Data Placement and Replica Selection for Improving Co-location in Distributed Environments

Ashwin Kumar Kayyoor Amol Deshpande Samir Khuller
{ashwin@cs.umd.edu, amol@cs.umd.edu, samir@cs.umd.edu}
University of Maryland at College Park

ABSTRACT

Increasing need for large-scale data analytics in a number of application domains has led to a dramatic rise in the number of distributed data management systems, both parallel relational databases, and systems that support alternative frameworks like MapReduce. There is thus an increasing contention on scarce data center resources like network bandwidth (especially cross-rack bandwidth); the energy requirements for powering the computing equipment are also growing dramatically. In this work, we exploit the fact that most distributed environments need to use replication for fault tolerance, and we devise workload-aware replica selection and placement algorithms that attempt to minimize the total resources consumed in a distributed environment. More specifically, we address the problem of minimizing *average query span*, i.e., the average number of machines that are involved in processing of a query through co-location of related data items, for a given query workload; as we illustrate, under reasonable assumptions, this directly reduces the total amount of resources consumed by the query and thus the total energy consumed during the query execution. We model the query workload as a hypergraph over a set of data items (which could be relation partitions, or file chunks), and formulate and analyze the problem of replica placement by drawing connections to several well-studied graph theoretic concepts. We use these connections to develop a series of algorithms to decide which data items to replicate, and where to place the replicas. We evaluate our proposed techniques by building a trace-driven simulation framework and by conducting an extensive performance evaluation. Our experiments show that careful data placement and replication can dramatically reduce the average query spans.

1. INTRODUCTION

Massive amounts of data are being generated every day in a variety of domains ranging from scientific applications to social networks to retail. The stores of data on which modern businesses rely are already vast and increasing at an unprecedented pace. Organizations are capturing data at deeper levels of detail and keeping more history than they ever have before. Managing all of the data is thus emerging as one of the key challenges of the new decade. This deluge of data has led to an increased use of parallel and distributed data management systems like parallel databases or MapReduce frameworks like Hadoop to analyze and gain insights from the data. Complex analysis queries are run on these data management systems in order to identify interesting trends, make unusual patterns stand out, or verify hypotheses. In parallel databases, the queries typically consist of multiple joins, group definitions on multiple attributes, and complex aggregations. On Hadoop, the tasks have similar flavor with simplest of map-reduce programs being aggregation tasks that form the basis of analysis queries. There have also been many attempts to combine the scalability of Hadoop and declarative querying abilities of relational databases [39, 31].

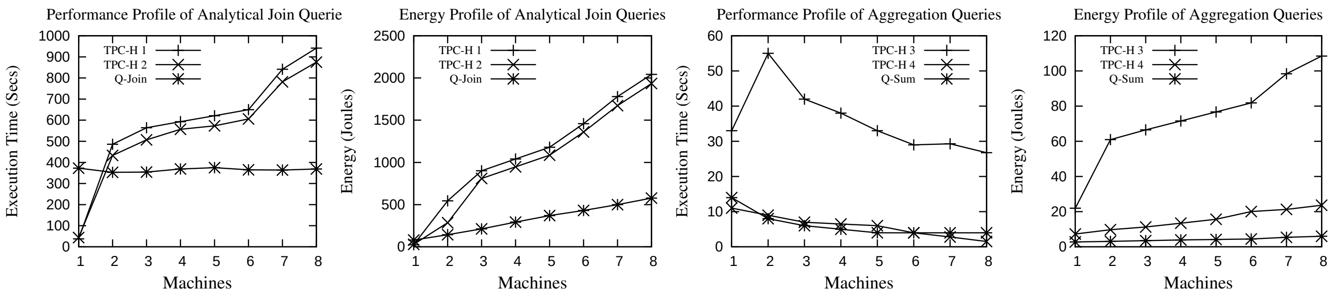
For fault tolerance, load balancing and availability, these systems usually keep several copies of each data item (e.g., Hadoop file system (HDFS) maintains at least 3 copies of each data item by default [42]). Our goal in this work is to show how to exploit this inherent replication in these systems to minimize the number of machines that are involved in executing a query, called the *query span* (we use th term query to denote both SQL queries and Hadoop tasks). There are several motivating reasons for doing this:

Minimize the communication overhead: Query span directly impacts the total communication that must be performed to execute a query. This is clearly a concern in distributed setups (e.g., grid systems [38] or multi-datacenter deployments); however even within a data center, communication network is oversubscribed, and especially cross-rack communication bandwidth can be a bottleneck [23, 10]. HDFS, for instance, tries to place all replicas of a data item in a single rack to minimize inter-rack data transfers [42]. Our algorithms can be used to further guide these decisions and cluster replicas of multiple data items on to a single rack to improve network performance for queries that access multiple data items, which HDFS currently ignores. In cloud computing, the total communication directly impacts the total dollar cost of executing the query.

Minimize the total amount of resources consumed: It is well-known that parallelism comes with significant startup and coordination overheads, and we typically see sub-linear speedups as a result of these overheads and data skew [32]. Although the response time of a query usually decreases in a parallel setting, the total amount of resources consumed typically increases with increased parallelism.

Reduce the energy footprint: Computing equipment in US costs data center operators millions of dollars annually for energy, and also impacts the environment. Energy costs are ever increasing and hardware costs are decreasing – as a result soon the energy costs to operate and cool a data center may exceed the cost of the hardware itself. Minimizing the total amount of resources consumed directly reduces the total energy consumption of the task.

To support these claims and to motivate query span as a key metric



(a) (b) (c) (d)

Figure 1: Results illustrating the overheads in parallel query processing

to optimize, we conducted a set of experiments analyzing the effect of query span on the total amount of resources consumed, and the total energy consumed, under a variety of settings. First setting is a horizontally partitioned MySQL cluster, where we evaluate a total four TPC-H template queries. Two of the queries are complex analytical join queries (TPC-H1, TPC-H2), whereas the other two are simple aggregation queries on a single table. In the second setting, we implemented our own distributed query processor on the top of multiple MySQL instances running on a cluster where predicate evaluations are pushed on to the individual nodes and data is shipped to a single node for perform the final steps. On this setup we evaluate two queries: a complex join query (Q-Join) and a simple aggregate query on a single table (Q-Sum). In Figures 1(a) and 1(b), we plot the execution times and the energy consumed as the number of machines across which the tables are partitioned (and hence query span) increases. As we can see, the execution times of the TPC-H queries run on MySQL cluster actually increased with parallelism, which may be because of nested loop join implementation in MySQL cluster (a known problem that is being fixed). Our implementation shows that execution time remains constant, but in all cases, energy costs increase with query span. In the second experiment with simpler queries (Figures 1(c) and 1(d)), though execution times decrease as the query span increases, energy consumption increases in all cases. The energy consumed is computed using the Itanium server power model calculated by using Mantis full-system power modelling technique [16]. We use the *dstat* tool to collect various system performance counters such as CPU utilization, network read and writes, I/O, and memory footprint, which along with the power model is used to compute the total energy consumed. From our experiments it is evident that, as the number of machines involved in processing a query increases, total resources consumed to process the query also rise.

In this paper, we address the problem of minimizing the average query span for a query workload through judicious replica selection (by choosing which data items to replicate and how many times), and data placement. A recent system, CoHadoop [17], also aims at co-locating related data items to improve performance of Hadoop; the algorithms that we develop here can be used to further guide the data placement decisions in their system. Our techniques work on an abstract representation of the query workload, and are applicable to both multi-site data warehouses and general purpose data centers. We assume that a query workload trace is provided that lists the data items that need to be accessed to answer each query. The data items could be database relations, parts of database relations (e.g., tuples or columns), or arbitrary files. We represent such a workload as a *hypergraph*, where the nodes are the data items and each query is translated into a *hyperedge* over the nodes. The goal is to store each data item (node in the graph) onto a subset of machines/sites (also called *partitions*), obeying the storage capacity requirements for the partitions. Note that the partitions do not have to be machines, but could instead represent racks or even datacenters. This specifies the layout completely. The cost for each query is defined to be smallest number of partitions that contain all the data the query needs. Our goal is to find a layout that minimizes the average cost over all queries. Our algorithms can optimize for load or storage constraints, or both.

Our key contributions include formulating and analyzing this problem, drawing connections to several problems studied in the graph algorithms literature, and developing efficient algorithms for data placement. In addition, we examine the special case when each query accesses at most two data items – in this case the hypergraph is simply a graph. For this case, we are able to develop theoretical bounds for special classes of graphs that gives an understanding of the trade-off between energy cost and storage.

We can use similar techniques to partition large graphs across a distributed cluster; smart replication of some of the (boundary) nodes can result in significant savings in the communication cost to answer queries (e.g., to answer subgraph pattern queries). More recently, Curino et al. [13] also proposed a workload-aware approach for database partitioning and replication to minimize the number of sites involved in distributed transactions; our algorithms can be applied to that problem as well. However, we note that replication costs become critical in that case. We plan to look into modifying our algorithms to take into account the replication costs in future work. Our techniques are also applicable in partition farms such as MAID [11], PDC [33], or Rabbit [6], that utilize a subset of a partition array as a workhorse to store popular data so that other partitions could be turned off or sent to lower energy modes.

Significant work has been done on the converse problem of minimizing query response times or latencies. *Declustering* refers to the approach of leveraging parallelism in the partition subsystem by spreading out blocks across different partitions so that multiblock requests can be executed in parallel. In contrast, we try to cluster data items together to minimize the number of sites required to satisfy a complex analytical query.

Minimizing average query spans through replication and data placement raises two concerns. First, does it adversely affect load balancing? Focusing simply on minimizing query spans can lead to a load imbalance across the partitions. However, we don't believe this to be a major concern, and we believe total resource consumption should be the key optimization goal. Most analytical workloads are typically not latency-sensitive, and we can use temporal scheduling (by postponing certain queries) to balance loads across machines. We can also easily modify our algorithms to incorporate load constraints. A second concern is the cost of replica maintenance. However, most distributed systems do replication for fault tolerance, and hence we do not add any extra overhead. Secondly, most systems focused on large-scale analytics do batch inserts, and the overall cost of inserts is relatively low.

We have also built a trace-driven simulation framework that enables us to systematically compare different algorithms, by automatically generating varying types of query workloads and by calculating the total energy cost of a query trace. We conducted an extensive experimental evaluation using our framework, and our results show that our techniques can result in high reduction in query span compared to baseline or random data placement approaches that can help minimize distributed overheads.

Outline: We begin with a discussion of closely related work (Section 2). We formally define the problem that we address in the paper and analyze it (Section 3). We present a series of algorithms to solve the problem (Section 4), and present an extensive performance evaluation using a trace-driven simulation framework that we have built (Section 5).

2. RELATED WORK

Data partitioning and replication plays an increasingly important role in large scale distributed networks such as content delivery networks (CDN), distributed databases and distributed systems such as peer-to-peer networks. Recent work [46, 14, 3] has shown that judicious placement of data and replication improves the efficiency of query processing algorithms. There has been some recent interest on improving data colocation in large scale processing systems like Hadoop. Recent work by Eltabakh et al. [17] on CoHadoop is very close to our work, where they provide an extension for Hadoop with a lightweight mechanism that allows applications to control where data is stored. They focus on data colocation to improve the efficiency of many operations, including indexing, grouping, aggregation, columnar storage, joins, and sessionization. Our techniques are complimentary to their work. Hadoop++ [14] is another closely related work, where it exploits data pre-partitioning and colocation. There is substantial amount of work on replica placement that focuses on minimization of network latency and bandwidth. Neves et al. [29] propose a technique for replication in CDN where they replicate data on to subset of servers to handle requests so that the traffic cost in the network is minimized. There has been a lot of work on dynamic/adaptive replica management [43, 34, 35, 22, 36, 45], where replicas are dynamically placed, moved or deleted based on the read/write access frequencies of the data items again with the goal of minimizing bandwidth and access latency. Our work is complimentary to this line of work, in a way that we replicate the data considering the query workload nature by modelling it as hypergraph. Extending our approach to track changes in the query workload and adapt the replication decisions is an interesting direction for future work that we are planning to pursue.

Graphs have been used as a tool to model various distributed storage problems and to come up with replication strategies to achieve a specific objective. Du et al. [15] study Quality-of-Service (QoS)-aware replica placement problem in a general graph model. In their model, vertices are the servers with various weights representing node characteristics and edges representing the communication costs. Other work has modeled network topology as a graph and developed replication strategies or approximations (replica placement in general graphs is NP-complete) [44]. This is different from what we are doing in this paper: we model query workload as a hypergraph whereas these works model network topology as graph. On the other hand, we assume a uniform network topology in that the communication cost between any pair of nodes is identical; we believe this better approximates the current networks. Curino et al. [12] model an OLTP query workload as a graph, and also use graph partitioning techniques for placement of the tuples. They however do not develop new partitioning algorithms; our techniques can be used to design better data placement algorithms for their setting as well.

Our work is different from several other works on data placement [26, 27, 30] where the database query workload is also modeled as a hypergraph and partitioning techniques are used to drive data placement decisions. Liu et al. [27] propose a novel declustering technique based on max-cut partitioning of a weighted similarity graph. Aykanat et al. [26] observe that the approach where each query over a set of relations is represented by a clique over those relations, does not accurately capture the cost function, and instead propose directly using the hypergraph representation of the query workload. Tosun et al. [40, 41] and Ferhatosmanoglu et al. [19] propose using replication along with declustering for achieving optimal parallel I/O for spatial range queries. The goal with all of that prior work is typically minimization of latencies and query response times by spreading out the work over a large number of partitions or devices. Under the framework that we consider here, this is exactly the wrong optimization goal – we would like to cluster data required for each query on as few partitions as possible.

The problems we study are closely related to several well-studied problems in graph theory and can be considered generalizations of those problems. A basic special case of our main problem is the *minimum graph bisection* problem (which is NP-Hard), where the goal is to partition the input graph into two equal sized partitions, while minimizing the number of edges that are cut [8]. There is much work on both that problem and its generalization to hypergraphs and to k-way partitioning [28, 24, 25]. The work on *community detection* over complex networks [20] has also proposed many schemes for partitioning graphs to minimize the connections between partitions; however the resulting partitions there do not have to be balanced – a critical requirement for us. Another closely related problem is that of finding *dense subgraphs* in a graph, where the goal is to find a group of vertices where the number of edges in the induced subgraph is maximized [18]. Finally, there is much work on finding *small* separators in graphs. Several theoretical results on known about this problem. We discuss these connections in more detail later when we describe our proposed algorithms.

3. PROBLEM DEFINITION; ANALYSIS

Next, we formally define the problem that we study, and draw connections to some closely related prior work on graph algorithms. We also analyze a special case of the problem formally, and show an interesting theoretical result.

Problem Definition: Given a set of data items $\mathcal{D}$ and a set of partitions, our goal is to decide which data items to replicate and how to place them on the partitions to minimize the average span of an expected query workload; *span* of a query is defined to be the minimum number of partitions that must be accessed to answer the query. To make the problem more concrete, we assume that we are given a set of queries over the data items, and our goal is to minimize the average span over these queries. For simplicity, we assume that we are given a total of N identical partitions each with capacity C units, and further that the data items are all unit-sized (we will relax this assumption later). Clearly, the number of data items must be smaller than $N \times C$ (so that each data item can be placed on at least one partition). Further, let N_e denote the minimum number of partitions needed to place the data items (i.e., $N_e = \lceil |\mathcal{D}|/C \rceil$).

The query workload can be represented as a hypergraph, $\mathcal{H} = (V, E)$, where the nodes are the data items and each (hyper)edge

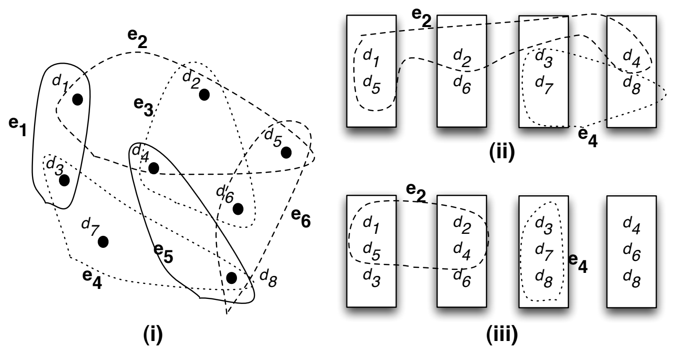


Figure 2: (i) Modeling a query workload as a hypergraph – d_i denotes the data items, and e_i denotes the queries represented as hyperedges; (ii) A layout w/o replication onto 4 partitions – the span of two of the hyperedges is also shown; (iii) A layout with replication – span for both queries reduces by 1.

$e \in E$ corresponds to a query in the workload. Figure 2 shows an illustrative example, where we have 6 queries over 8 data items, each of which is represented as a hyperedge over the data items. The figure also shows two layouts of the data items onto 4 partitions of capacity 3 each, without replication and with replication.

Calculating Span: When there is no replication, calculating the span of a query is straightforward since each data item is associated with a single partition. However, if there is replication, the problem becomes NP-Hard. It is essentially identical to the *minimum set cover* problem [21], where we are given a collection of subsets of a set (in our case, the partitions) and a query subset, and we are asked to find the minimum number of subsets (partitions) required to cover the query subset.

As an example, for query e_2 in Figure 2, the span in the first layout is 3. However, in the second layout, we have to choose which of the two copies of d_4 to use for the query. Using the first copy (on second partition) leads to the lowest span of 2. Overall, the average query span for the first layout is $\frac{13}{6}$, but use of replication in the second layout reduces this to $\frac{8}{6}$.

We use a standard greedy algorithm for choosing replicas to use for a query and for calculating the span. For each of the partitions, we compute the size of its intersection with the query subset. We choose the partition with the highest intersection size, remove all items from the query subset that are contained in the partition, and iterate until there are no items left in the query subset. This simple greedy algorithm provides the best known approximation to the set cover problem ($\log |Q|$, where $|Q|$ is the query size).

Hypergraph Partitioning: Without replication, the problem we defined above is essentially the k-way (balanced) hypergraph partitioning problem that has been very well-studied in the literature. However, the optimization goal of minimizing the average span is unique to this setting; prior work has typically studied how to minimize the number of *cut* hyperedges instead. Several packages are available for partitioning very large hypergraphs efficiently [1, 2]. The proposed algorithms are typically heuristics or combinations of heuristics, and most often the source code is not available. We use one such package (hMETIS) as the basis of our algorithms.

Finding Dense Subgraphs of a specified size: Given a set of nodes S in a graph, the *density* of the subgraph induced by S is defined to be the ratio of the number of edges in the induced subgraph and $|S|$. The dense subgraph problem is to find the densest subgraph of a given size. To understand the connection to the dense subgraph problem, consider a scenario where we have exactly one "extra" partition for replicating the data items (i.e., $N_e = N - 1$). Further, assume that each query refers to exactly two data items, i.e., the hypergraph $\mathcal{H}$ is just a graph. One approach would then be to first partition the data items into $N - 1$ partitions without replication, and then try to use this extra partition optimally. To do this, we can construct a *residual* graph, which contains all edges that were cut in this partitioning. The span of each of the queries corresponding to these edges is exactly 2. Now, we find the subgraph of size C such that the number of induced edges (among the nodes of the subgraph) is maximized, and we place these data items on the extra partition. The span of the queries corresponding to these edges are all reduced from 2 to 1, and hence this is an optimal way to utilize the extra partition. We can generalize this intuition to hypergraphs and this forms the basis of one of our algorithms.

Unfortunately, the problem of finding the most dense subgraph of a specified size is NP-Hard (with no good worst case approximation guarantees), so we have to resort to heuristics. One such heuristic that we adapt in our work is as follows: recursively remove the lowest degree node from the residual graph (and all its incident edges) till the size of the residual graph is exactly C. This heuristic has been analysed by Asahiro et al. [7] who find that this simple greedy algorithm can solve this problem with approximation ratio of approximately $2(\frac{|V|}{C} - 1)$ (when $C \leq |V|/3$).

Sublinear Separators in Graphs: Consider the special case where $\mathcal{H}$ is a graph, and further assume that there are only 2 partitions (i.e., $N = 2$). Further, lets say that the graph has a small *separator*, i.e., a set of nodes whose deletion results in two connected components of size at most $n/2$. In that case, we can replicate the separator nodes (assuming there is enough redundancy) and thus guarantee that each query has span exactly 1. The key here is the existence of small separators of bounded sizes. Such separators are known to exist for many classes of graphs, e.g., for any family of graphs that excludes a minor [4].

A separator theorem is usually of the form that, any n-vertex graph can be partitioned into two sets A, B, such that $|A \cap B| = c\sqrt{n}$ for some constant c, $|A - B| < 2n/3$, $|B - A| < 2n/3$, and there are no edges from a node in $A - B$ to a node in $B - A$. This directly suggests an algorithm that recursively applies the separator theorem to find a partitioning of the graph into as many pieces as required, replicating the separator nodes to minimize the average span. Such an algorithm is unlikely to be feasible in practice, but may be used to obtain theoretical bounds or approximation algorithms. For example, we prove that:

THEOREM 1. *Let G be a graph with n nodes that excludes a minor of constant size. Further, let N_e denote the number of partitions minimally required to hold the nodes of G (i.e., $N_e = \lceil n/C \rceil$). Then, asymptotically, $N_e^{1.73}$ partitions are enough to partition the nodes of G with replication so that each edge is contained completely in at least one partition.*

Proof: The proof relies on the following theorem by Alon et al. [4]:

THEOREM 2. *Let G be a graph with n nodes that excludes a fixed minor with h nodes. Then we can always find a separation (A, B) such that $|A \cap B| \leq h^{\frac{3}{2}} n^{\frac{1}{2}}$, $|A - B|, |B - A| \leq \frac{2}{3}n$.*

Consider a recursive partitioning of G using this theorem. We first find a separation of G into A and B. Since A and B are subgraphs of G, they also exclude the same minor. Hence we can further partition A and B into two (overlapping) partitions each. Now, both $|A|$ and $|B|$ are $\leq \frac{2}{3}n + h^{\frac{3}{2}} n^{\frac{1}{2}}$. For large n, the second term is dominated by ϵn, for any $\epsilon > 0$. We choose some such $\epsilon = 1/300$. Then, we can write: $|A|, |B| \leq (\frac{2}{3} + \epsilon)n = 0.67n$ for large enough n.

Now we continue recursively for l steps getting us 2^l subgraphs of the original graph G, such that each of the subgraphs fits in one partition. Note that, by construction, every edge is contained in at least one of these subgraphs; thus 2^l partitions are sufficient for data placement as required. Since the partition capacities are $O(n)$, we can use the above formula to compute l. We need: $0.67^l n < C = n/N_e$. Solving for l, we get: $l > log_2(N_e^{1.73})$. Hence, the number of partitions needed to partition G with replication so that each edge is contained in at least one partition is less than $N_e^{1.73}$.

Although the bound looks strong, note that the above class of graphs can have at most $O(n)$ edges (i.e., these types of graphs are typically sparse). Proving similar bounds for dense graphs would be much harder and is an interesting future direction.

For general graphs, in Appendix A, we show that:

THEOREM 3. *If the optimal solution uses βN_e partitions to place the data items so that each edge is contained in at least one partition, then either we can get an approximation with factor $\frac{2}{2-\alpha}$ for $0 \leq \alpha \leq 1$ using N_e partitions, or a placement using $\frac{CN_e\beta}{2\alpha}$ partitions with span 1 for each edge.*

4. DATA PLACEMENT ALGORITHMS

In this section, we present several algorithms for data placement with replication, with the goal to minimize the average query span. Instead of starting from scratch, we chose to base our algorithms on existing hypergraph partitioning packages. As we discussed in the previous sections, the problem of balanced and unbalanced hypergraph partitioning has received a tremendous amount of attention in various communities, especially the VLSI community. Several very good packages are freely available for solving large partitioning problems [1, 24, 2, 9]. We use a hypergraph partitioning algorithm (called HPA) as a blackbox in our algorithms, and focus on replicating data items appropriately to reduce the average query span. An HPA algorithm typically tries to find a balanced partitioning (i.e., all partitions are of approximately equal size) that minimizes some optimization goal. Usually, allowing for unbalanced partitions results in better partitioning. In the algorithm descriptions below, we assume that the HPA algorithm can return an exactly balanced partition, where all partitions are of equal size, if needed.

Following the discussion in the previous section, we develop four classes of algorithms:

- **Iterative HPA (IHPA)**: Here we repeatedly use HPA until all the extra space is utilized.
- **Dense Subgraph-based (DS)**: Here we use a dense subgraph finding algorithm to utilize the redundancy.
- **Pre-replication (PR)**: Here we attempt to identify a set of nodes to replicate a priori, modify the input graph by replicating those nodes, and then run HPA to get a final placement.
- **Local Move-based (LM)**: Starting with a partition returned by HPA, we improve it by replicating a small group of data items at a time.

As expected the space of different variants of the above algorithms is very large. We experimented with many such variants in our work. We begin with a brief listing of some of the key subroutines that we use in the pseudocodes. We then describe a representative set of algorithms that we use in our performance evaluation.

4.1 Preliminaries; Subroutines

The inputs to the data placement algorithm are: (1) the hypergraph, $\mathcal{H}(V, E)$, with vertex set V and (hyper)edge set E that captures the query workload, and (2) the number of partitions, N and (3) the capacity of each partition C. We use N_e to denote the minimum number of partitions needed to partition the hypergraph ($N_e \leq N$).

Our algorithms use a hypergraph partitioning algorithm (HPA) as a blackbox. HPA takes as input the hypergraph to be partitioned, the number of partitions, and an *unbalance factor* (UBfactor). The unbalance factor is set so that HPA has the maximum freedom, but the number of nodes placed in any partition does not exceed C. For instance, if $|V| = N_e \times C$ and if HPA is asked to partition into N_e partitions, then the unbalance factor is set to be the minimum. However, if HPA is called with $N' > N_e$ partitions, then we appropriately set the unbalance factor to the maximum possible. The formula we use in our experiments to set unbalance factor is:

$$UBfactor = 100 * \frac{partitionCapacity * noPartitions - totalItems}{totalItems * noPartitions}$$

We modify the output of HPA slightly to ensure that the partition capacity constraints are not violated. This is done as follows: if there is a partition that has higher than maximum number of nodes, we move a small group of nodes to another partition with fewer than maximum number of nodes. We use one of our algorithms developed below (LMBR) for this purpose.

In the pseudocodes shown, apart from HPA, we also assume existence of the following subroutines:

- **getSpanningPartitions($\mathcal{G}, e$):** Let the current placement (during the course of the algorithm) be $\mathcal{G} = \{G_1, \cdots, G_N\}$ where $G_1, \cdots, G_n$ denote the subgraphs of $\mathcal{G}$ assigned to the different partitions and may not be disjoint (i.e., same node may be contained in two or more partitions because of replication). Given a hyperedge e, this procedure finds a minimal subset of the partitions $MD_e \subseteq \mathcal{G}$, such that every node in e is contained in at least one partition in MD_e. We use the greedy Set Cover algorithm for this purpose. We start with the partition G_i that has the maximum overlap with e, and include it in MD_e. We then remove all the nodes in e that are contained in G_i (i.e., "covered" by G_i) and repeat till all nodes are covered.
- **getQuerySpan($\mathcal{G}, e$):** Given a current placement $\{G_1, \cdots, G_N\}$ and a hyperedge e, this procedure finds the span of the hyperedge e. We use the same algorithm as above, but return $|MD_e|$ instead of MD_e.
- **getAccessedItems($\mathcal{G}, e, g \in \mathcal{G}$):** Given a current placement $\mathcal{G} = \{G_1, \cdots, G_N\}$, a hyperedge e and a partition $g \in \mathcal{G}$, this returns the set of items that the query corresponding to e would access from partition g, as computed by the greedy Set Cover algorithm. This may be empty even if $e \cap g \neq \phi$.
- **pruneHypergraphBySpan($\mathcal{G}, \mathcal{H}, minSpan$):** Given a current placement $\mathcal{G}$ and a value of $minSpan$, this routine removes all hyperedges from $\mathcal{H}$ with span less than or equal to $minSpan$.
- **getKDensestNodes($\mathcal{H}, K$):** Given a hypergraph $\mathcal{H}$, this procedure returns a dense subgraph containing at nodes having total weight of atmost K. We use a greedy algorithm for this purpose: we find the lowest degree node and remove that node and all edges incident on it; if the graph still has nodes having total weight more than K, we repeat the process by finding the lowest degree node in the new graph.
- **pruneHypergraphToSize($\mathcal{H}, K$):** Given a current placement $\mathcal{G}$ and a value of K, this routine uses the same algorithm as

for getKDensestNodes to find a (dense) hypergraph over nodes having total weight of K.

- **totalWeight(V, W_v):** Given a set of vertices V and weight vector of vertices $W_v, v \in V$, this routine returns the total weight of vertices.

We note that, because of the modularized way our framework is designed, we can easily use different, more efficient algorithms for solving these subproblems.

4.2 Iterative HPA (IHPA)

Here, we start by using HPA to get a partitioning of the data items into exactly N_e partitions (recall that N_e is the minimum number of partitions needed to store the data items). We then prune the original hypergraph $\mathcal{H}(V, E)$ to get a residual hypergraph $\mathcal{H}^{'}(V^{'}, E^{'})$ as follows: we remove all hyperedges that are completely contained in a single partition (i.e., hyperedges with span 1), and we then remove all the data items that are not contained in any hyperedge. If the number of nodes in the $\mathcal{H}'$ is less than $(N - N_e)C$ (i.e., if the data items fit in the remaining empty partitions), we apply HPA to obtain a balanced partitioning of $\mathcal{H}'$ and place the partitions on the remaining partitions. This process is repeated if there are still empty partitions.

If the number of nodes in $\mathcal{H}'$ is larger than the remaining capacity, we prune the graph further by removing the hyperedges with the lowest span one at a time (these hyperedges are likely to see the least improvement by replication) and the data items that now have 0 degree, until the number of nodes in $\mathcal{H}'$ becomes sufficiently low; then we apply HPA to obtain a balanced partitioning of $\mathcal{H}'$ and place the partitions on the remaining partitions. If there are still empty partitions, we repeat the process by reconstructing a new residual graph. Algorithm 1 depicts the pseudocode for this technique.

Algorithm 1 Iterative HPA (IHPA)

```
Require: $\mathcal{H}(V, E), N, C$
 1: Run HPA to get an initial partitioning into $N_e$ partitions: $\mathcal{G} = \{G_1, G_2, \ldots, G_{N_e}\}$;
 2: $edgeCost$ = avgDataItemsPerQuery($\mathcal{H}$);
 3: while $edgeCost \neq 0$ and $|\mathcal{G}| \neq N$ do
 4:     $\mathcal{H}^{'}(V', E')$ = pruneHypergraphBySpan($\mathcal{G}, \mathcal{H}, edgeCost$);
 5:     $N_{cur} = \frac{totalWeight(V^{'}, W_{v'})}{C}$;
 6:     if $|\mathcal{G}| + N_{cur} \leq N$ and $|\mathcal{H}^{'}| \neq 0$ then
 7:         $\mathcal{G} = \mathcal{G} \cup$ HPA($\mathcal{H}^{'}, N_{cur}$);
 8:     else if $|\mathcal{G}| + N_{cur} > N$ then
 9:         $\mathcal{G} = \mathcal{G} \cup$ HPA($\mathcal{H}^{'}, N - |\mathcal{G}|$);
10:     else
11:         decrement $edgeCost$ by 1;
12:     end if
13: end while
14: return final partitions $G_1, G_2, \cdots, G_N$
```

4.3 Dense Subgraph-based (DS)

This algorithm directly follows from the discussion in the previous section. As above, we use HPA to get an initial partitioning. We then fill the remaining $N - N_e$ partitions one at a time, by identifying a dense subgraph of the residual hypergraph. This is done by removing the lowest degree nodes from $\mathcal{H}'$ until the number of nodes in it reaches C (the partition capacity). These data items are then placed on one of the remaining partitions, and the procedure is repeated until all partitions are utilized. Pseudocode is shown in Algorithm 2.

Algorithm 2 Dense Subgraph-based (DS)

```
Require: $\mathcal{H}(V, E), N, C$
 1: Run HPA to get an initial partitioning into $N_e$ partitions: $\mathcal{G} = \{G_1, G_2, \ldots, G_{N_e}\}$;
 2: $\mathcal{H}^{'} = \mathcal{H}$;
 3: while $|\mathcal{G}| \neq N$ do
 4:     $\mathcal{H}^{'}$ = pruneHypergraphBySpan($\mathcal{G}, \mathcal{H}, 1$);
 5:     if $|\mathcal{H}'| = 0$ then
 6:         break;
 7:     end if
 8:     $denseNodes$ = getKDensestNodes($\mathcal{H}^{'}, C$);
 9:     Add a partition containing $denseNodes$ to $\mathcal{G}$;
10: end while
11: return final partitions $G_1, G_2, \cdots, G_N$
```

4.4 Pre-Replication-based Algorithm (PRA)

This algorithm is based on the idea of identifying small separators and replicating them. However, we do not directly adapt the recursive algorithm described in Section 3 for two reasons. First, since we have a fixed space budget for replication, we must somehow distribute this budget to the various stages and it is unclear how to do that effectively. More importantly, the basic algorithm of bisecting a graph and then recursing is not considered a good approach for achieving good partitioning [37, 25].

We instead propose the following algorithm. We start with a partitioning returned by HPA, and identify "important" nodes such that by replicating these nodes, the average query span would be reduced the most. Then, we create a new hypergraph by replicating these nodes (until we have enough nodes to fill all the partitions), and run HPA once again to attain a final partitioning. However, neither of these steps is straightforward.

Identifying Important Nodes: The goal is to decide which nodes will offer the most benefit if replicated. We start with a partitioning obtained using HPA, and then analyze the partitions to decide this. We describe the intuition first. Consider a node a that belongs to some partition G_i. Now count the number of those hyperedges that contain a but do not contain any other node in G_i; we denote this number by $score_a$. If this number is high, then the node is a good candidate for replication since replicating the node is likely to reduce the query spans for several queries. We use the partitioning returned by HPA to rank all the nodes in the decreasing order by this count, and then process the nodes one at a time.

Replicating Important Nodes: Let d be the node with the highest value of $score_d$ among all nodes. We now have to decide how many copies of d to create, and more importantly, which copies to assign to which hyperedge. Figure 3(ii) illustrates the problems with an arbitrary assignment. Here we replicate the node d to get one more copy d', and then we assign these two copies to the hyperedges e_1, e_2, e_3, e_4 as shown (i.e., we modify some of the hyperedges to remove d and add d' instead). However, the assignment shown is not a good one for a somewhat subtle reason. Since e_1 and e_3 (which are assigned the original d) do not share any other nodes, it is likely that they will span different sets of partitions, and one of them is likely to still pay a penalty for node d. On the other hand, the assignment shown in Figure 3(iii) is better because here the copies are assigned in a way that would reduce the average query span.

We formalize this intuition in the following algorithm. For node d, let $E_d = \{e_{d_1}, e_{d_2}, \cdots, e_{d_k}\}$ denote the set of hyperedges that contain d. For hyperedge e_{d_i}, let $\mathcal{G}_{d_i}$ denote the set of partitions that e_{d_i} spans. We then identify a set of partitions, S, such that each of $\mathcal{G}_{d_i}$ contains at least one partition from this set (i.e., $S \cap \mathcal{G}_{d_i} \neq$

Algorithm 3 Pre-replication-based Algorithm (PRA)

```
Require: $\mathcal{H}(V,E), N, C$
 1: Run HPA to get an initial partitioning into $N_e$ partitions: $\mathcal{G} = \{G_1, G_2, \ldots, G_{N_e}\}$;
 2: for $v \in V$ do
 3:     let $v$ be contained in partition $G_v$;
 4:     compute $score_v = |\{e \in E \mid e \cap G_v = \{v\}\}|$;
 5: end for
 6: $H^r = H$;
 7: for $v \in V$ in decreasing order by $score_v$ do
 8:     $E_v = \{e \in E \mid v \in e\}$;
 9:     $G_v = \{$getSpanningPartitions$(\mathcal{G}, e) \mid e \in E_v\}$;
10:     $S =$ getHittingSet$(G_v)$;
11:     for $g \in S$ do
12:         $copy_g =$ makeNewCopy$(v)$;
13:         for $e \in E_v$ s.t. $g \in$getSpanningPartitions$(\mathcal{G}, e)$ do
14:             $e = e - \{v\} + \{copy_g\}$;
15:         end for
16:     end for
17: end for
18: $\mathcal{G} =$ HPA$(\mathcal{H}^r, N)$;
19: return final partitions $G_1, \cdots, G_N$
```

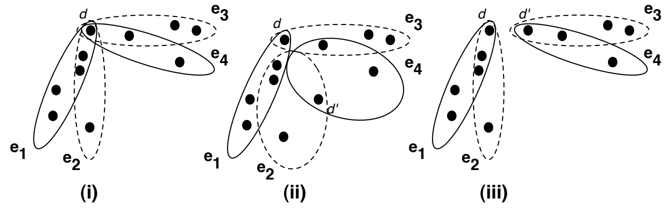


Figure 3: When replicating a node, distribution of the copies to the hyperedges must be done carefully. Distribute the replica copies such that it results in entanglement of the incident hyperedges.

ϕ). Such a set is called a "hitting set". We then replicate d to make a total of $|S|$ copies. Finally, we assign the copies to the hyperedges according to the hitting set, i.e., we uniquely associate the copies of d with the members of S, and for a hyperedge e_{d_i}, we assign it a copy such that the associated element from S is contained in $\mathcal{G}_{d_i}$ (if there are multiple such elements, we choose one arbitrarily).

The problem of finding the smallest hitting set is NP-Hard. We use a simple greedy heuristic. We find the partition that is common to the maximum number of sets $\mathcal{G}_{d_i}$, include it in the hitting set, remove all sets that contain it, and repeat. Algorithm 3 depicts the pseudocode for this technique.

4.5 Local Move Based Replication (LMBR)

Finally, we consider algorithms based on local greedy decisions about what to replicate, starting with a partitioning returned by HPA. For simplicity and efficiency, we chose to employ moves involving two partitions. More specifically, at each step, we copy a small group of data items from one partition to another. The decisions are made greedily by finding the move that results in the highest decrease in the average query span ("benefit") per data item copied ("cost"). For this purpose, at all times, we maintain a priority queue containing the best moves from $partition_i$ to $partition_j$, for all $i \neq j$. For two partitions $partition_i, partition_j$, the best group of data items to be copied from $partition_i$ to $partition_j$ is calculated as follows. Let $E_{ij} = \{e_{ij_1}, \cdots, e_{ij_l}\}$ denote the hyperedges that contain data items from both the partitions. We construct a hypergraph $H_{i \to j}$ on the data items of $partition_i$ as follows: for every edge e_{ij_k}, we add a hyperedge to $H_{i \to j}$ on the data items common to e_{ij_k} and $partition_i$. Figure 4 illustrates this process with an example.

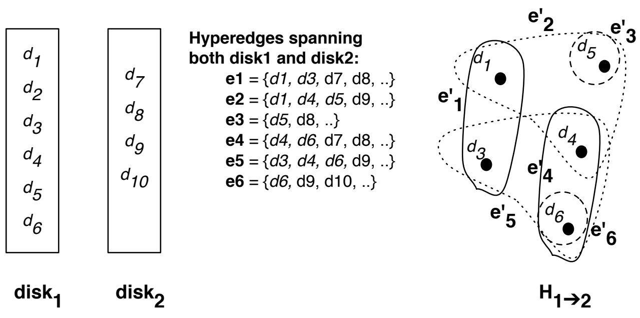

d1
d2
d3
d4
d5
d6
d7
d8
d9
d10
disk1
disk2
Hyperedges spanning both disk1 and disk2:
e1 = {d1, d3, d7, d8, ..}
e2 = {d1, d4, d5, d9, ..}
e3 = {d5, d8, ..}
e4 = {d4, d6, d7, d8, ..}
e5 = {d3, d4, d6, d9, ..}
e6 = {d6, d9, d10, ..}
e'2
e'3
e'1
d1
d5
d4
d3
e'4
e'5
d6
e'6
H1→2

Figure 4: Constructing $H_{1\to 2}$: e.g., corresponding to hyperedge e_1 that spans both partitions, we have a hyperedge e'_1 over d_1 and d_3.

Now, if we were to copy a group of data items X from $partition_i$ to $partition_j$, the resulting decrease in total span (across all edges) is exactly the number of hyperedges in $H_{i \to j}$ that are completely contained in X. Thus, the problem of finding the best move from $partition_i$ to $partition_j$ is similar to the problem of finding a dense subgraph, with the main difference being that, we want to minimize the cost/benefit ratio and not maximize the benefit alone. Hence, we modify the algorithm for finding dense subgraph as follows. We first compute the cost/benefit ratio for the entire group of nodes in $H_{i \to j}$. The cost is set to ∞ if the number of nodes to be copied is more than the empty space in $partition_j$. We then remove the lowest degree node from $H_{i \to j}$ (and any incident hyperedges), and again compute the cost/benefit ratio. We pick the group of items that results in the lowest cost/benefit ratio.

After finding the best moves for every pair of partitions, we choose the overall best move, and copy the data items accordingly. We then recompute the best moves for those pairs which were affected by this move (i.e., the pairs containing the destination partition), and recurse until all the partitions are full.

Improved LMBR: Although the above looks like a reasonable algorithm, it did not perform very well in our first set of experiments. As described above, the algorithm has a serious flaw. Going back to the example in Figure 4, say we chose to copy data item d_6 from $partition_1$ to $partition_2$. In the next step, the same move would still rank the highest. This is because the construction of hypergraph $H_{1\to 2}$ is oblivious to the fact that d_6 is also now present in $partition_2$. Further, it is also possible that, because of replication, neither of the partitions is actually accessed at all when executing the queries corresponding to e_4, e_5 or e_6.

To handle these two issues, during the execution of the algorithm, we maintain the exact list of partitions that would be activated for each query; this is calculated using the Set Cover algorithm described in Section 3. Now when we consider whether to copy a group of items from $partition_i$ to $partition_j$, we make sure that the benefit reflects the actual query span reduction given this mapping of queries to partitions. Pseudocodes for this algorithm is give in Algorithm 4 and 5.

4.6 3-Way Replication Algorithms

As we have already discussed, many large-scale data management systems provide default 3-way replication. Here we briefly discuss how the algorithms described above can be modified to handle 3-way replication.

PRA-Based 3-Way Replication: We identify PRA the most suitable algorithm to do this effectively, and modify PRA as follows.

Algorithm 4 Improved LMBR

```
Require: $\mathcal{H}(V,E), N, C$
 1: Run HPA to get initial partitions $\mathcal{G} = \{G_1, G_2, \ldots, G_N\}$ into $N$ partitions;
 2: Compute the set cover $MD_e$ for each query $e$;
 3: Initialize PQ (priority queue) to empty;
 4: for $g = G_1$ to $G_N$ do
 5:     for $g' = G_1$ to $G_N, g \neq g'$ do
 6:         PQ.insert($g \to g'$, maxGain($\mathcal{G}, g, g'$));
 7:     end for
 8: end for
 9: while all partitions are not full do
10:     ($g_{src} \to g_{dest}$) = PQ.bestMove();
11:     copy appropriate items from $g_{src}$ to $g_{dest}$;
12:     for $g = G_1$ to $G_N, g \neq g_{dest}$ do
13:         PQ.update($g \to g_{dest}$, maxGain($\mathcal{G}, g, g_{dest}$));
14:         PQ.update($g_{dest} \to g$, maxGain($\mathcal{G}, g_{dest}, g$));
15:     end for
16: end while
17: return final partitions $G_1, \cdots, G_N$;
```

Algorithm 5 Improved LMBR maxGain Method

```
Require: $\mathcal{G} = \{G_1, \cdots, G_N\}, \mathcal{H}(V,E), G_{src} \in \mathcal{G}, G_{dest} \in \mathcal{G}$
 1: $E_{src} = \{e \in E \mid$ getAccessedItems($\mathcal{G}, e, G_{src}$) $\neq \phi\}$;
 2: $E_{dest} = \{e \in E \mid$ getAccessedItems($\mathcal{G}, e, G_{dest}$) $\neq \phi\}$;
 3: $E = E_{src} \cap E_{dest}$;
 4: if $|E| \neq 0$ then
 5:     $V' = \cup_{e \in E}$ getAccessedItems($\mathcal{G}, e, G_{src}$);
 6:     $E' = \{$getAccessedItems($\mathcal{G}, e, G_{src}$)$|e \in E\}$;
 7:     create hypergraph $\mathcal{H}'(V', E')$;
 8:     $C_{dest} = C - |G_{dest}|$;
 9:     if $C_{dest} \neq 0$ then
10:         $H'$ = pruneHypergraphToSize($H', C_{dest}$);
11:         while $|H'| > 0$ do
12:             compute gain = $|E'|/|V'|$
13:             remove lowest degree node from $H'$ and incident edges;
14:         end while
15:     end if
16: end if
17: return the best value of gain found in the process and the corresponding $V'$;
```

Because we are interested in replicating all the nodes 3-way, we eliminate the step of finding important nodes from PRA and we replicate each node 3-way by using our "hitting set" technique to decide which copy must be shared with what hyperedges. PRA basically aims to separate the incident hyperedges in the hypergraph by distributing the copies of node d smartly to incident hyperedges.

Simple Distribution Algorithm: In this algorithm, for each node d in the hypergraph we find the set of incident hyperedges E_d. We assign 3 copies of d among $|E_d|$ edges randomly, by assigning every $\frac{|E_d|}{3}$ hyperedges single copy of d. Only difference between this algorithm and PRA based 3-way replication algorithm is that PRA based algorithm makes best effort to distribute the copies of node d among incident hyperedges E_d.

IHPA-Based Algorithm: In IHPA for 3-way replication we run HPA to get partitioning without replication. We remove all the hyperedges with span 1 from the input graph, and run HPA again on the residual graph to get additional partitions. We repeat this process one more time to replicate each node exactly 3 times.

4.7 Discussion

We presented four heuristics for data placement with replication. There are clearly many other variations of these algorithms, some of which may work better for some inputs, that can be implemented quickly and efficiently using our framework and the core operations that it supports (e.g., finding dense subgraphs). In practice, taking the best of the solutions produced by running several of these algorithms would guarantee good data placements.

Finally, while describing the algorithms, we assumed a homogeneous setup where all partitions are identical and all data items have equal size. We have also extended the algorithms to the case of heterogeneous data items. The hMETIS package that we use and also other hypergraph partitioning packages, allow the nodes to have weights. For heterogeneous case the dense subgraph algorithm is modified to account for the weights, by removing the node with the lowest value of degree till we have nodes having total specified weight (for both DS and LMBR). Similarly, PRA is modified by allowing the replication in the original hypergraph such that total weight of replicated nodes is no greater than the sum of all extra available partition capacities. We omit the full details due to lack of space.

5. EXPERIMENTAL EVALUATION

We are building a trace-driven simulator to experiment with different data placement and scheduling policies. The simulator instantiates a number of partitions as needed by the experimental setup, uses a data placement algorithm for distributing the data among the partitions, and replays a query trace against it to measure the query span profiles.

We conducted an extensive experimental study to evaluate our algorithms, using several real and synthetic datasets. Specifically, we used the following three datasets:

- Random: Instead of generating a query workload completely randomly, we use a different approach to better understand the structure of the problem. We first generate a random *data item graph* of a specified density (edges to nodes ratio). We then randomly generate queries such that the data items in the query form a connected subgraph in the data item graph. For low density data item graphs, this induces significant structure in the query workload that good data placement algorithms can exploit for better performance. Figure 5(a) shows an example data object graph where the numbers indicate the data item sizes (in MB). Figure 5(b) shows several queries that may be generated using this data item graph – each of the queries forms a connected subgraph in the data item graph.
- Snowflake: This is a special case of the above where the data item graph is a tree. This workload attempts to mimic a standard SQL query workload. An example data item graph corresponding to the Snowflake dataset is shown in Figure 5(c). Here the large squares indicate the first-level relations, and the small squares indicate the second-level relations. We treat each column of each relation as a separate data item. An SQL query over such a schema that does not contain a Cartesian product corresponds to a connected subgraph in this graph.
- ISPD98 Benchmark Data Sets: In addition to the above synthetic datasets, we tested our algorithms on standard ISPD98 benchmarks [5]. ISPD98 circuit benchmark suite contains 18 circuits ranging from 12,752 to about 210,000 nodes. Hypergraph density (hyperedges to nodes ratio) in all the ISPD98 circuit benchmarks is close to 1, i.e., these graphs are quite sparse. We show results for the first 10 circuit datasets, that contain 12,752 to 69,429 nodes.

We compare the performance of six algorithms: (1) **Random**, where the data is replicated and distributed randomly, (2) **HPA**, the baseline hypergraph partitioning algorithm, (3-6) the four algorithms

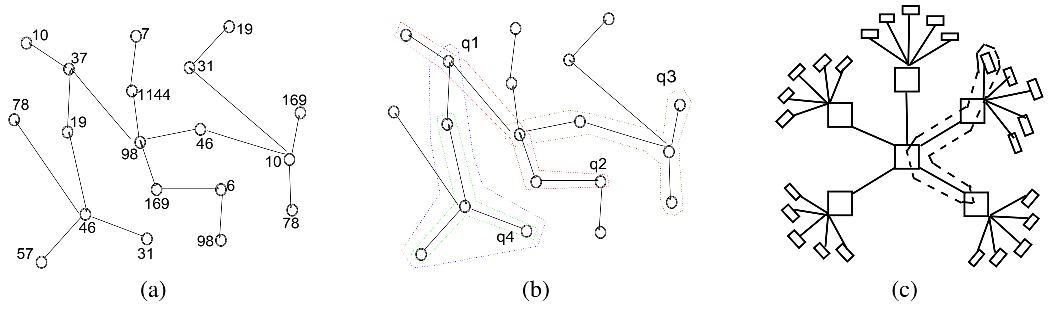


Figure 5: (a) An example data item graph; (b) Queries $q1, q2, q3, q4$ are generated by choosing connected subgraphs of the data item graph; (c) The data item graph corresponding to a Snowflake schema.

hMETIS (HPA) Parameter Values	
Parameters	**value**
$noPartitions$	Varies
$UBfactor$	1 for almost balanced partitioning, else varies
$Nruns$	50
$CType$	2
$RType$	1
$VCycle$	1
$Reconst$	1
$dbglvl$	0

Table 1: HPA Parameter Values

that we propose, **IHPA, PRA, DS,** and **LMBR** (Section 4). We use the hMETIS hypergraph partitioning algorithm [24, 1] as our HPA algorithm. For reproducibility, we list the values of the remaining hMETIS parameters in Table 1. The experiments were run on a Intel Core2 Duo CPU 2.10GHz, 4GB RAM, Windows PC running Windows 7. All plotted numbers (except the numbers for the ISPD98 benchmark) are averages over 10 random runs.

The key parameters of the dataset that we vary are: (1) $|D|$, the number of data items, (2-3) *minQuerySize* and *maxQuerySize*, the bounds on the query sizes that are generated, (4) *NQ*, the number of queries, (5) *C*, the partition capacity, (6) *numPartitions (NPar)*, the number of partitions, and (7) *density* of the data item graph (defined to be the ratio of the number of edges to the number of nodes). The default values were: $|\text{D}|$ = 1000, minQuerySize = 3, maxQuerySize = 11, NQ = 4000, C = 50, NPar = 40, and density = 20.

In several of the plots, we also show the average number of data items per query, denoted *ADI*.

5.1 Random Dataset

We begin with showing the results for the Random dataset with homogeneous data items.

Increasing Number of Partitions (ND): First, we run experiments with increasing the number of partitions. With the default parameters, a minimum of 20 partitions are needed to store the data items. We increase the number of partitions from 20 to 45, and compute the average query spans, and average execution times, for the six algorithms over 10 runs. Figures 6(a), and 6(b) show the results of the experiment. HPA does not do replication, and hence the corresponding plot is a straight line. The performance of the rest of the algorithms, including Random, improves as we allow for replication. Among those, LMBR performs the best, with IHPA a close second. We saw this behavior consistently across almost all of our experiments (including the other datasets). LMBR's performance does come with a significantly higher execution times as shown in Figure 6(b). This is because LMBR tends to do a lot of small moves, whereas the other algorithms tend to have a small number of steps (e.g., DS runs the densest subgraph algorithm a fixed number of times, whereas PRA only has three phases). Since data placement is a one-time offline operation, the high execution time of LMBR may be inconsequential compared to the reduction in *query span* it guarantees.

Increasing Query Size (ADI): Second, we vary the number of data items per query from 2 to 10 (by setting minQuerySize = maxQuerySize), choosing the default values for the other parameters. As expected (Figure 6(c)), the average span increase rapidly as the query size increases. The relative performance of the different algorithms is largely unchanged, with LMBR and IHPA performing the best.

Increasing Number of Queries (NQ): Next, we vary the number of queries from 1,000 to 11,000, thus increasing the density of the hypergraph (Figure 6(d)). The average query span increases rapidly in the beginning and much more slowly beyond 5,000 queries. Once again the LMBR algorithm finds the best solution by a significant margin compared to the other algorithms.

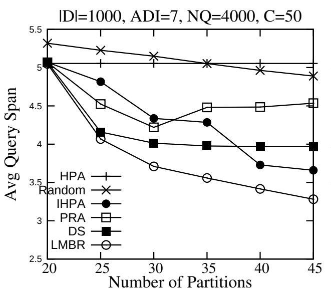
Increasing Data Item Graph Density: Finally, we vary the data item graph density while from 2 (very sparse) to 20 (dense). The number of partitions was set to 40. As we can see in Figure 6(e), for low density graphs, the average span of the queries is quite low, and it increases rapidly as the density increases. Note that the average query size did not change, so the performance gap is entirely because of the structure of the query hypergraph for low density data item graphs. Further, we note that the curves flatten out as the density increases, and don't change significantly beyond 10, indicating that the query workload essentially looks random to the algorithms beyond that point.

Overall, our experimental study indicates that LMBR, despite its high running time, should be the data placement algorithm used for minimizing query span/multi-site overheads and energy consumption in such scenarios (where we do not have any constraints on the number of replicas that must or can be created).

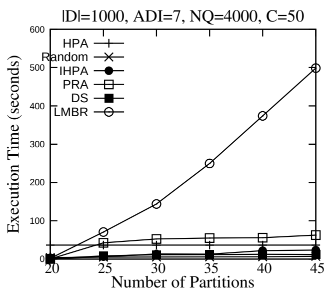
5.1.1 3-Way Replication

Figures 6(f), 6(g) and 6(h) show a set of experimental results comparing the 3-way replication algorithms that we have discussed in Section 4.6.

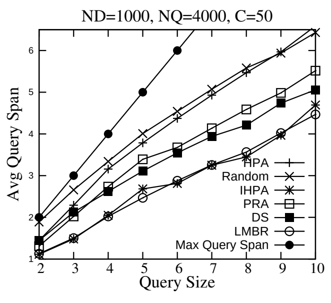
Increasing Number of Queries (NQ): Increasing the number of queries, thus increasing the density of the graph, we observe that PRA based 3-way replication algorithm performs the best. This



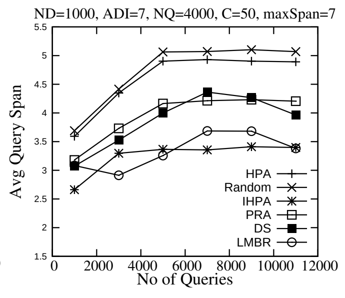
(a)



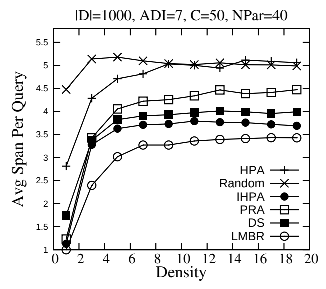
(b)



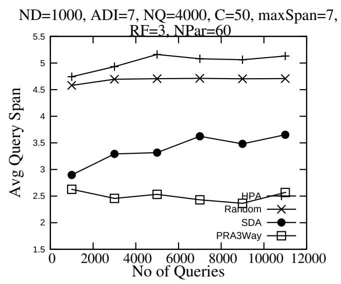
(c)



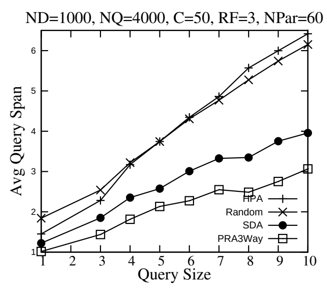
(d)



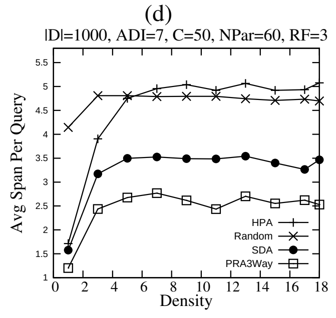
(e)



(f)



(g)



(h)

Figure 6: $(a)-(e)$ Results of the experiments on the Random dataset with homogeneous data items illustrate the benefits of intelligent data placement with replication; the LMBR algorithm produces the best data placement in almost all scenarios. Note that, for clarity, the y-axes for several of the graphs do not start at 0. $(f)-(h)$ 3-way replication results with replication factor of each node $RF = 3$.

is in comparison with HPA (no replication), Random 3-way replication and simple distribution algorithm (SDA). As the number of hyperedges increases in the graph average number of hyperedges incident per node also increases. This effects the SDA algorithm, because SDA tries to distribute the 3 copies of the node randomly to the number of hyperedges incident on it. So as average number of incident hyperedges per node increases, it is more likely for SDA to make bad decisions about distribution of replicas among incident hyperedges, hence SDA's average span increases with number of queries. On the other hand, PRA employs *hitting set* technique to do a more smarter replica distribution among the incident hyperedges. Increase in number of queries doesn't seem to effect the query span for PRA, which indicates the effectiveness of PRA approach. Hence, PRA based technique performs consistently better than SDA in this experiment.

Increasing Query Size (ADI): Query span for all the algorithms increases with an increase in average data items per query. As we saw that density of the hypergraph affects PRA and SDA, where increase in density doesn't affect PRA. In this experiment increase in hyperedge size doesn't affect the density of the hypergraph. Hence query span increases for SDA and PRA. PRA again performs consistently better than other algorithms.

Increasing Data Item Graph Density: PRA again performs the best compared to Random and SDA when density of the graph is varied. Analysis is similar to what we have discussed before in Section 5.1.

We do not compare with LMBR for this scenario due to its high running time, and because it cannot guarantee the replication constraint of 3-way replication.

5.2 Snowflake Dataset

Figures 7(a) and 7(b) show a set of experimental results for the Snowflake dataset. Each of the plotted numbers corresponds to an average over 10 random query workloads. The data item graph itself was generated with the following parameters: the number of levels in the graph was 3, the degree of each relation (the maximum number of tables it may join with) is set to 5, and the number of attributes per table is set to 15. The total number of data items was 2000, requiring a minimum of 20 partitions to store them. Note that we assume homogeneous data items in this case. We plot the average query spans, and the average execution times as the number of partitions increases from 20 to 45.

We also conducted a similar set of experiments with heterogeneous data item sizes, where we generated TPC-H style queries with data item sizes adhering to the TPC-H benchmark. We chose the scale factor of 25, which means the highest data item size is 28GB and smallest data item size is 25KB. This results in a high skew among the table column sizes. Data item size is calculated as $Size(columnDatatype) * noRows$. The partition capacity was fixed at 100GB, and we once again plot the average query spans and the average execution times as the number of partitions increases from 20 to 45. The results are shown in Figures 8(a) and 8(b).

Our results here corroborate the results on the Random dataset. We once again see that LMBR performs the best, finding significantly better data layouts than the other algorithms. The performance differences are quite drastic with homogeneous data item sizes – with 45 partitions, LMBR is able to achieve an average query span of just 1.5, whereas the baseline HPA results in an average span of 3.5. However, we observe that with heterogeneous data item sizes, the advantages of using smart data placement algorithms are lower. With an extreme skew among the data item sizes, the replication and data placement choices are very limited.

5.3 ISPD98 Benchmark Dataset

Finally, Figure 9 shows the comparative results for first ten of hypergraphs from the ISPD98 Benchmark Suite, commonly used in the hypergraph partitioning literature. The number of hyperedges in the datasets range from 14111 to 75196 and number of nodes range from 12752 to 69429. Here we set the partition capacity so that exactly 20 partitions are sufficient to store the data items, and we plot the results with number of partitions set to 35. The hy-

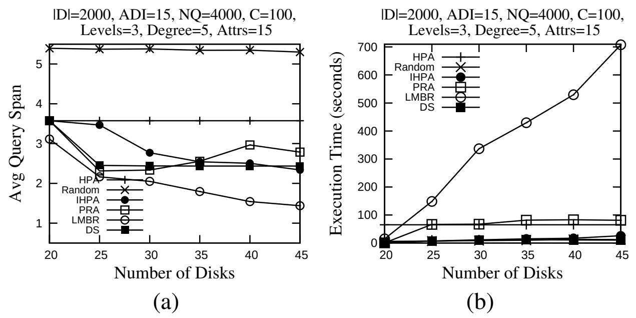


Figure 7: Results of the Experiments on the Snowflake Dataset

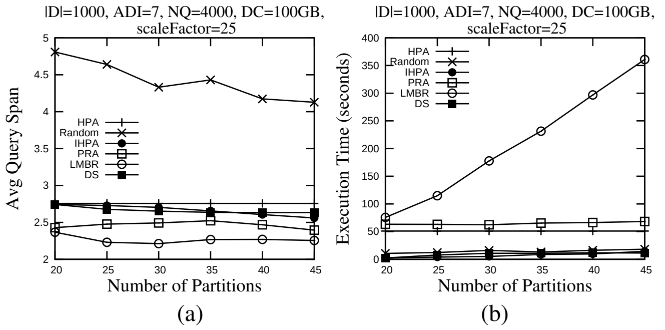


Figure 8: Results of the Experiments on a TPC-H style Benchmark with unequal data item sizes. The relation sizes were calculated assuming a scale factor of 25.

pergraphs in this dataset tend to have fairly low densities, resulting in low query spans. In fact, LMBR is able to achieve an average query span of close to the minimum possible (i.e., 1) with 35 partitions. Most of the other algorithms perform about 20 to 40% worse compared to LMBR.

These additional experiments further corroborate our claim that intelligent data placement with replication can significantly reduce the coordination overheads in data centers, and further that our LMBR algorithm outperforms rest of the algorithms significantly.

6. CONCLUSIONS

In this paper, we solve the combined problem of data placement and replication, given a query workload, to minimize the total resource consumption and by proxy, the total energy consumption, in very large distributed or multi-site read-only data stores. Directly optimizing for either of these metrics is likely infeasible in most practical scenarios because of the large number of factors involved. We instead identify query span, the number of machines involved in executing a query, as having a direct and significant impact on the total resource consumption, and focus on minimizing the average query span for a given query workload. We formulated and analyzed the problems of data placement and replica selection for this metric, and drew connections to several well-studied graph theoretic concepts. We used these connections to develop a series of algorithms to solve this problem, and our extensive experimental evaluation over several datasets demonstrated that our algorithms can result in drastic reductions in average query spans. We are planning to extend our work in several different directions. As we discussed earlier, we believe that temporal scheduling algorithms can be used to correct the load imbalance that may result from optimizing for query span alone; although analysis tasks are usually not

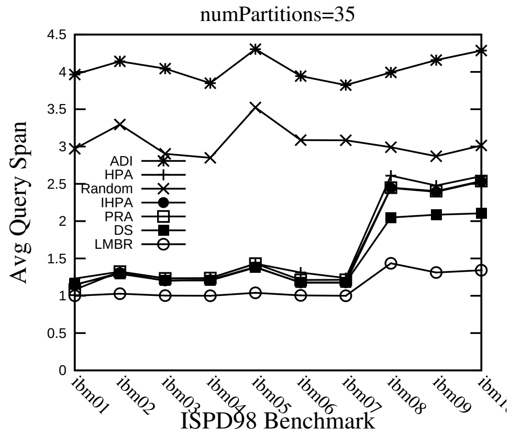


Figure 9: Results of the experiments on the first 10 hypergraphs, *ibm01, …, ibm10*, from the ISPD98 Benchmark Dataset

latency sensitive, there are still often deadlines that need to be satisfied. We plan to study how to incorporate such deadlines into our framework. We are also planning to study how to efficiently track changes in the query workload nature online, and how to adapt the replication decisions online.

7. ADDITIONAL AUTHORS

8. REFERENCES

[1] hMETIS: A hypergraph partitioning package, http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview.

[2] MLPart, http://vlsicad.ucsd.edu/gsrc/bookshelf/slots/partitioning/mlpart/.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, August 2009.

[4] N. Alon, P. D. Seymour, and R. Thomas. A separator theorem for graphs with an excluded minor and its applications. In *STOC*, 1990.

[5] C. J. Alpert. The ISPD98 circuit benchmark suite. In *Proc. of Intl. Symposium on Physical Design*, 1998.

[6] H. Amur, J. Cipar, V. Gupta, G. Ganger, M. Kozuch, and K. Schwan. Robust and flexible power-proportional storage. In *SoCC*, 2010.

[7] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. Greedily finding a dense subgraph. In *SWAT*, 1996.

[8] R. B. Boppana. Eigenvalues and graph bisection: An average-case analysis. In *FOCS*, 1987.

[9] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Design and implementation of move-based heuristics for VLSI hypergraph partitioning. *J. Exp. Algorithmics*, 5:5, 2000.

[10] M. M. M. K. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *SIGCOMM*, pages 98–109, 2011.

[11] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Supercomputing*, 2002.

[12] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational cloud: a database service for the cloud. In *CIDR*, pages 235–240, 2011.

[13] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[14] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3:515–529, September 2010.

[15] Z. Du, J. Hu, Y. Chen, Z. Cheng, and X. Wang. Optimized qos-aware replica placement heuristics and applications in astronomy data grid. *Journal of Systems and Software*, 84(7):1224 – 1232, 2011.

[16] D. Economou, S. Rivoire, and C. Kozyrakis. Full-system power analysis and modeling for server environments. In *In Workshop on Modeling Benchmarking and Simulation (MOBS)*, 2006.

[17] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011.

[18] U. Feige, G. Kortsarz, and D. Peleg. The dense k-subgraph problem. *Algorithmica*, 1999.

[19] H. Ferhatosmanoglu, A. S. Tosun, and A. Ramachandran. Replicated declustering of spatial data. In *PODS*, 2004.

[20] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75 – 174, 2010.

[21] M. Garey and D. Johnson. *"Computers and Intractability: A Guide to the Theory of NP-Completeness"*. 1979.

[22] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive replication in peer-to-peer systems. In *ICDCS*, 2004.

[23] L.-Y. Ho, J.-J. Wu, and P. Liu. Optimal algorithms for cross-rack communication optimization in mapreduce framework. In *IEEE International Conference on Cloud Computing*, 2011.

[24] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *IEEE VLSI*, pages 69–529, 1999.

[25] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proc. of DAC*, pages 343–348, 1998.

[26] M. Koyutürk and C. Aykanat. Iterative-improvement-based declustering heuristics for multi-disk databases. *Information Systems*, 2005.

[27] D.-R. Liu and S. Shekhar. Partitioning similarity graphs: A framework for declustering problems. *Information Systems*, 1996.

[28] H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. *J. Parallel Distrib. Comput.*, 69(9), 2009.

[29] T. A. Neves, L. M. de A. Drummond, L. S. Ochi, C. Albuquerque, and E. Uchoa. Solving replica placement and request distribution in content distribution networks. *Electronic Notes in Discrete Mathematics*, 36:89–96, 2010.

[30] K. Y. Oktay, A. Turk, and C. Aykanat. Selective replicated declustering for arbitrary queries. In *Euro-Par*, 2009.

[31] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[32] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[33] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *Supercomputing*, 2004.

[34] K. Ranganathan and I. Foster. Identifying dynamic replication strategies for a high-performance data grid. In *GRID*, 2001.

[35] K. Ranganathan, A. Iamnitchi, and I. Foster. Improving data availability through dynamic model-driven replication in large peer-to-peer communities. In *CCGRID*, 2002.

[36] M. Shorfuzzaman, P. Graham, and R. Eskicioglu. Adaptive popularity-driven replica placement inÂăhierarchical data grids. *The Journal of Supercomputing*, 51:374–392, 2010.

[37] H. D. Simon and S.-H. Teng. How good is recursive bisection? *SIAM J. Sci. Comput.*, 18(5):1436–1445, 1997.

[38] D. Thain and M. Livny. Building reliable clients and servers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[39] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2:1626–1629, August 2009.

[40] A. A. Tosun and H. Ferhatosmanoglu. Optimal parallel I/O using replication. In *ICPP*, 1997.

[41] A. S. Tosun. Replicated declustering for arbitrary queries. In *ACM symposium on Applied computing*, 2004.

[42] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 1st edition, June 2009.

[43] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM TODS*, 22:255–314, 1997.

[44] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM TODS*, 16:181–205, March 1991.

[45] L. Zhang, H. Tian, and S. Steglich. A new replica placement algorithm for improving the performance in cdns. *Int. J. Distrib. Sen. Netw.*, 5:35–35, January 2009.

[46] M. T. zsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer, 3rd edition, 2011.

APPENDIX

A. ANALYSIS FOR GENERAL GRAPHS

Given a graph $G = (V, E)$ (special case when the hypergraph $\mathcal{H}$ has size two edges) – our objective is to store the data items in a collection of partitions, each of capacity C. For each edge the cost is either 1 or 2. This gives rise to a trivial 2-approximation since $|E|$ is a lower bound on the optimal solution and $2|E|$ is a trivial upper bound on the solution that picks an arbitrary layout. Note that replication is allowed, and we may store more than one copy of each data item.

Assume that there is an optimal solution that creates at least one copy of each data item – uses $N_e (= \frac{n}{C})$ partitions (for simplicity we assume that n is a multiple of C). We now prove the bound for the following method. We order the nodes in decreasing order by degree.

For each node v_i, assume that E_i is the set of edges adjacent to v_i that go to nodes v_j with $j > i$. We use N_i partitions to store v_i where in the first partition we store v_i together with its first $C - 1$ neighbors, the second partition with v_i together with its next $C - 1$ neighbors etc. We thus use $N_i = \lceil \frac{|E_i|}{C-1} \rceil$ partitions for each node v_i.

The total number of partitions used is $\sum_{i=1}^{n} N_i = \sum_{i=1}^{n} \lceil \frac{|E_i|}{C-1} \rceil$.

Now consider an optimal solution with cost OPT that stores the nodes of G using N' partitions. Note that with N' partitions, each holding C nodes, the maximum number of local edges (edges for which the optimal solution incurs a cost of 1) within each partition is at most $\frac{C(C-1)}{2}$. We thus get $|E^*| \leq N' \frac{C(C-1)}{2}$ where E^* is the set of local edges in an optimal solution. Note that $OPT = |E^*| + 2(|E| - |E^*|) = 2|E| - |E^*|$ where OPT is the cost of an optimal solution.

We first note that if $|E^*| \leq \alpha|E|$ then we get a better lower bound on OPT, namely that $OPT \geq (2-\alpha)|E|$. Thus our solution, which has cost at most $2|E| \leq \frac{2}{2-\alpha} OPT$. This gives us a good approximation when α is significantly smaller than 1.

If $|E^*| > \alpha|E|$ then we get $\alpha|E| < |E^*| \leq N' \frac{C(C-1)}{2}$. Dividing by $\alpha(C - 1)$ we get $\frac{|E|}{C-1} < |E^*| \leq N' \frac{C}{2\alpha}$. Since $|E| = \sum_i |E_i|$ we get $\sum_i \frac{|E_i|}{C-1} < |E^*| \leq N' \frac{C}{2\alpha}$.

Recall that the total number of partitions we used is $\sum_{i=1}^{n} N_i = \sum_{i=1}^{n} \lceil \frac{|E_i|}{C-1} \rceil$. Ignoring the fact that we really need to take the ceiling, we can re-write this as $\sum_{i=1}^{n} \frac{|E_i|}{C-1} < N' \frac{C}{2\alpha}$. If $N' = \beta \frac{n}{C}$ for some constant β, then we get $\frac{n\beta}{2\alpha}$ as the bound on the number of partitions

We thus conclude:

THEOREM 4. *If the optimal solution uses βN_e partitions, where $N_e = \frac{|G|}{C}$ then either we can get an approximation with factor $\frac{2}{2-\alpha}$ for $0 \leq \alpha \leq 1$ using N_e partitions, or a placement in which each edge is contained in a single partition using $\frac{C N_e \beta}{2\alpha}$ partitions.*