# Neuroshard: Towards Automatic Multi-objective Sharding with Deep Reinforcement Learning

Tamer Eldeeb
Columbia University
tamer.eldeeb@columbia.edu

Zhengneng Chen
Seamoney
zhengneng.chen@seamoney.com

Asaf Cidon
Columbia University
asaf.cidon@columbia.edu

Junfeng Yang
Columbia University
junfeng@cs.columbia.edu

## ABSTRACT

Large databases whose data does not fit on a single server need to shard their rows across multiple different database instances. Distributed transactions are significantly more expensive than local transactions, so a popular approach is to collect a trace of past accesses to the database and model it as a graph (or a hypergraph), and solve an NP-Hard partitioning problem with an objective of minimizing the fanout, or the number of database instances that need to participate in each query. Due to the large amount of data that needs to be sharded, this problem cannot be solved optimally, and therefore, databases use heuristic partitioning algorithms, which can be fairly effective in practice. However, fanout is only one objective that affects performance. Other important objectives include load balancing, which ensures that no single database instance becomes too overloaded, or equalizing the write traffic for each database to avoid lock contention and I/O amplification. Designing heuristics for more than one objective is difficult and error-prone.

We present Neuroshard, the first system that learns shard assignments directly from the workload, and optimizes for multiple sharding objectives simultaneously. Neuroshard represents past queries as a neural hypergraph, and uses Deep Reinforcement Learning with Multi-Task learning to generate a learned partitioner that is able to optimize for multiple objectives in parallel. We implement Neuroshard on a distributed database that uses MariaDB, and got very promising initial results showing that this approach can achieve our versatility and scalability goals, in contrast to baseline approaches that optimize for only one objective which can work well in one context but perform poorly in another.

**ACM Reference Format:**
Tamer Eldeeb, Zhengneng Chen, Asaf Cidon, and Junfeng Yang. 2022. Neuroshard: Towards Automatic Multi-objective Sharding with Deep Reinforcement Learning. In *Exploiting Artificial Intelligence Techniques for Data (aiDM'22 ), June 17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533702.3534908


*aiDM'22 , June 17, 2022, Philadelphia, PA, USA*

ACM ISBN 978-1-4503-9377-5/22/06…$15.00
https://doi.org/10.1145/3533702.3534908

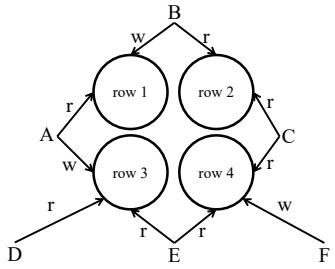
## 1 INTRODUCTION

Horizontal sharding is a decades-old technique to scale production databases. When a database's load or storage capacity overwhelms a single server, operators split the rows in the database and store them in multiple servers. Notable examples of horizontal sharding are Facebook's social graph [31] and Google's ad serving database [39].

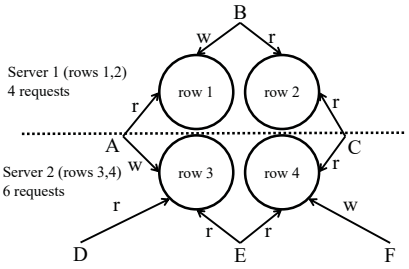
The choice of assigning rows to servers affects many aspects of a system's performance, so operators often need to simultaneously optimize for multiple, sometimes conflicting, objectives. One example objective is to assign a roughly equal number of rows to each server, balancing storage load. Another is to assign rows such that the servers receive roughly the same number of queries, balancing compute and network load. This problem is exacerbated by real-world requirements, such as variable-sized objects, servers with heterogeneous capacities, and queries with variable complexity. Beyond load balancing, another class of objectives is to minimize the fanout – the number of shards that a query touches – because a distributed query is typically much more expensive than a single-server query. However, fanout minimization requires clustering the rows assessed together frequently into the same shard, which is at odds with load balancing objectives if many rows in the cluster are hot.

Given these intricate, combinatorial objectives, an ideal sharding scheme should flexibly adapt and optimize for them simultaneously. Unfortunately, existing sharding algorithms are designed primarily for single objectives. For instance, commonly-used random, hash, range and round-robin partitioning [15] are good at balancing load but ignore fanout minimization completely. To minimize average query fanout, recent work, including Schism [12], SWORD [36] and SHP [22], collect a trace of past accesses to the database, model it as a (hyper)graph that links the rows accessed together in a query, and then compute the row-to-shard assignments by solving a partitioning problem. In general, graph partitioning is an NP-Hard problem [3], so existing work uses solvers that utilize hand-designed heuristics that require considerable expertise and experimentation to produce good solutions for just the objective of minimizing fanout. Given the diverse and competing objectives, designing hand-crafted heuristics that work well for each objective and their combinations is a painstaking task.

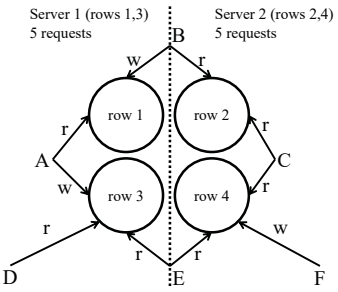
This paper presents *Neuroshard*, a sharding system that tackles the decades-old horizontal scaling problem using a novel learned approach. Neuroshard can automatically optimize for multiple classic




**(a) A set of requests (A-F) accessing 4 database rows.**




**(b) Partition option 1: rows {1,2} and rows {3,4} are each stored on a separate database. While this partitioning requires only two of the accesses (A,C) to involve more than one database, it leads to a load imbalance.**




**(c) Partition option 2: rows {1,3} and rows {2,4} are each stored on a separate database. This partitioning both minimizes the number of queries that require more than one database, as well as balance the load.**

**Figure 1: Toy example of the sharding problem. In the example, we have 4 rows, which are accessed by 6 requests (marked A-F). Each request either reads (r) or writes (w) one or two of the rows. Our goal is to partition the rows into two equal shards.**

sharding objectives, such as fanout minimization and load balancing, by learning effective partitioning heuristics directly from the input trace.

Similar to prior work [22, 36] Neuroshard works by collecting a trace of past queries and representing it as a hypergraph, where the rows are vertices and the queries are hyperedges, and then partitions that hypergraph. This process is done regularly offline at some frequency. The intuition is that computing a sharding assignment that works well for past accesses will most likely serve to improve the performance on future accesses as well. This offline approach is orthogonal and complementary to online sharding in systems [38, 43] in which the system actively moves rows to improve the performance.

Neuroshard utilizes Deep Reinforcement Learning (RL) to produce a learned partitioner that is then used to partition the hypergraph of past queries. However, the challenge in directly applying RL to our setting is that the query hypergraph may contain thousands or even millions of queries (or hyperedges), which would make the the state and action space of the RL quite large. Inspired by prior work on graph partitioning [47], we make the observation that if at each stage of partitioning we only consider adding the "neighboring" rows (or the rows that were accessed in the same query), we can achieve good partitions, while also significantly limiting the number of vertices we consider at each step of the algorithm. This structure naturally lends itself to a reinforcement learning formulation. Therefore, Neuroshard uses a trained RL agent to score each one of the candidate hypergraph elements, where the reward is a function of multiple sharding objectives. Neuroshard uses techniques from Multi-Task learning to optimize for multiple objectives in parallel.

We implement Neuroshard on a distributed database based on MariaDB [1], and compare it to three heuristic baselines: Neighbor Expansion [47], hMetis [23] and hash partitioning. We compare the different algorithms on microbenchmarks and on the Epinions [30] social network dataset. Our evaluation shows that while Neuroshard does not always provide the best performance, it consistently provides good performance, while the heuristic-based schemes' performance varies. For example, hMetis and Neighbor Expansion perform well in workloads where fanout minimization is a primary objective and the load is spread relatively evenly across queries, but perform poorly in skewed workloads where load balancing is an important objective. In contrast, Neuroshard is able to balance multiple objectives simultaneously (*i.e.,* fanout and load balancing) in both of these types of workloads.

We make three primary contributions:

(1) **Learned sharding.** Neuroshard is the first system to use a learned approach for directly assigning rows to shards.
(2) **RL formulation.** A novel RL framework for the hypergraph partitioning problem (§4). Our design uses ideas from the Neighbor Expansion [47] algorithm to restrict the state and action spaces so that the RL agent only needs to learn how to make good local decisions based on a small subset of the hypergraph (the neighborhood). We formulate two popular sharding objectives as RL rewards: fanout minimization (§4.4) and load balancing (§4.5)
(3) **Multiple objectives.** A general approach for multi-objective sharding using Multi-Task Learning (§6) that can automatically incorporate new objectives as RL rewards.

## 2 MODELING THE SHARDING PROBLEM

The sharding problem can be formally modeled as a hypergraph partitioning problem [12, 22, 36]. A hypergraph is a generalization of a graph for which each edge (called hyperedge) can connect any number of vertices rather than just two. In the case of database sharding, a trace of recent past queries to the database that records the rows accessed by each query can be thought of as a hypergraph, with the rows as the vertices (*e.g.,* rows 1-4 in our toy example in figure 1) and the queries (*e.g.,* queries A-F in our example), which may involve more than 2 rows, as the hyperedges.

The goal of hypergraph partitioning is to divide the vertices of a hypergraph into a number of equal size partitions, usually with a goal such as minimizing the number of hyperedges that cross partitions (fanout), or equalizing the sum of the degrees of the vertices in a partition (load), which we formalize next.

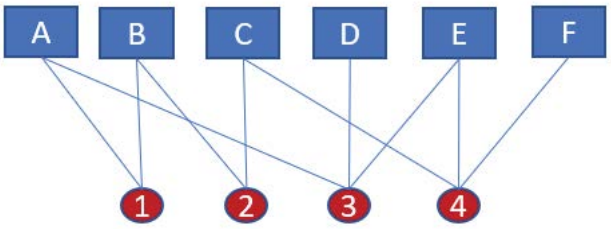



**Figure 2: Toy example represented as a bipartite graph.**

## 2.1 Fanout minimization

We follow the notation of SHP [22] and describe the hypergraph fanout minimization problem in the equivalent terms of bipartite graphs, where the vertices are either requests and rows, and edges are drawn between each request and the rows it is accessing. As a reference, our toy example is depicted in a bipartite graph form in Figure 2.

Let G = (Q ∪ R, E) be an undirected bipartite graph with disjoint sets of query vertices, Q, and row vertices, R. The goal is to partition R into k parts, i.e. find a collection of k disjoint subsets $V_1,...,V_k$ (also called partitions) covering R that minimizes an objective function. The output partitions should be balanced, that is,

$$|V_i| \leq (1+\epsilon)\frac{n}{k}$$

for all $1 \leq i \leq k$ and some $\epsilon \geq 0$, where n = |R|. Intuitively, this captures the requirement that partitions should all be assigned a similar number of vertices (or rows in our case). Given a partitioning $P = \{V_1,...,V_k\}$ and a query vertex $q \in Q$, informally the fanout of q is the number of distinct partitions that contain row vertices adjacent to q. Formally, we can define

$$fanout(P, q) = |\{V_i : V_i \cap N(q) \neq \emptyset\}|$$

Where N(q) is the set of vertices adjacent to q in G. Note that we have $N(q) \subseteq R$, since the graph is bipartite. We can now define the quality of the partitioning P as the average query fanout:

$$fanout(P) = \frac{1}{|Q|}\sum_{q \in Q} fanout(P, q)$$

The fanout minimization problem is, given a bipartite graph G, an integer k > 1, and a real number $\epsilon \geq 0$, find a partitioning of G into k partitions with the minimum average fanout.

## 2.2 Load Balancing

Another very common objective for sharding is load balancing. The setting for the load balancing problem is similar to that of fanout minimization. We are also given a hypergraph represented as a bipartite graph G = (Q ∪ R, E), and the goal is still to partition R into k partitions subject to the same constraints on the number of vertices assigned to each partition. Additionally, each vertex r in R has an associated value $\mathbf{W_r}$, which we call the *load weight*. This is meant to represent how much load is caused by processing queries that access a particular row r. It is up to the application to specify the load weight for each row; a natural scheme is to use the degree of the vertex in r as its load weight (which Neuroshard uses by default).

Given a partitioning $P = \{V_1,...,V_k\}$ and a partition $i \leq k$ we can define the load of partition i as the sum of the load weight of all vertices in $V_i$. Formally,

$$load(P, i) = \sum_{r \in V_i} W_r$$

One possible measure of load imbalance is the difference between the most and least loaded partitions. Let $\mathbf{p_{max}}$ be the partition with the highest value of load. Similarly, let $\mathbf{p_{min}}$ be the partition with the lowest value. We can now define the quality of the partitioning P as

$$Imbalance(P) = load(P, p_{max}) - load(P, p_{min})$$

Hence, the load balancing problem is, given a graph bipartite G, an integer k > 1, and a real number $\epsilon \geq 0$, find a partitioning of G into k partitions with the minimum imbalance.

We make a simplifying modelling decision by assuming that the weight load of a vertex is static and does not rely on the partitioning P. In reality, this may not be entirely accurate because the amount of work involved in a distributed query or transaction can be significantly more than a local one. As a result, the amount of load generated by an access to a row does depends on whether that access is distributed or local which is affected by the partitioning.

# 3 OVERVIEW OF LEARNED SHARDING

RL is concerned with the development of *agents* that learn from direct interaction with their environment. In an RL setting, an agent starts out knowing nothing about the given task, and learns by taking incremental actions, observing how these actions affect the environment, and receiving a reward that depends on its performance on the task. Despite not having any prior knowledge about the task, RL training algorithms allow the agent to improve its performance at it. This aligns very well with our goal of designing a general and versatile sharding framework that is able to perform well on various workloads and objectives, without needing to hand design heuristics for each one. By casting the sharding problem into an RL training problem, we only need to design the reward signal, and then leverage RL to train the sharding agent.

## 3.1 Deep Reinforcement Learning Primer

Combining classical RL with Deep Learning [17] techniques (dubbed Deep Reinforcement Learning [34]) has been key to many recent breakthroughs such as in game playing [40] and many applications in computer systems [9, 27, 28]. We now give an overview of the Deep RL concepts that we use in this paper, and in particular a family of techniques called policy gradient methods [42]. For a comprehensive treatment of the subject please refer to [41].

*Setting.* The usual setting of Reinforcement Learning is the discrete-time Markov Decision Process (MDP), in which an *agent* is interacting with an *environment*. At each time step $\mathbf{t}$ the agent observes some environment state $\mathbf{s_t}$, and performs an action $\mathbf{a_t}$. As a result of the action taken, the agent receives a reward $\mathbf{r_t}$ and the environment state transitions to $\mathbf{s_{t+1}}$. The state transitions and rewards are

stochastic and are assumed to have the Markov property; i.e. the probability of receiving a reward $r_t$ and the probability of transitioning to a state $s_{t+1}$ depend only on the state of the environment $s_t$ and the action taken by the agent $a_t$. The goal of the agent is to maximize the expected cumulative discounted reward:

$$\mathbb{E}\Big[\sum_t \gamma^t r_t\Big]$$

Where $\gamma \in (0, 1]$ is a factor discounting future rewards. At each step the agent takes a decision based on a *policy* $\boldsymbol{\pi}$, which defined as the conditional probability distribution of actions given states. In other words, $\pi$(s, a) is the probability that action a is taken when the state is s. For all but the most trivial of applications, the size of the state space makes it infeasible to store the policy explicitly in a tabular form. Hence, a common approach is to use function approximators to approximate $\pi$ using a manageable number of parameters $\theta$. Deep Neural Networks have emerged as a popular choice for the function approximator used to represent the parameterized policy $\pi_\theta$.

*Policy gradient methods.* Like prior work [27], we utilize a class of RL algorithms in Neuroshard learns by performing *gradient descent* over the policy parameters. In Neuroshard we use the REINFORCE [42] algorithm, which works by obtaining an estimate for the gradient of the cumulative discounted reward empirically by sampling (*i.e.,* running the MDP by following the current policy $\pi_\theta$) then updates the parameters using that estimate, according to the following formula:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta log \pi_\theta(s_t, a_t) v_t$$

Where $\alpha$ is the learning rate and $v_t$ is the empirically computed cumulative reward. Like prior work [27], we use a variant [37] of the REINFORCE algorithm that subtracts a baseline value from each return $v_t$, which is useful to reduce the variance of gradient estimates. We describe the training procedure in more detail in §5.2.

## 3.2 Sharding Problem Formulation for RL

We now need to formulate the sharding problem in RL terms. We have three main design considerations. First, problems with an incremental structure are good fits for RL because the agent can learn to perform a specific action and get an incremental reward at each step. Second, it is ideal if the agent needs to learn only how to make good local decisions. If the agent requires much global knowledge, training tends to be difficult and inference performance overhead tends to be high. Worse, a model trained on a graph may not generalize to a different graph. Third, it is desirable to reduce the action search space such that the agent model can stay simple.

These design considerations lead us to solution structure based on Neighbor Expansion (NE) [47], an algorithm for *graph* edge partitioning. The goal of the graph partitioning problem is to divide the *edges* of a graph evenly into equal-size partitions while minimizing the average number of partitions that are incident to a vertex (called the *replication factor*). It is analogous to the fanout minimization problem we defined previously on hypergraphs in §2.1. The NE algorithm then exploits graph structure and can produce high-quality partitions for the replication factor objective, and automatically balances the number of edges partitions in each partition. It also has the nice properties of being fast to run [47] and highly scalable [19], making it an effective approach for graph edge partitioning.

We design a Hypergraph Neighbor Expansion (HNE) algorithm, an adaptation of Neighbor Expansion to the hypergraph partitioning problem.

We show the pseudo code of HNE in Algorithm 1. The basic idea is fairly simple; the algorithm proceeds by building partitions sequentially. It maintains two hyperedge sets, the core set **C**, and the set **S** which is the set of all hyperedges incident to the vertices assigned to the current partition. We call a hyperedge q *unassigned* if it has not yet been added to the core set C of any partition. At every step (starting with a random seed hyperedge), one of the hyperedges in S (that is not already a member of C) is selected as a core hyperedge and added to the set C, then all its remaining vertices are added to the partition. All of C's unassigned neighbors (two hyperedges are *neighbors* or *adjacent* if they are both incident to at least one common vertex) are then added to the set S. This is repeated until the partition is filled up. A hand-crafted greedy heuristic determines which hyperedge to select at every step of the process to add to C.

Algorithm 1 describes an algorithmic framework that meets our design considerations. it has an *incremental* structure of assigning candidate hyperedges gradually. The decision to assign a hyperedge to a partition is *local*, depending on the neighboring relationship. It is *hyperedge-centric*, meaning that each step selects a hyperedge and then assigns all vertices on the hyperedge to a partion. This coarsens the action space compared to a *vertex-centric* approach that adds one vertex at a time. In an extremely skewed graph, a hyperedge may connect to many vertices, posing a problem for this treatment, but in practice, OLTP workloads which we target do not typically have such skewed distributions.

```
C ← ∅;
S ← ∅;
while Partition is not full do
    candidates ← S \ C;
    if candidates = ∅ then
        Seed candidates with a random, unassigned
          hyperedge;
    end
    Select the best candidate h based on scoring heuristic.;
    Add h to C.;
    Add all of h's unassigned adjacent hyperedges to S.;
    Add all of h's unassigned vertices to partition, and remove
      them from the hypergraph.;
end
```

**Algorithm 1:** Hypergraph Neighbor Expansion (HNE)

The scoring heuristic we use for HNE is also inspired by the neighborhood heuristic in the original Neighbor Expansion algorithm [47]. Let **HN(q)** be the set of unassigned neighbors of hyperedge q. The neighborhood heuristic we use is to select the candidate hyperedge that has the minimum value of | HN(q) \ S |. This choice is greedy in that it selects the hyperedge that results in the smallest increase in fanout at this step.

To simplify the presentation of Algorithm 1 we omitted the handling of the following corner case: If the partition fills up while adding h's vertices, we do not just add all of h's vertices and violate the balancing constraints. Instead, we initialize the procedure for the next partition by making S = {h} instead of the empty set. We also omit discussing how to handle vertices that do not have any incident hyperedges as these can be handled straightforwardly in various ways.

### 3.3 Neuroshard RL Algorithmic Framework

```
C ← ∅;
S ← ∅;
while Partition is not full do
    candidates ← S \ C;
    while |candidates| < threshold do
        Seed candidates with a random, unassigned
          hyperedge;
    end
    candidate_probabilities ← DNN(candidates,
      partition_state);
    Select a candidate hyperedge h based on probabilities;
    Add h to C.;
    Add all of h's adjacent hyperedges to S.;
    Add all of h's unassigned incident vertices to the partition.;
end
```

**Algorithm 2:** Neuroshard RL algorithmic framework.

Instead of using a hand-crafted scoring heuristic, Neuroshard adopts a trained agent, represented as a Neural Network, that assigns a probability to each hyperedge at each step of the algorithm. This agent is trained using RL, with a reward being a function of multiple objectives. As we show in §7.3, by using this approach Neuroshard is able to train directly on larger traces than prior work (e.g. [13]) and generalize to much larger hypergraphs. We highlight one other minor, but significant, difference from NE/HNE in Neuroshard: If the size of the candidate set drops below a threshold, the algorithm will seed with random unassigned hyperedges (if available) to keep the size of candidates set at the threshold. This is important, to avoid getting stuck with candidates that would be good only for only a single objective (e.g., fanout minimization) but hurt other objectives.

## 4 REINFORCEMENT LEARNING DESIGN

In this section, we describe the parts of training Neuroshard as a single-objective Reinforcement Learning problem, namely the **environment**, **state**, **actions**, and **rewards**.

### 4.1 Environment

In our RL setting, the environment is made up of 3 components:

*The Hypergraph.* This part of the environment is static and does not change throughout a training episode. It includes the structure of the hypergraph, i.e. the set of vertices and hyperedges, or the mapping of requests to data rows. The hypergraph is also augmented with important information, such as the degree of each vertex and hyperedge, whether a hyperedge is a read-only or read-write query, and the load weight of each vertex (in our experiments we use the vertex degree as the load weight, but this can be up to the application).

*Partition state.* This is information about the state of the partitions. It is represented by the vector $[\mathbf{T_v}, \mathbf{C_v}, \mathbf{C_l}, \mathbf{P}, \mathbf{Max_l}, \mathbf{Min_l}]$. These values are defined as follows: $\mathbf{T_v}$ is the target number of vertices per partition. This is a hard constraint. $\mathbf{C_v}$ is the number of vertices in the partition that's currently being built. $\mathbf{C_l}$ is the load weight of all the vertices in the current partition. $\mathbf{P}$ is the number of partitions left to be built after finishing the current partition. $\mathbf{Max_l}$ is the highest load of a complete partition so far. Similarly, $\mathbf{Min_l}$ is the minimum load of a complete partition. Both $\mathbf{Max_l}$ and $\mathbf{Min_l}$ are initialized to the target total load weight per partition (i.e. the sum of the load weight of all vertices divided by the number of partitions) at the beginning of the episode. Once a partition is complete, the new value of $\mathrm{Max_l}$ is computed by maxing the previous value with the new partition's load. The value of $\mathrm{Min_l}$ is updated in a similar fashion. This choice of initial value and how to update is significant; as we'll discuss in §4.5.

*Assignment state.* This is state associated with each vertex and hyperedge indicating their partition assignment status. Specifically, each element can be in one of three states: Unassigned, Assigned, or Assigned to current partition. For hyperedges, being assigned to a partition i means having been added to the core set C of i (note that, due to the corner case described at the end of §3.2, a hyperedge q can actually be assigned to more than 1 core set, but this does not require any special handling). Additionally, each hyperedge is also associated with a bit indicating whether it is in the set S of the current partition or not. For each hyperedge $q \in S$, we also maintain its HNE heuristic score, i.e. $| \mathrm{HN}(q) \setminus S |$.

### 4.2 Observable State

Technically, the agent has access to all of the environment state. However, it only makes use of a small portion to decide on the action to take, namely the neighborhood of the candidates set, as well as the partition state. Theoretically, this makes the problem harder since the agent's inability to take into account the full state results in a partially-observed MDP [35], but like prior work [27], we find that the approach works well in practice nevertheless. For efficiency, the agent is also passed the candidates set explicitly even though it can technically compute it from the environment state.

### 4.3 Actions

At every step the agent selects one hyperedge from the candidates set. Hence, its action space at every step is the same as the set of candidates. Once the agent makes a selection, the state of the environment transitions as described in Algorithm 1.

*4.3.1 Restricting the Size of the Candidates Set.* As noted by prior work [32], applying neighbor expansion to hypergraph partitioning can run into the challenge where the candidates set grows very large very quickly. A very large candidate set makes training very expensive in terms of memory and computational requirements

for each episode, as well as much harder in terms of the ability of the agent to learn useful rules as it would need to run more episodes to sample more trajectories. This is more common in hypergraphs that represent social networks since these can have extremely large hyperedges which are not very common in the OLTP workloads we target. Nevertheless, we found it helpful to limit the size of the candidates set considered by the agent at each step to an upper bound $\mathbf{Cand_{max}}$. This is a trade-off since that this adds a hyperparameter to the model that needs tuning for best performance. We find that randomly selecting a subset of the candidate hyperedges of size $\text{Cand}_{\text{max}}$ at each step works well in our evaluation, but other more disciplined strategies such as described by prior work [32] are also possible.

## 4.4 Fanout Reward

To guide the agent towards producing good solutions for the fanout objective, we design the reward signal as follows: Suppose the agent chose hyperedge $q_t$ as its action in time-step t. It receives a reward

$$r_t = \frac{-1}{|Q|}|HN(q_t) \setminus S|$$

Recall from §3.2 that HN(q) is the set of *unassigned* hyperedges that are adjacent to hyperedge q. Furthermore, we say that a hyperedge q is *incident* to a partition i iff $V_i \cap N(q) \neq \emptyset$. In other words, q is incident to partition i if any of the row vertices accessed by q are assigned to i. Note that the numerator of $r_t$ is the (negative of the) number of hyperedges that became incident to the partition as a result of the agent's action $a_t$, *i.e.,* the number of queries that are incident to the partition at time t+1 but were not already incident to the partition at time t. Hence, $r_t$ is the incremental change in the fanout objective as a result of $a_t$. In other words, we have

$$fanout(P) = -\sum_t r_t$$

.

RL training works to *maximize* the sum of expected rewards, which is the same as *minimizing* its negative. Hence, our reward signal design causes training to minimize fanout(P).

*4.4.1 Distinguishing Between Reads and Writes.* It is usually the case that distributed writes are significantly more expensive than reads. This is easy to account for in Neuroshard by adjusting the reward signal to weigh the reward differently based on whether $h_t$ represents a read-only or a read-write query.

## 4.5 Load-balancing Reward

Suppose the agent chose a hyperedge $\mathbf{q_t}$ as its action in time-step t. This causes a set of unassigned row vertices adjacent to $q_t$ to be assigned to the partition. We call this set of newly assigned row vertices $\mathbf{A_t}$. Let $\mathbf{WA_t}$ be the sum of the load weights for each vertex in $A_t$, that is,

$$WA_t = \sum_{v \in A_t} W_v$$

Recall from §4.1 that $\mathbf{Max_l}$ is the highest load of a complete partition so far, *i.e.,* up to time-step t. Likewise, $\mathbf{Min_l}$ is the lowest load of a complete partition so far. Let $\mathbf{load_t}$ be the load of the current partition at the start of time-step t. Note that $\text{load}_{t+1} = \text{load}_t + \text{WA}_t$.

The agent will receive a reward $r_t$ that is made of two different components. We start by defining the components:

- $\mathbf{rmax_t}$, defined as: $\max(0, WA_t - \max(0, load_t - Max_l))$. Note that if $\text{load}_{t+1} \leq \text{Max}_l$ this sets $\text{rmax}_t$ to 0. Otherwise, the $\text{rmax}_t$ is set to the incremental increase over $\text{Max}_l$ caused by action $a_t$.
- $\mathbf{rmin_t}$. The value of $\text{rmin}_t$ is always 0 if action $a_t$ does not cause the current partition to fill up and start a new partition. Otherwise, it is defined as: $|\max(0, Min_l - (load_t + WA_t))|$. In other words, if the newly-finished partition has at least the same load as $\text{Min}_l$, $\text{rmin}_t$ is set to zero. Otherwise, it is set to the difference in load.

Recall from §4.1 that both $\text{Max}_l$ and $\text{Min}_l$ are initialized to the same value $\mathbf{Target_l}$ at the beginning of the training episode. Also recall from §2.2 that, given a complete partitioning P, $\mathbf{p_{max}}$ and $\mathbf{p_{min}}$ are the most and least loaded partitions, respectively. Consider the way we defined $\text{rmax}_t$. Every time the agent's action causes the current partition's load to increase over the prior value of $\text{Max}_l$, we set $\text{rmax}_t$ to the difference. Hence, we have

$$\sum_t rmax_t = load(P, p_{max}) - Target_l$$

Similarly, we can show that

$$\sum_t rmin_t = Target_l - load(P, p_{min})$$

We can now define the reward $r_t$ that the agent receives:

$$r_t = -(rmax_t + rmin_t)$$

This has the following property:

$$\sum_t r_t = -(\sum_t rmax_t + \sum_t rmin_t) = -(load(P, p_{max}) - load(P, p_{min}))$$

.

Giving

$$\sum_t r_t = -Imbalance(P)$$

We now briefly discuss the choice of using $\text{Target}_l$ as the initial value for both $\text{Max}_l$ and $\text{Min}_l$. Initially, we chose not to define these values while processing the first partition, then use the load of the first partition to initialize the values starting from the second partition. However, we quickly realized that giving the agent the target load at the start is very helpful and makes attributing rewards to actions significantly easier. This way, the agent gets penalties (i.e. the negative-valued rewards) only and as soon as its choices start causing deviation from the ideal value.

# 5 NEURAL NETWORK AGENT

In this section, we describe the internal design of the agent including how it represents the policy as a neural network, and finish by describing the training procedure. We leverage a powerful technique called Graph Neural Network to capture key features of the underlying hypergraph, which enables a simple, three-layer policy network that yields effective sharding results (see §7).

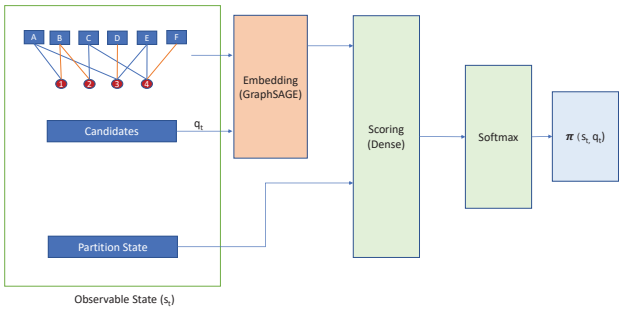



**Figure 3: Policy neural network architecture**

## 5.1 Agent Design

The agent's main function is to choose a hyperedge from the candidate set at every time step t. Let $\mathbf{s_t}$ be the observable state at time t (as described in §4.2), and $\mathbf{candidates_t}$ be the set of candidate hyperedges. Conceptually, the agent goes through three steps: First, it computes an embedding vector representation for each hyperedge q ∈ $\text{candidates}_t$. Second, it uses these embedding vectors to compute a score for each hyperedge q. Finally, these scores are converted to probabilities and a hyperedge $q_t$ is selected based on these probabilities. A deep neural network representing a parameterized policy $\boldsymbol{\pi_\theta(s, q)}$ is used to accomplish all these steps. As depicted in Figure 3, its architecture consists of three components, which we now describe.

*Embedding Layer.* Graph Neural Networks [49] (GNNs) are the standard tool used to create embedding vectors for graph elements such as vertices or edges. The goal is to encode the high-dimensional information about a vertexâĂŹs graph neighborhood into a dense vector embedding that is suitable as input to downstream layers [18]. Given our bipartite graph representation of the hypergraph, we are able to leverage graph neural network architectures to generate an embedding vector representation for each hyperedge q in $\text{candidates}_t$. We use GraphSAGE [18], a popular GNN architecture in Neuroshard as the embedding layer. We set the hyper-parameter K to 2, which means the computed embedding vectors for a vertex contain information from its 2-hop neighborhood. This means each embedding vector for a hyperedge q encodes information about its adjacent hyperedges, not just the row vertices it accesses. We found that using the LSTM aggregator in GraphSAGE works well in our evaluation. Finally, GraphSAGE requires each vertex in the graph to be initialized with a feature vector. We use the static and dynamic components of the environment (§4.1) to create the initial feature vector for each vertex in the bipartite graph. To simplify the implementation, we use the same feature representation for both query and row vertices.

*Scoring Layer.* This layer takes as input the embedding vector produced by the embedding layer, as well as the partition state vector (§4.1) and produces a single real number representing a "score" for the input hyepredge q. We use a dense fully-connected neural network (with 2 hidden layers) for the scoring layer for its simplicity and efficiency. Note that our network evaluates each hyperedge separately to produce a score without taking the other hyperedges in the candidates set as input. The intuition behind this design decision is that the vectors produced by the embedding layer already encode information about adjacent hyperedges (which are often going to be candidates themselves). Nevertheless, a more sophisticated architecture able look at all the candidate hyperedges simultaneously (such as LSTM [21] or Transformer [45]) might perform better for other more complex workloads, which we leave for future work.

*Softmax Layer.* We use a standard Softmax [17] function to map the scores to probabilities.

An action is then sampled based on the probabilities computed for each candidate. The neural network representing $\pi_\theta$(s, q) is trained end-to-end using the training procedure we describe in the following section.

## 5.2 Training Procedure

We use a fairly standard training procedure, similar to prior work [27]. The training proceeds in *iterations* until convergence (which can be set as a fixed number of iterations, or until partitioning quality reaches a user-defined threshold). Each iteration consists of running N *episodes*. An episode consists of completely partitioning the hypergraph using Algorithm 2 with the current value of $\theta$, and recording the state, action and reward for each time step in the episode. After completing all episodes, we use this information to apply the modified REINFORCE equation to compute the gradient (as explained in §3.1), and take one step in its direction at the end of the iteration. The pseudocode for the training procedure is presented in 3. Note that for simplicity, we assume all episodes take the same number of time-steps L.

**for** *each iteration* **do**
  $\Delta\theta \leftarrow 0$;
  **for** *episode i := 1 ... N* **do**
    Fully partition the hypergraph and record $\{s_1^i, a_1^i, r_1^i, ...s_L^i, a_L^i, r_L^i\}$;
    **for** *t := 1 ... L* **do**
      `// Compute cumulative reward from t`
      `// onwards`
      $v_t^i \leftarrow \sum_{s=t}^{L} r_s^i$
    **end**
  **end**
  **for** *t := 1 ... L* **do**
    $b_t \leftarrow \frac{1}{N} \sum_{i=1}^{N} v_t^i$ `// Compute baseline`
    **for** *i := 1 to N* **do**
      $\Delta\theta \leftarrow \Delta\theta + \alpha \bigtriangledown_\theta log\pi_\theta(s_t^i, a_t^i)(v_t^i - b_t)$
    **end**
  **end**
  $\theta \leftarrow \theta + \Delta\theta$
**end**

**Algorithm 3:** Training Procedure

Training is an expensive process, but it only needs to be done offline and with less frequency than the resharding process. Theoretically, the agent can be trained only once and never be updated

after that, but retraining regularly has the benefit of being able to adapt to workload changes over time. Neuroshard is primarily designed to be trained on a single workload so that it can discover and exploit workload specific properties, but another possibility is to train on traces collected from different workloads simultaneously, with the purpose of creating a robust, general-purpose sharding agent and avoid overfitting. We think this could be an interesting direction to explore, but we leave that for future work.

## 6 LEARNING WITH MULTIPLE OBJECTIVES

In the previous sections we showed how to model each of the fanout minimization and the load balancing problems as an RL problem individually. While this is potentially useful on its own, a key feature of Neuroshard is the ability to optimize for both simultaneously. In this section we describe how we accomplish this, as well as how we augmented Neuroshard with general support for multi-objective training so that additional objectives can be added as long as a suitable reward signals can be defined for them.

*Architecture.* We use the same policy network (Figure 3) when training with multiple objectives. This requires the embedding layer to extract useful features relevant for optimizing all objectives. We considered more complex schemes using multiple embedding layers but we found that this adds complexity to training, in terms of implementation and speed, without much benefit in our evaluation.

*Rewards.* We started with the following straightforward approach: Define the reward $r_t$ as the sum of the individual objective rewards, *i.e.,* at each time-step t, we add the rewards of fan-out minimization and load balancing together. This did not work well in practice for various reasons. First, the units of the reward components are very different, which makes it easy for one of the objectives to completely dominate the training. We attempted to normalize the units of the rewards (*e.g.,* by using a weighted sum of reward components instead of just giving each component an equal weight), but were also not very successful: It was brittle, requires manual tuning and experimentation for each workload and would be very hard to generalize. We also found that certain hyper-parameters (*e.g.,* the learning rate) that work well for one task may not work well for the other. Nevertheless, we gained a valuable insight from this approach: By making the overall reward a linear combination of the individual component rewards, the overall objective function, or loss function in machine learning parlance, is a linear combination of the individual component loss functions, *i.e.,*

$$L = \sum_i w_i L_i$$

This allows casting the multi-objective Neuroshard training as a multi-task learning problem, and leverage existing techniques for that problem. Multi-Task learning techniques are popular in computer vision, but to the best of our knowledge this is the first time this is applied to a multi-objective combinatorial optimization problem like hypergraph partitioning. We use GradNorm [10] in Neuroshard. It is designed for problems where the overall loss is a linear combination of the individual task loss. GradNorm works to dynamically tune the gradient magnitudes so that gradients for different tasks are placed on a common scale and dynamically adjust gradient norms to ensure that the training for different tasks progresses at similar rates, instead of using a hyperparameter to combine the objectives that remains fixed across training iterations. To apply GradNorm in Neuroshard we make the following choices:

- **W**. This is the subset of network weights to apply GradNorm to. In our case since all weights are shared among both tasks we just set W to all the weights in the policy network.
- $\alpha$. This is the "asymmetry" hyper-parameter. It is tunable and might benefit from hyper-parameter search, but intuitively the value of $\alpha$ should be high when the tasks are dissimilar to each other which is generally the case in our setting.

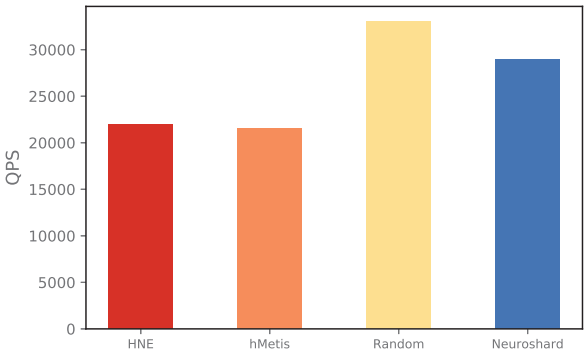



**Figure 4: micro-benchmark throughput with 4 shards**

## 7 PRELIMINARY EVALUATION

We use a shared-nothing distributed database where each node is a small Google Cloud VM of type n1-standard-1 (1 vCPU, 3.75 GB memory) running a MariaDB [1] instance. The instances are not sharding-aware, we just load the subset of rows assigned to each instance at the beginning of each run. The clients running the queries know the mapping of rows to shards, and they route the queries to each node as appropriate, using the standard XA API [6] to co-ordinate the transaction if the query is distributed. To calculate throughput, we first run a workload for 1 minute warm up, then measure the average queries per second (QPS) for a period of 5 minutes after that.

### 7.1 Baselines

For the rest of the section we compare the performance of Neuroshard's sharding with the following baseline approaches:

- **hMetis** [23]. A popular heuristic-based hypergraph partitioner. We use the standard flags by using the **shmetis** executable. For fairness, we set the value of balance flag (UBfactor) to the tightest possible. This is a baseline that is optimized for the fanout objective. Note that while hMetis supports specifying vertex weights, using this option would not guarantee that each partition has roughly the same number of vertices which does not satisfy our problem statement.
- **HNE**. We also use Algorithm 1 as a fanout optimizing baseline.

- **Random**. This scheme chooses a shard uniformly at random for each row. It is an idealization of hash partitioning, and is optimized for load balancing.

## 7.2 Multi-objective Microbenchmark

To study the versatility of Neuroshard and the importance of optimizing for multiple objectives simultaneously, we created a simple but illustrative synthetic workload.

The database consists of 400 rows, where each row belongs to exactly one of the following sets:

- **Parent Rows**. 4 rows
- **Child Rows**. These are 96 rows in total. Each parent row is associated with 24 child rows.
- **The rest**. 300 rows that get accessed independently

The workload is made up of the following (read-only) queries:

- 1% of the queries read a subset of 2 or more of the parent rows.
- 49% read one parent row, and 0 or 1 of its child rows.
- 50% are point queries that read one of the 300 other rows.

Our goal in this experiment is to shard the database into four shards. Note that if we only cared about fanout minimization, then the perfect solution would be to put all of the Parent and Child rows in a single shard, and distribute the rest equally across the 3 remaining shards. This, however, would mean that 50% of the queries go to a single shard, creating a large load imbalance.

We shard the database into four shards. We generate a trace of 5000 queries and use it for training. We then generate another trace made of 5000 queries that follow the same distribution, and shard it using different algorithms, then measure the QPS by generating a workload that follows the same distribution. The results are shown in Figure 4.

**Analysis.** HNE and hMetis compute the sharding that minimizes fanout perfectly by putting all of the Parent and Child rows in a single shard. This results in a large load imbalance as we mentioned previously, causing them to have the worst throughput on this microbenchmark. We find that Neuroshard usually computes a sharding that assigns the parent rows to two shards; two rows each. It almost perfectly co-locates the child rows with their parents, leading to close to zero distributed queries. This better load-balancing of parent rows is the reason why the QPS using Neuroshard is significantly better than HNE or hMetis. Using random hash partitioning on this workload actually has a lot of variance in terms of the overall number of rows assigned to each shard, as well as the distribution of parent/child rows among the shards. To reduce the noise we started by assigning one parent row to each shard then applying random partitioning to the rest of the rows. The resulting sharding scheme has the best load balancing in this workload which leads to it having the best QPS despite having a significant share of distributed queries; since this is a read-only workload, the cost of distribution is not quite high. Even though Neuroshard's QPS is not the highest in this workload, these results demonstrate the robustness of Neuroshard compared to single-objective methods.

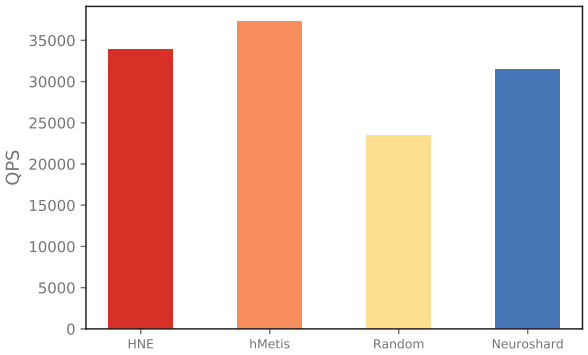



**Figure 5: Epinions workload throughput with 5 shards**

## 7.3 Epinions

To evaluate Neuroshard's scalability and ability to handle real-world datasets, we use a workload based on Epinions [30], a real-world social network dataset that is a popular choice for evaluating sharding schemes because its many-to-many relationships make it hard to shard [12]. The database is composed of two tables: **reviews** and **trust**. Each row in the reviews table has the schema *<user, item, rating>* and represents a review of an item by a user. Each row in the trust table has the schema *<user_a, user_b>* and records the fact that user_a trusts reviews by user_b. We shard the reviews table by the trusting user column (*i.e.,* user_a), and we shard the items table by the item column. We shard the database on 5 MariaDB servers.

*Workload.* There is no available trace of queries from the epinions website, so we designed a workload based on the websiteâĂŹs most popular functionality: users that view an item and want to retrieve a list of reviews for that item by the users they trust. Thus, each application-level query is represented by a pair *<user, item>*. Our workload generator can generate these application-level queries based on different distributions. We chose one such distribution in this section for illustration (we got similar results with other distributions): Items are sorted by popularity (*i.e.,* number of reviews), and the top 20% of items are considered *hot*, while the rest is considered *cold*. The workload generator generates a query as follows: First it selects an item **i**, with i being hot with probability 80% and cold with probability 20%. Then, a user **u** is selected uniformly at random from the set of users trusted by iâĂŹs reviewers. Recall that we implement query routing in the client. If both the i and u are colocated on the same server only one query needs to be sent by the client. Otherwise, the client needs to send queries to two servers. Hence, sharding has a big impact on throughput (up to 50% reduction in SQL queries), and potentially also latency.

*Generating training and test traces.* We generate two traces by the workload generator. The first is a training trace that is made up of 4000 queries, and is used to train Neuroshard. We then generate a partitioning trace of 100,000 queries. This is used by Neuroshard and the other baseline techniques to partition the database. We

then measure the overall system QPS given by each technique by sending queries based on the workload. The results are in Figure 5.

***Analysis.*** Partitioning quality with respect to fanout minimization is the dominant factor in this workload, which is why it is unsurprising that single-objective hMetis and HNE have the best performance, while Random which completely ignores fanout minimization does the worst. Neuroshard appears to suffer a bit due to its attempts at jointly optimizing for load balancing, but its performance is still comparable to HNE. This shows that Neuroshard is robust to changes in the workload unlike single-objective approaches such as Random. Furthermore, we used a small trace for Neuroshard training, and yet it was still able to learn heuristics that were effective on the much larger test trace. This is very important, as it would be unfeasible to use very long traces to train Neuroshard; not only would that be computationally very expensive, but excessively long time horizons are known to diminish the effectiveness of RL algorithms [41, 46].

## 8 RELATED WORK

**Hypergraph partitioning.** As an NP-Hard problem with many practical applications, many popular heuristic solvers for hypergraph partitioning have been developed such as hMetis[23] and Zoltan [14]. More recent work such as SHP [22] and HYPE [32] has focused on scalability and the ability to partition large hypergraphs. All these tools only optimize for fanout minimization (or the closely related cut-size). HYPE is also a generalization of Neighbor Expansion [47] but takes a vertex-centric approach compared to the hyperedge centric approach that we take. We adopt some of the performance optimizations they suggested such as limiting the size of the candidate set in our implementation.

**Sharding as (hyper)graph partitioning.** Schism [12] pioneered modelling database sharding and replication as a graph partitioning problem. SWORD [36] builds up on this approach but uses coarser granularity and applies hypergraph partitioning instead. They use heuristic solvers like Metis [24] and hMetis [23] which optimize for a single objective, unlike Neuroshard, but they also model replication which Neuroshard currently does not.

**Online sharding.** Clay [38], E-Store [43] are online heuristic sharding algorithms that actively monitor the system to identify hot tuples, and then potentially migrate them (along with other jointly accessed row to a different server). However, these systems make some assumptions about the workload: Clay primarily target in-memory workloads with a high skew in load, E-Store assumes all tables form a tree-schema based on foreign key relationships. In contrast, Neuroshard aims to be broadly applicable to a variety of workloads and environments. It would be interesting to investigate how a similar RL approach can improve online sharding, which we leave for future work.

**Automated database tuning.** There is a long and rich history of research [8] that aims to automatically tune various aspects of a database *configuration* to improve performance on a given workload, such as by selecting indexes to speed-up access, recommending a physical layout or generally select values for various configuration knobs available in the DBMS. More recently, machine learning [44] and specifically, deep RL (*e.g.,* [25, 48]) techniques have been applied to this problem.

**Deep learning for combinatorial optimization.** A large recent body of work (e.g. [2, 4, 5, 26]) has explored using deep learning for solving NP-Hard combinatorial problems in general, and on graph problems in particular. Dai et. al. [13] proposed a general framework for solving optimization problems over graphs using RL. Their approach uses deep-Q-learning and requires learning a representation for the entire graph, as opposed to the policy methods we use that needs only to learn a localized policy. To the best of our knowledge, we are the first to explore using deep RL for multi-objective hypergraph partitioning.

**Deep RL for systems.** Deep RL has been applied successfully to a wide range of problems from video streaming [28] and network traffic scheduing [9, 11] to compute resource management [27] and even physical device placement [33]. We were inspired by DeepRM [27] and Decima [29], which apply Deep RL to the resource management and job scheduling domains. Both use policy methods, and Decima also uses GNNs to produce state representation. Unlike Neuroshard, the graphs in Decima's state are comparatively small, and while Decima and DeepRM support different objectives, users would have to select just one to use in training. Both systems do not optimize for multiple objectives simultaneously.

**Deep RL for Partitioning**. Recent work (e.g., [7, 16, 20, 50]) has explored using Deep RL methods for partitioning. GridFormation [16] proposes an online partitioning approach using Deep-Q-learning, which is applied to vertical partitioning in [7]. In contrast, Neuroshard is an offline approach for horizontal sharding. Hilprecht et. al. [20] use deep-Q-learning to build a general-purpose partitioning advisor that can recommend a table partitioning scheme for a new workload. It targets analytical workloads in which each table can be partitioned on one or more columns, while Neuroshard's focus is OLTP workloads and supports fine-grained row to shard assignments. The state and action spaces in this setting are much smaller than in Neuroshard, but evaluating agent's solutions and computing the rewards is much more computationally challenging.

## 9 CONCLUSIONS

Neuroshard is the first system that uses a learned approach for directly learning horizontal sharding assignments. It models past accesses to rows as a hypergraph, and then applies deep RL to generate an efficient partitioner that works by incrementally scoring a relatively small set of candidates derived from the rows accessed together as part of the same queries. It uses GradNorm, a multi-task learning technique, to automatically tune the gradients of multiple different task losses, so that it can balance between multiple different objectives simultaneously.

This work takes the first steps in realizing a learned sharding system, but leaves several open problems for future work. First, we plan to incorporate more complex objectives and constraints, including different machine capacities and network topologies. Second, for large and complex production traces, we expect that we will need to devise sampling techniques to keep the episode length from being too large, while allowing Neuroshard to learn effective local heuristics. Third, while building one partition at a time is simple and efficient, we suspect extending Neuroshard to build more than one simultaneously would allow it exploit power of two choices more effectively. Finally, we do not address how to transition to a

new partitioning scheme when one is computed, nor do we attempt to minimize data movement when computing a new partitioning. These are important practical considerations, which we leave for future work.

## ACKNOWLEDGMENTS

We thank Kexin Pei and Chengzhi Mao for many useful discussions on Deep Learning and introducing us to GradNorm. We also thank Haonan Wang for assisting with experimental evaluation. This project was supported in part by NSF CNS 2106530 and CNS 1564055; ARO award W911NF-21-1-0078; ONR N00014-17-1-2788; DiDi Faculty Research Award; J.P. Morgan Faculty Research Award; and Accenture Research Award.

## REFERENCES

[1] 2021. MariaDB Server: The open source relational database. https://mariadb.org/.
[2] Kenshin Abe, Zijian Xu, Issei Sato, and Masashi Sugiyama. 2019. Solving NP-Hard Problems on Graphs by Reinforcement Learning without Domain Knowledge. *arXiv preprint arXiv:1905.11623* (2019).
[3] Konstantin Andreev and Harald Racke. 2006. Balanced graph partitioning. *Theory of Computing Systems* 39, 6 (2006), 929–939.
[4] Thomas D Barrett, William R Clements, Jakob N Foerster, and AI Lvovsky. 2019. Exploratory Combinatorial Optimization with Reinforcement Learning. *arXiv preprint arXiv:1909.04063* (2019).
[5] Irwan Bello, Hieu Pham, Quoc Le, Mohammad Norouzi, and Samy Bengio. 2017. Neural Combinatorial Optimization with Reinforcement Learning. In *ICLR (Workshop)*.
[6] Philip A. Bernstein and Eric Newcomer. 2009. *Principles of transaction processing*. Morgan Kaufmann. 330–x336 pages.
[7] Gabriel Campero Durand, Rufat Piriyev, Marcus Pinnecke, David Broneske, Balasubramanian Gurumurthy, and Gunter Saake. 2019. Automated Vertical Partitioning with Deep Reinforcement Learning. In *New Trends in Databases and Information Systems*. 126–134.
[8] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. 3âĂŞ14.
[9] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. 2018. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 191–205.
[10] Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich. 2018. GradNorm: Gradient Normalization for Adaptive Loss Balancing in Deep Multitask Networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Vol. 80. 794–803.
[11] Sandeep Chinchali, Pan Hu, Tianshu Chu, Manu Sharma, Manu Bansal, Rakesh Misra, Marco Pavone, and Sachin Katti. 2018. Cellular network traffic scheduling with deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*.
[12] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. (2010).
[13] Hanjun Dai, Elias Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*. 6348–6358.
[14] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. 2002. Zoltan data management services for parallel dynamic applications. *Computing in Science & Engineering* 4, 2 (2002), 90–96.
[15] David J DeWitt, Shahram Ghandeharizadeh, Donovan A Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. 1990. The Gamma database machine project. (1990).
[16] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Broneske, Gunter Saake, Maya S. Sekeran, Fabián Rodriguez, and Laxmi Balami. 2018. GridFormation: Towards Self-Driven Online Data Partitioning Using Reinforcement Learning. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM'18)*.
[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.
[18] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems*. 1024–1034.
[19] Masatoshi Hanai, Toyotaro Suzumura, Wen Jun Tan, Elvis Liu, Georgios Theodoropoulos, and Wentong Cai. 2019. Distributed Edge Partitioning for Trillion-Edge Graphs. *Proc. VLDB Endow.* (Sept. 2019), 2379âĂŞ2392.
[20] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 143âĂŞ157.
[21] Sepp Hochreiter and JÃijrgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* (1997).
[22] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Alessandro Presta, and Yaroslav Akhremtsev. 2017. Social hash partitioner: a scalable distributed hypergraph partitioner. *arXiv preprint arXiv:1707.06665* (2017).
[23] George Karypis. 1998. hMETIS 1.5: A hypergraph partitioning package. *Technical Report* (1998).
[24] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).
[25] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* (Aug. 2019), 2118âĂŞ2130.
[26] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. 2018. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*. 539–548.
[27] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*. 50–56.
[28] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 197–210.
[29] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. 270–288.
[30] Paolo Massa and Paolo Avesani. 2005. Controversial users demand local trust metrics: an experimental study on epinions.com community. *AAAIâĂŹ05* (2005).
[31] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-tree database storage engine serving Facebook's social graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.
[32] Christian Mayer, Ruben Mayer, Sukanya Bhowmik, Lukas Epple, and Kurt Rothermel. 2018. Hype: Massive hypergraph partitioning with neighborhood expansion. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 458–467.
[33] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*. PMLR, 2430–2439.
[34] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518 (2015), 529–533.
[35] George E. Monahan. 1982. State of the art - a survey of partially observable markov decision processes: theory, models, and algorithms. *Management Science* (1982).
[36] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. 2013. SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) *(EDBT '13)*. 430âĂŞ441.
[37] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust Region Policy Optimization. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Vol. 37. 1889–1897.
[38] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.
[39] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. 2013. F1: A Distributed SQL Database That Scales. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1068âĂŞ1079. https://doi.org/10.14778/2536222.2536232
[40] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (Jan. 2016), 484–489.
[41] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning*. MIT Press.

[42] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*. 1057âĂŞ1063.

[43] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.

[44] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1009âĂŞ1024.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Uszkoreit Jakob, Llion Jones, Aidan N. Gomez, ÅĄukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in neural information processing systems*.

[46] Haonan Wang, Hao He, Mohammad Alizadeh, and Hongzi Mao. 2019. Learning Caching Policies with Subsampling. In *NeurIPS Machine Learning for Systems Workshop*.

[47] Chenzi Zhang, Fan Wei, Qin Liu, Zhihao Gavin Tang, and Zhenguo Li. 2017. Graph edge partitioning via neighborhood heuristic. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 605–614.

[48] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 415âĂŞ432.

[49] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81.

[50] Jia Zou, Amitabh Das, Pratik Barhate, Arun Iyengar, Binhang Yuan, Dimitrije Jankov, and Chris Jermaine. 2021. Lachesis: Automated Partitioning for UDF-Centric Analytics. *Proc. VLDB Endow.* 14, 8 (2021), 1262–1275.