are considered, $M \in \{8, 16\}$. The maximum server capacity C is set to be a factor of $M(\frac{b+50}{100})^{log_2 M}$ times of the average capacity per server, where $b \in (0, 50)$ is the tunable balance parameter; this formula is used in hMetis [13]. We consider two cases: $b = 1$ and $b = 2$. With $M = 8$ servers and $b = 1$, we allow for capacity C to be 6% higher than the average capacity per server (N/M); when $b = 2$, this percentage is 12%. With $M = 16$, these figures are 8% and 16% for $b = 1$ and $b = 2$, respectively.

A. Results for the arXiv dataset (Figure 1)

For this dataset, the following is observed. First, the read cost reduction of each online heuristic seems greater as more servers are deployed (M is increased from 8 to 16) or the balancing constraint is less strict (b is increased from 1 to 2). Using an online heuristic, the read cost ranges from 76% to 83% of that of RAND-NoMove, while the average move cost varies 30%-50% of the average query size.

Second, there is a clear performance gap: the better performing heuristics are ALL-HD and IND-MA and the clearly worse heuristics are IND-HD and ALL-LM. In all four configurations, the best heuristic is IND-MA which offers a read cost that is about 77% of RAND-NoMove and a move cost only 30% of the average query size. Note that the average query size is 3.1 items per query for this dataset, meaning that by moving one item or so in each query time on average, IND-MA offers a read cost that is 23% better than the benchmark.

B. Results for the Github dataset (Figure 2)

Like the previous case, we also observe in Github that the read cost reduction of each online heuristic seems greater as more servers are deployed or the balancing constraint is less strict. Another similar observation is that ALL-HD and IND-MA seem comparable to each other. However, the read cost improvement in Github is not as high as that in the arXiv case; it is 83.5% of the RAND-NoMove cost at best (when $M = 16$ and $b = 2$; see Figure 2(d)). The move cost is also higher; the best move cost is more than 60% of the average query size (9.3). Furthermore, while IND-MA is the best for arXiv, ALL-LM is the best for the Github dataset. ALL-LM is the only heuristic that is Pareto-optimal in all configurations. Not only that, ALL-LM is clearly better than the other heuristics in three out of four configurations.

Note that the Github dataset is larger than the arXiv dataset in every category (number of items, number of queries, maximum query size, and average query size), but they share a common property that the number of hyperedges is similar to the number of vertices.

C. Results for the Retail dataset (Figure 3)

The benefit of applying online heuristics on the Retail dataset is less significant compared to the previous two datasets. At best, the read cost is 89% (of the RAND-NoMove cost) and the move cost is 70% (of the average query size). This could be because the transaction hypergraph is highly dense, where the number of hyperedges is much higher than that of vertices.

However, still, the performance gap is somewhat clear between the heuristics. The best, that are Pareto-optimal in all configurations, are ALL-LM (best move cost) and IND-MA (best read cost). The worst is always IND-HD.

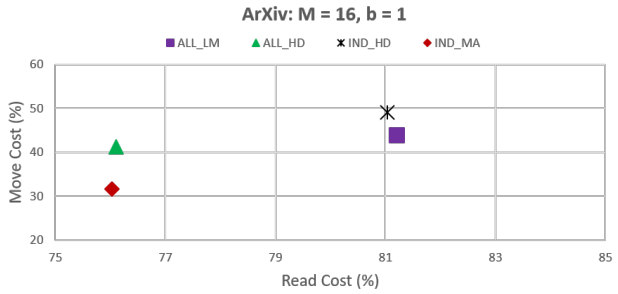
D. Results for the Actor-Movie dataset (Figure 4)

This dataset is the largest in terms of the number of items and the number of queries. It represents a large hypergraph that is highly sparse, in which the number of hyperedges is much smaller than the number of vertices.

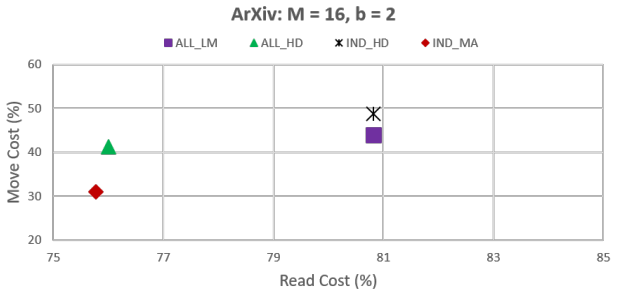
Interestingly, compared to the other datasets which are smaller, the read cost improvement is found much more substantial for this dataset (read cost as low as 62% of the benchmark) while the move cost remains moderate (less than 60%). However, there is no clear overall winner. If minimizing read cost is the priority, the best heuristic is ALL-HD. On the other hand, if the move cost is of importance, the best heuristic

(a) 8 servers, max capacity 6% above average per server

(b) 8 servers, max capacity 12% above average per server



(c) 16 servers, max capacity 8% above average per server



(d) 16 servers, max capacity 16% above average per server

Fig. 1. Comparative results for the arXiv dataset

TABLE III
SUMMARY OF RESULTS: HERE, A HEURISTIC IS MARKED 'X' IF IT IS "RECOMMENDED" FOR THE CORRESPONDING DATASET (PERFORMS CONSISTENTLY BEST), AND 'O' IF "NOT RECOMMENDED" (CLEARLY WORSE THAN MOST OTHER HEURISTICS)

Dataset	arXiv	Github	Retail	Actor-movie
Read Cost	76%-83%	83%-91%	89%-97%	62%-86%
Move Cost	30%-50%	60%-80%	70%-90%	40%-60%
ALL-LM		x	x	
ALL-HD				x
IND-HD	o	o	o	
IND-MA	x		x	x

is IND-MA. We would not recommend ALL-LM because it is always worse than IND-HD.

E. Summary of Results

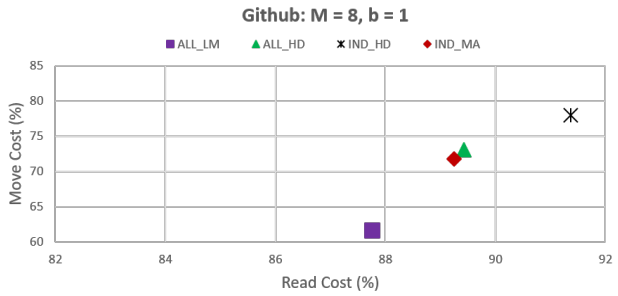
Table III summarizes the read cost range, the move cost range, and the recommended/not-recommended heuristic(s) when applied on each dataset. The online heuristics are effective for arXiv and Actor-movie datasets, but not so for Github and Retail. It seems that IND-MA ("stored where the nearest neighbor is" heuristic) is a safe choice for the online heuristic whereas IND-HD ("stored where the most neighbors are" heuristic) is a risky choice. However, this suggestion is weak due to the limited set of the datasets evaluated. A stronger conclusion would be that if we can move some items, fewer than the average query size, during the process of each query, the presented online heuristics can be effective; it is shown in our study always better than RAND-NoMove. Furthermore, we should not disregard any of these heuristics because our evaluation observes no consensus winner.

VI. CONCLUSIONS

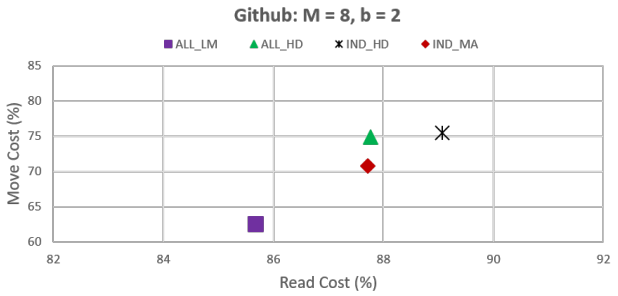
This paper initiates a novel problem on online partitioning that minimizes the sequential read and move costs of processing a query sequence. Although the online decision must be made optimal for the next query which has yet to arrive, our study has shown consistently that there is a real benefit by migrating items between the servers during the query sequence. Four efficient online heuristic algorithms have been presented, all offering better read cost than does the "move-nothing" hash-based partitioning approach, while incurring moderate move costs. Although we do not recommend any specific heuristic as always best, and better heuristics may exist beyond this paper, our study validates the worthiness of the proposed partitioning problem and opens room for future research. Specifically, our work will be extended in several ways: (1) evaluate with more datasets at larger scale, (2) explore online algorithms assuming some known probability pattern about the query sequence, (3) incorporate other constraints such as geo-location related constraints, and (4) albeit the most challenging, provide a competitive analysis on the theoretical bounds of online algorithms for this problem.

REFERENCES

[1] WWW, "Facebook's memcached multiget hole: More machines != more capacity," Source: http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html.

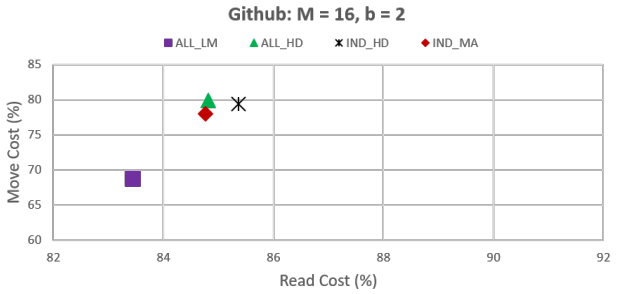


(a) 8 servers, max capacity 6% above average per server



(b) 8 servers, max capacity 12% above average per server

(c) 16 servers, max capacity 8% above average per server



(d) 16 servers, max capacity 16% above average per server

Fig. 2. Comparative results for the Github dataset

[2] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *PVLDB*, vol. 3, no. 1, pp. 48–57, 2010.

[3] B. Yu and J. Pan, "Location-aware associated data placement for geo-distributed data-intensive applications," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*, April 2015, pp. 603–611.

[4] N. Kallen, R. Pointer, E. Ceaser, and J. Kalucki, "Introducing gizzard, a framework for creating distributed datastores," Twitter Engineering Website, April 2006, http://engineering.twitter.com/2010/04/ introducing-gizzard-framework-for.html.

[5] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, April 2010.

[6] 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 Oper. Syst. Rev.*, vol. 41, pp. 205–220, October 2007.

[7] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The little engine(s) that could: scaling online social networks," in *Proceedings of the ACM SIGCOMM 2010 Conference*. New York, NY, USA: ACM, 2010, pp. 375–386.

[8] D. A. Tran and T. Zhang, "S-PUT: an EA-based framework for socially aware data partitioning," *Computer Networks*, vol. 75, pp. 504–518, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2014.08.026

[9] T. Duong-Ba, T. Nguyen, B. Bose, and D. Tran, "Distributed client-server assignment for online social network applications," *Emerging Topics in Computing, IEEE Transactions on*, vol. 2, no. 4, pp. 422–435, Dec 2014.

[10] L. Jiao, J. Li, T. Xu, and X. Fu, "Cost optimization for online social networks on geo-distributed clouds," in *Proceedings of IEEE ICNP (ICNP 2012)*, 2012, pp. 1–10.

[11] M. Broecheler, A. Pugliese, and V. Subrahmanian, "Dogma: A disk-oriented graph matching algorithm for rdf databases," in *8th International Semantic Web Conference (ISWC2009)*, October 2009.

[12] K. Lee and L. Liu, "Scaling queries over big rdf graphs with semantic hash partitioning," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1894–1905, Sep. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2556549.2556571

[13] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," in *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, ser. DAC '99. New York, NY, USA: ACM, 1999, pp. 343–348. [Online]. Available: http://doi.acm.org/10.1145/309847.309954

[14] Ü. Çatalyürek and C. Aykanat, *PaToH (Partitioning Tool for Hypergraphs)*. Boston, MA: Springer US, 2011, pp. 1479–1487.

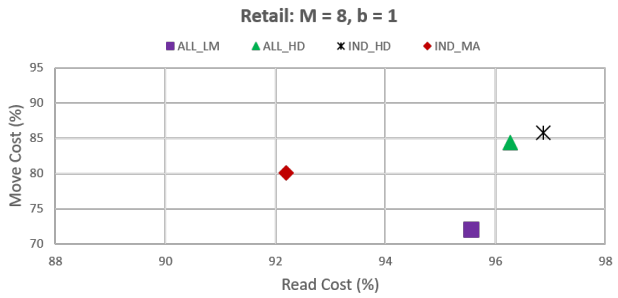
[15] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen, "A repartitioning hypergraph model for dynamic load balancing," *J. Parallel Distrib. Comput.*, vol. 69, no. 8, pp. 711–724, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2009.04.011

[16] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, ser. WSDM '14. New York, NY, USA: ACM, 2014, pp. 333–342. [Online]. Available: http://doi.acm.org/10.1145/2556195.2556213

[17] D. Alistarh, J. Iglesias, and M. Vojnovic, "Streaming min-max hypergraph partitioning," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 1900–1908. [Online]. Available: http://papers.nips.cc/paper/5897-streaming-min-max-hypergraph-partitioning.pdf

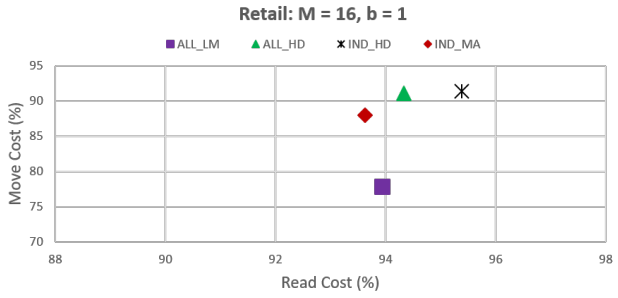
[18] L. Lyaudet, "Np-hard and linear variants of hypergraph partitioning," *Theoretical Computer Science*, vol. 411, no. 1, pp. 10 – 21, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397509006252

[19] A. Borodin, N. Linial, and M. E. Saks, "An optimal on-line algorithm for metrical task system," *J. ACM*, vol. 39, no. 4, pp. 745–763, Oct. 1992. [Online]. Available: http://doi.acm.org/10.1145/146585.146588

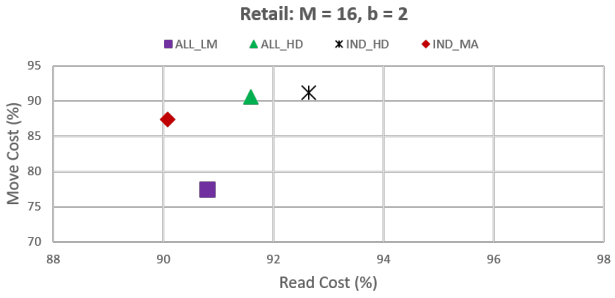


(a) 8 servers, max capacity 6% above average per server

(b) 8 servers, max capacity 12% above average per server

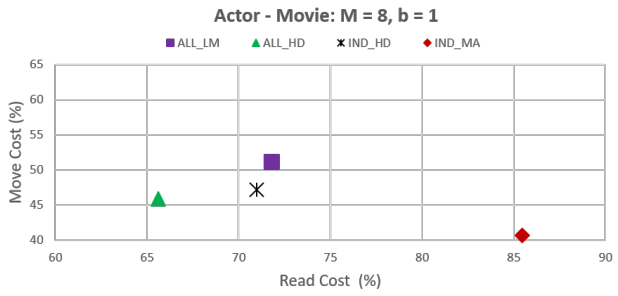


(c) 16 servers, max capacity 8% above average per server

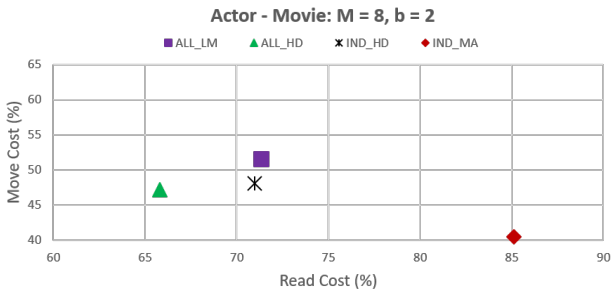


(d) 16 servers, max capacity 16% above average per server

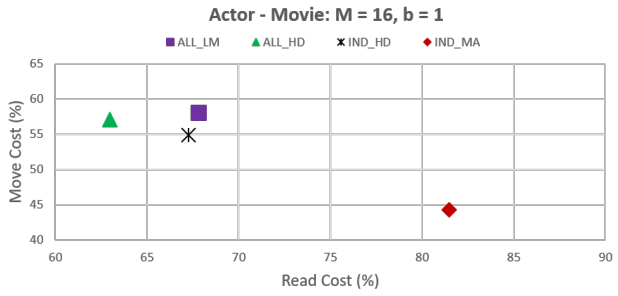
Fig. 3. Comparative results for the Retail dataset



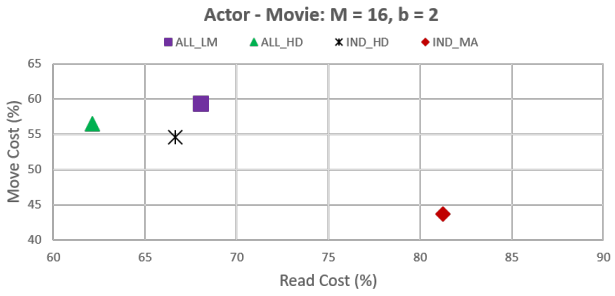
(a) 8 servers, max capacity 6% above average per server



(b) 8 servers, max capacity 12% above average per server



(c) 16 servers, max capacity 8% above average per server



(d) 16 servers, max capacity 16% above average per server

Fig. 4. Comparative results for the Actor-Movie dataset