

LIFT: Reinforcement Learning in Computer Systems by Learning From Demonstrations

Michael Schaarschmidt
University of Cambridge
michael.schaarschmidt@cl.cam.ac.uk

Alexander Kuhnle
University of Cambridge
alexander.kuhnle@cl.cam.ac.uk

Ben Ellis
University of Cambridge
be255@cam.ac.uk

Kai Fricke
Helmut Schmidt University
fricke@hsu-hh.de

Felix Gessert
Baqend
fg@baqend.com

Eiko Yoneki
University of Cambridge
eiko.yoneki@cl.cam.ac.uk

ABSTRACT

Reinforcement learning approaches have long appealed to the data management community due to their ability to learn to control dynamic behavior from raw system performance. Recent successes in combining deep neural networks with reinforcement learning have sparked significant new interest in this domain. However, practical solutions remain elusive due to large training data requirements, algorithmic instability, and lack of standard tools.

In this work, we introduce LIFT, an end-to-end software stack for applying deep reinforcement learning to data management tasks. While prior work has frequently explored applications in simulations, LIFT centers on utilizing human expertise to learn from demonstrations, thus lowering online training times. We further introduce TensorForce, a TensorFlow library for applied deep reinforcement learning exposing a unified declarative interface to common RL algorithms, thus providing a backend to LIFT. We demonstrate the utility of LIFT in two case studies in database compound indexing and resource management in stream processing. Results show LIFT controllers initialized from demonstrations can outperform human baselines and heuristics across latency metrics and space usage by up to 70%.

KEYWORDS

ACM proceedings, L^AT_EX, text tagging

1 INTRODUCTION

Model-free reinforcement learning (RL) techniques offer a generic framework for optimizing decision making from raw feedback signals such as system performance [67], thus not requiring an analytical model of the system. In recent years, deep reinforcement learning (DRL) approaches which combine RL with deep neural networks have enjoyed successes in a variety of domains such as games (Go [66], Atari [16, 49, 50, 74]), and applied domains such as industrial process control [25] or robotic manipulation [71]. RL

approaches have also long appealed to computer systems researchers, with experimental applications in domains such as adaptive routing or server resource management spanning back over 20 years [35, 36, 68, 69]. The advent of deep RL in combination with widely available deep learning frameworks has renewed interest in this approach. More recent examples include automated TensorFlow device placements [46, 47], client-side bit-rate selection for video streaming [43], and simplified cluster scheduling [42].

However, practical RL deployments in computer systems and data management remain difficult due to large training data requirements and expensive decision evaluations (e.g. multiple minutes to deploy a cluster configuration). RL algorithms also suffer from inferior predictability and stability compared to simpler heuristics [26, 41]. Consequently, proof-of-concept successes in simplified and highly controlled simulations have infrequently lead to practical deployments. Nonetheless, DRL remains appealing as it combines the ability of deep neural networks to identify and combine features in unforeseen ways with learning from raw system feedback. The long-term aim is to automate manual feature and algorithm design in computer systems and potentially learn complex behaviour outperforming manual designs.

In this work, we explore these limitations by outlining a software stack for practical DRL, with focus on guiding learning via existing log data or demonstrated examples. The key idea of our paper is that in modern data processing engines, fine-granular log data can be used to extract demonstrations of desired dynamic configurations. Such demonstrations can be used to pretrain a control model, which is subsequently refined when deployed in its concrete application context. To this end, we make the following contributions:

We present *LIFT* (§4), a high level framework for LearnIng From Traces which provides common components to interface and map between systems and reinforcement learning, thus removing boilerplate code. We further introduce TensorForce, a highly modularized DRL library focusing on a declarative API for common algorithms, which serves as an

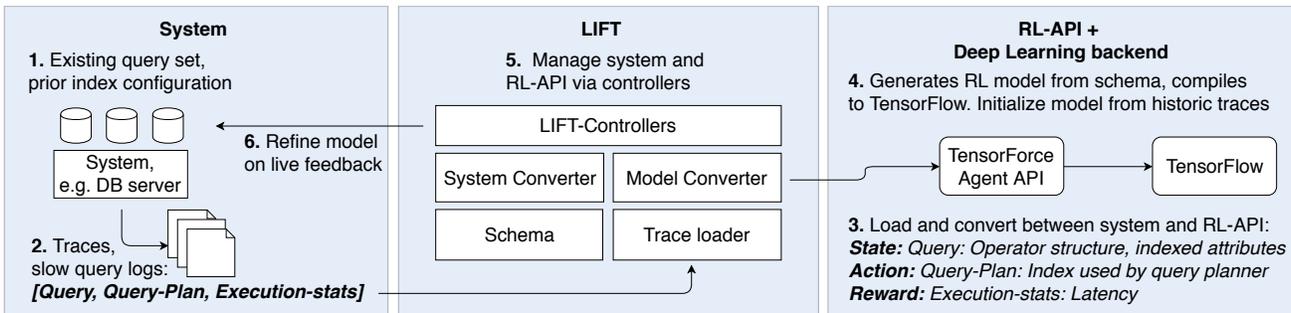


Figure 1: LIFT workflow.

algorithmic backend for LIFT. LIFT allows users to specify data layouts of states and action spaces which are used by TensorFlow to generate TensorFlow models for executing RL tasks (§4.2). In the evaluation (§6), we demonstrate the utility of our LIFT prototype in two experimental data management case studies. First, we use LIFT to generate a controller for automatic compound database indexing (§5). Indexing is an attractive use case for RL as the optimal index set for an application depends on the complex interaction of workload, query operators within each query, data distribution, and query planner heuristics. While analytical solutions are difficult to build and vary per database and query planner, rich feedback from slow query logs enables RL controllers to identify effective solutions. Experimental results show that a LIFT-controller pretrained from imperfect rule-based demonstrations can be refined within few hours to outperform various rule and expert baselines by up to 70%. We also use LIFT to learn task parallelism configurations on Heron [34], a state of the art stream processing engine.

Figure 1 illustrates LIFT’s role and components. The slow query log from a database containing queries, the executed query plan, and execution statistics are read into LIFT. Via a user-defined schema and converter, LIFT interprets traces and/or provided rules as demonstrations to train an offline model. In the indexing case study, this is achieved by mapping query shape and existing indices to a state, the command required to create the index used to an action, and query performance to a reward. Traces must hence contain not only runtime performance but also corresponding configurations which can be used to reconstruct a command (action) leading to that configuration. For example, the slow query log may contain the query plan including index used, and this can be converted to the command creating that index. Schema layouts are passed to TensorFlow to generate a corresponding TensorFlow graph. The states, actions, and rewards are then used to train a controller model to adopt the strategy (e.g. hand-designed rule or expert decision) behind prior indexing. Finally, LIFT is deployed in online mode to either refine indexing on an existing query set, or within a new application to replace manual tuning.

2 BACKGROUND

We give a brief introduction to RL with focus on practical concerns. RL is not a single optimization strategy but a class of methods used to solve the reinforcement learning problem. Informally, RL is utilized when no supervised feedback for a decision is available but reward signals indicating relative performance. For example, a cluster scheduler allocating tasks to resources may receive feedback from task completion times, but not whether a scheduling decision was optimal.

We consider the classic formulation wherein an agent interacts with an environment ϵ described by states $s \in \mathcal{S}$ and aims to learn a policy π that governs which action $a \in \mathcal{A}$ to take in each state [67]. At each discrete time step t , the agent takes an action a_t according to its policy $\pi(a|s)$, transitions into a new state s_{t+1} according to the environment dynamics, and observes a reward r_t from a reward function $R(s, a)$. The goal of the agent is to maximize cumulative expected rewards $R = \mathbb{E}[\sum_t \gamma^t r_t]$, where future rewards are discounted by γ . State transitions and rewards are often assumed to be stochastic, and to satisfy the Markov property so each transition only depends on the prior state s_{t-1} .

In data management tasks, the state is typically represented as a combination of the current workload and configuration, embedded into a continuous vector space. To deal with the resulting large state spaces and generalize from seen to unseen states, RL is used in conjunction with *value function approximators* such as neural networks where the expected cumulative return from taking an action a in state s is estimated by a function parametrized by trainable parameters θ (i.e. the neural network weights). Formally, the action-value function Q^π is given as

$$Q^\pi(s, a; \theta) = \mathbb{E}[R_t | s_t = s, a]. \quad (1)$$

The goal of learning is to determine the optimal $Q^*(s, a)$ which maximizes expected returns. Concretely, when using Q-learning based algorithms, the neural network produces in its final layer one output per action representing its Q-value. The resulting policy is implicitly derived by greedily selecting the action with the highest Q-value while occasionally

selecting random actions for exploration. Updates are performed by performing iteratively (over a sequence indexed by i) gradient descent on the loss $J(\theta)_i$ [50]:

$$J_i(\theta)_i = \mathbb{E}_{s, a \sim \pi} [(y_i - Q(s, a; \theta_i))^2] \quad (2)$$

with $y = R(s, a) + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$. Intuitively, this loss is the (squared) difference between the observed reward when taking a in s plus the discounted estimate of future returns from the new state s' , and the current estimate of $Q(s, a; \theta)$, or in other words how much the Q-function has to be modified to account for observing a new reward.

In Deep Q-learning as introduced by Mnih et al. [50], experience tuples of the form (s_t, a_t, r_t, s_{t+1}) are collected and inserted into a replay buffer, and updates are performed by sampling random batches to compute gradients. Further, learning is stabilized by using a separate target network to evaluate the Q-target y , which is only synchronized with the training network with delay. In contrast, policy gradient (PG) methods directly update a parametrized policy function $\pi(a|s; \theta)$ such as a Gaussian or categorical distribution. This is typically (e.g. in the classical REINFORCE algorithm [76]) achieved by obtaining a sample estimates of current policy performance and updating θ in the direction $\nabla_{\theta} \log \pi(a_t|s_t; \theta)(R_t - b_t(s_t))$. Detailed surveys of contemporary work are given by Li and Arulkumaran et al. [6, 37].

RL approaches remain attractive due to their theoretical value proposition to learn from raw feedback. However, despite over two decades of research on RL in computer systems, practical applications remain difficult to realize due to various limitations. In the following, we discuss concrete issues before introducing LIFT.

3 PRACTICAL ISSUES

RL algorithms are known to suffer from various limitations which we highlight here in the context of data management. **Training data requirements.** First, RL methods are notoriously sample-inefficient and solving common benchmark tasks (e.g. Atari) in simulators can require up to 10^7 - 10^9 problem interactions (states) when using recent approaches [16]. In data management experiments, performing a single step (e.g. a scheduling decision) and observing its impact may take between seconds and hours (e.g. deciding on resources for a job and evaluating its runtime). Consequently, training through online interaction can be impractical for some tasks, and training in production systems is further undesirable as initial behavior is random to explore. A common strategy to accelerate training is to train RL agents in simulation [42, 43]. This approach enables researchers to explore proof-of-concept experiments but also introduces the risk of making unrealistic assumptions and oversimplifying the problem domain, thus making successful simulation-to-real

transfer unlikely. Some research domains have access to verified simulators (e.g. network protocols) but this is not the case for many ad-hoc problems in data management.

Another common approach is to execute online training on a staging environment or a smaller deployment of the system. For example, in their recent work on hierarchical device placement in TensorFlow [46], Mirhoseini et al. report that training their placement mechanism on a small scale deployment for 12.5 GPU-hours saves 265 GPU hours in subsequent training of a neural network. Here, RL was used as a direct search mechanism where the aim of training is to identify a single final configuration which is not modified later. Successful online training is further difficult if the goal of the controller is to react to unpredictable and sudden workload changes. This is because training workloads may not sufficiently cover the state space to generalize to drastic workload changes (while exploring the state space is usually possible in simulation).

Hyper-parameters and algorithmic stability. DRL algorithms require more configuration and hyper-parameter tuning than other machine learning approaches, as users need to tune neural network hyper-parameters, design of states/actions and rewards, and parameters of the reinforcement learning algorithm itself. A growing body of work in DRL attempts to address algorithmic limitations by more efficiently re-using training data, reducing variance of gradient estimates, and parallelizing training (especially in simulations) [16, 23, 61, 62]. Some of these efforts have recently received scrutiny as they have been shown difficult to reproduce [26, 41], often due to the introduction of various additional hyper-parameters which again need to be tuned. This is complicated by the fact that RL algorithms are often evaluated on the task they were trained on (i.e. testing performance on the game the algorithm was trained on). RL is effectively used for optimization on a single task, and, as Mania et al. argue [41], some algorithmic improvements in recent work may stem from overfitting rather than fundamental improvements.

Software tools. The reproducibility issues of RL algorithms are further exacerbated by a lack of standard tools. The practical successes of neural networks in diverse domains have led to the existence of widely adopted deep learning frameworks such as Google’s TensorFlow [2], Microsoft’s CNTK [63], or Apache MXnet [12]. These libraries provide common operators for implementing and executing machine learning algorithms while also omitting the complexity of directly interfacing hardware accelerators (e.g. GPUs, FPGAs, ASICs). However, RL algorithms cannot be used with similar ease as existing research code bases primarily focus on simulation environments, and thus require significant modifications to be used in practical applications. We introduce our RL library built on top of TensorFlow in section §4.2.

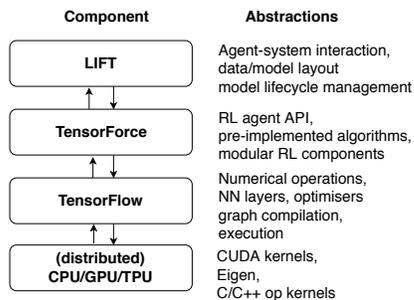


Figure 2: LIFT stack for applied RL.

The issues above continue to present significant obstacles in using RL. We investigate means to improve data efficiency and tooling by providing a software stack for deep RL focused on initializing controllers from pre-existing knowledge.

4 LIFT

4.1 System overview

We begin by giving a high level overview of our framework before discussing each component in detail. Generally, we distinguish between our algorithmic backend *TensorForce*, and *LIFT*, a collection of services which allow RL controllers to be deployed in different execution contexts, which we explain below (Figure 2). Frameworks such as TensorFlow [1] expose an API primarily on the abstraction level of numerical operators with an increasing number of modules containing neural network layers, optimizers, probability distributions, data set tools etc. However, currently no such modules exist within TensorFlow to expose RL functionality via similar APIs. *TensorForce* fills this gap by providing a unified API to a set of standard RL algorithms on top of TensorFlow.

The main abstractions *LIFT* operates on are RL models and system models. A model maps between RL agent output to system actions (e.g. configuration changes), or from system metrics to RL agent (e.g. parsing log entries to states, actions and rewards). *LIFT*'s primary purpose is to facilitate RL usage in new systems by providing commonly used functionality pertaining to model serialization and evaluation, and further by defining system data layout and automatically mapping them to the respective TensorFlow inputs and outputs. *LIFT* uses *TensorForce* as its backend in our example implementation but is independent of both *TensorForce* and TensorFlow, as to be able to use any RL implementation providing a minimal common API. In the following, we discuss the design of *TensorForce*.

4.2 TensorForce

Deep reinforcement learning is a rapidly evolving field and few standards exist with regard to usage outside controlled simulations. Various open source libraries such as OpenAI

baselines [65], Nervana coach [10], or Ray Rllib [38] exist. They are tightly coupled with simulation environments such as OpenAI gym [9] which provide unified interfaces to tasks for evaluating and comparing algorithms. In our experiments, we have found these research frameworks to be difficult to deploy in practical use cases for two additional reasons.

First, open source reinforcement learning libraries frequently rely on fixed neural network architectures. For example, the code we analyzed typically created network output layers for actions based on descriptors provided by simulations only supporting restricted actions (e.g. only either discrete or continuous actions per step, but not both). Substantial code modifications are required to support multiple separate types of actions (tasks) per step. This is because the purpose of these reference implementations is primarily to reproduce research results on a particular set of benchmark tasks, as opposed to providing configurable, generic models. Second, as discussed in §3, recent RL methods incorporate various optimization heuristics to help training efficiency and stability, thus increasing the number of tunable parameters. We found existing code bases to attempt reducing complexity by hard-coding heuristics of which users may be unaware. For example, one of the implementations we surveyed internally smoothes state vectors via an exponentially moving average, and clips reward values without documenting or exposing this feature. We hence introduce *TensorForce*, a general purpose DRL library which exposes a well-defined declarative interface to creating and transparently configuring state-of-the-art algorithms.

Design. Our aim is to give a unified interface to specify a decision model by describing its inputs and outputs without any restriction on the number and type of different inputs (states) or outputs (actions). Further, the specification contains the model to construct, network layers to use, and various further options to be applied such as exploration, input preprocessing (e.g. normalization or down-sampling) and output post-processing (e.g. noise), and algorithm-specific options such as memory size.

TensorForce is built on two principles: First, users should not be required to modify any library code to express their problem dynamics, as is often the case in current open source code, thus necessitating expressive configurations. Second, reinforcement learning use cases may drastically differ in design, e.g. environments may present continuous learning or episodic problems, algorithms may use memories to incorporate old experiences, or just learn from new observations. However, most of this arising complexity can be deterministically (depending on the model selected) handled internally. Consequently, we provide a unified API for all model and agent variants with just two methods at its core, one to request new actions for given states, one to observe rewards

```

from tensorforce.agents import PPOAgent
# Create a Proximal Policy Optimization agent
agent = PPOAgent(
    states=dict(type='float', shape=(10,)),
    actions=dict(
        discrete_action=dict(type='int', num_actions=10),
        binary_action=dict(type='bool')
    ),
    network=[
        dict(type='dense', size=64),
        dict(type='dense', size=64)
    ],
    step_optimizer=dict(
        type='adam',
        learning_rate=1e-4
    ),
    execution=dict(type='single'),
    states_preprocessing=[dict(type='running_standardize')]
)
// Connect to a client
client = DBClient(host='localhost', port=8080)
while True:
    # Poll client for new state, get prediction, execute
    action = agent.act(state=client.get_state())
    reward = client.execute(action)

    # Observe feedback
    agent.observe(reward=reward, terminal=False)

```

Listing 1: Agent API example

and notify the model of terminal states. Updates to the model are implicitly triggered according to configurations.

The advantage to our approach is that practitioners can explore different RL paradigms in their applications simply by loading another configuration without the need to modify application code (e.g. to explicitly trigger certain updates or model-specific events), or library code. The code is available open source under <https://github.com/reinforceio/tensorforce>.

Features. TensorForce implements both classical algorithms serving as an entry point for practitioners as well as newer methods, which we briefly describe. From the family of Q-learning algorithms, our library implements the original deep Q-learning [50], double deep Q-learning [74], normalized advantage functions for continuous Q-learning [22], n-step Q-learning [48], and deep Q learning from demonstrations incorporating expert knowledge [27].

Further, we provide classic policy gradients (REINFORCE) [76], trust region policy optimization [61], and proximal policy optimization (PPO) [62] from the spectrum of policy-based methods, which all support categorical, continuous and bounded action spaces. It is worth pointing out that many new algorithms only modify classic Q-learning or policy gradients by slightly changing the loss functions, and implementing them only requires a few lines of code on top of existing TensorForce components.

Example usage. We illustrate how users might interact with the API in Listing 1. Developers specify a configuration containing at least a network specification and a description of states and action formats. Here, a single state with 10 inputs and two separate actions per step, one boolean, one discrete with 10 options are required. Single-node execution is chosen, and incoming states are normalized via a state preprocessor. Crucially, a large number of commonly used heuristics is both optional and transparently configurable.

Next, a PPO (a state-of-the-art policy optimization method, e.g. used in OpenAI’s recent work on DOTA [52]) agent is created using the configuration, and a client is instantiated to interact with an example remote system which we desire to control. The agent can now be used by retrieving new state signals from the client, which needs to map system state (e.g. load) to inputs, and requesting actions from the agent. The client must implement these actions by mapping numerical representations such as the index of a discrete action to a change in the system. Finally, the agent has to observe the reward to provide feedback to the agent. The agent will automatically trigger updates to the underlying TensorFlow graph based on algorithm semantics, e.g. episode based, batch-based, or time-step based.

Developers are thus freed from dealing with low-level semantics of deep learning frameworks and can concentrate on mapping their system to inputs, rewards and actions. By changing a few lines in the configuration, algorithm, data collection, learning, or neural network logic can be fine-tuned. Finally, the JSON configurations can be conveniently passed to auto-tuners for hyper-parameter optimization.

4.3 LIFT

LIFT uses the declarative agent API and a small set of reusable components to realize three different execution modes which we describe in this section.

Pretraining. In pretraining mode, LIFT does not interact with a system but is provided with a trace data source such as a comma separated file, a database table, or a distributed file system. LIFT parses and maps these to demonstrations (described in detail in section 5), creates an RL agent supporting pretraining, and imports data. It then executes and monitors pretraining through evaluators, i.e. by validating model performance, and finally by serializing the model.

Agent-driven. In agent-driven or **active execution**, LIFT alternates between interacting with the system (i.e. the environment) and the RL agent via the TensorForce API. Here, execution time is almost exclusively governed by waiting on the environment, as we show in §6. The RL libraries we surveyed typically only offer agent-driven execution (e.g. OpenAI baselines) where this execution is tightly coupled with reinforcement learning logic. This is because training

common simulation tasks such as the Arcade Learning Environment [7] can be effectively parallelized to hundreds of instances due to marginal computational requirements per simulator process. These highly parallel training procedures are economically impractical for users without data center scale resources, as learning to control data processing systems requires significant I/O and compute.

Environment-driven. In environment-driven execution or **passive execution**, LIFT acts as a passive service as control flow is driven by external workload, e.g. a benchmark suite executed against a database. For example, LIFT may open up a websocket or RPC connection to a monitoring service to receive real-time performance metrics. The LIFT controller then continuously maps incoming metrics to states, passes them to the agent, and executes the necessary configuration changes on the system. Passive execution is primarily intended for deployment of trained models which can optionally perform incremental updates. All execution modes share a common set of components which users need to implement for their given system to facilitate the parsing and serialization overhead necessary to interface a system.

First, a **schema** is used to programmatically construct the layouts of states, actions and rewards. For example, in our compound indexing case study, the input size to the neural network depends on the number of available query operators and unique fields in the database. In our experience, successful application of RL initially requires frequent exploratory iterations over different state and action layouts. In LIFT, this is reflected by users implementing multiple exchangeable schemas. Downstream components for the execution modes use a schema to infer shape and type information.

Next, users implement a **model converter** as the central component for translating between RL model and controlled system via a small set of methods called throughout LIFT to i) map system output to agent states and agent actions (for pretraining), ii) map system output to rewards, and iii) map agent output to system configuration changes. LIFT’s generic components for each execution mode then use converters to deserialize and parse log traces, and to perform offline (pretraining) and online (agent- or environment-driven) training.

We summarize the idea behind LIFT as motivated by two observations. First, unlike common RL simulation tasks, controlling data processing systems requires separation of environment and RL agent due to different resource needs and communication patterns (e.g. access to system metrics through RPC or other protocols). Second, using RL in practical contexts currently requires a large amount of boilerplate code as no standard tools are available. LIFT enables researchers to focus on understanding their state, action and reward semantics and express them in a schema and system model, which generate the respective TensorFlow graphs via

the TensorFlow API. In the following section, we explain the pretraining process on the indexing case study.

Implementation. We implemented our LIFT prototype in ≈ 10000 lines of Python code which includes components for our example case studies. In this work, no low-latency access is required (e.g. for learning to represent data structures as described by Kraska et al. [33]) but we may implement a C++ serving layer in future case studies.

5 LEARNING FROM TRACES

5.1 Problem setup

We now illustrate the use of LIFT in an end-to-end example based on our compound database indexing application. In database management, effective query indexing strategies are crucial for meeting performance objectives. Index data structures can accelerate query execution times by multiple magnitudes by providing fast look-ups for specific query operators such as range comparisons (B-trees) or exist queries (Bloom filters). A single index can span multiple attributes, and query planners employ a wide range of heuristics to combine existing indices at runtime, e.g. by partial evaluation of a compound (multi-attribute) index. Determining optimal indices is complicated by space usage, maintenance cost, and the fact that indexing decisions cannot be made independently of runtime statistics, as index performance depends on attribute cardinality and workload distribution. In practice, indices are identified using various techniques ranging from offline tool-assisted analysis [3, 11, 14] to online and adaptive indexing strategies [21, 24, 29, 55]. Managed database-as-a-service (DBaaS) offerings sometimes offer a hybrid approach where indices for individual attributes are automatically created but users need to manually create compound indices.

We study MongoDB as a popular open source document database where data is organized as nested J/JSON documents. While a large body of work exists on adaptive indexing strategies for relational databases and columnar stores [55], compound indexing in document databases has received less attention. Document databases are offered by all major cloud service providers, e.g. Microsoft’s Azure CosmosDB offers native MongoDB support [44], Amazon’s AWS offers DynamoDB [4], and Google Cloud provides Cloud Datastore [20]. The document database services we surveyed offer varying specialized query operators, index design, and query planners using different indexing heuristics. The aim of automatic online index selection is to omit this operational task from service users. We initially focus on common query operators available in most query dialects, as we plan to extend our work to other database layouts and query languages. Table 1 gives an operator overview. In MongoDB, queries themselves are nested documents.

Table 1: MongoDB basic operator overview.

Operators	MongoDB operator
=, >, ≥, <, ≤, not in	<code>\$eq</code> , <code>\$gt</code> , <code>\$gte</code> , <code>\$lt</code> , <code>\$lte</code> , <code>\$nin</code>
and, or, nor, not	<code>\$and</code> , <code>\$or</code> , <code>\$nor</code> , <code>\$not</code>
limit, sort, count	<code>count()</code> , <code>limit(n)</code> , <code>sort(keys)</code>

5.2 Modeling indexing decisions

The MongoDB query planner uses a single index per query with the exception of `$or` expressions where each sub-expression can use a separate index. An index may span between 1 and k schema fields and is specified via an ordered sequence of tuples $(f_1, s_1), \dots, (f_n, s_n)$ where each tuple consists of a field name f_i and a sort direction s_i (ascending or descending). At runtime, the optimizer will use a number of heuristics to determine the best index to use.

Via index intersection, the optimizer can also partially utilize existing indices to resolve queries. For example, *prefix intersection* means that for any index sequence of length k , the optimizer can also use any ordered prefix of length $1..k-1$ to resolve queries which do not contain all k attributes in the full index. Consequently, while the tuple ordering of the index does not typically matter for individual queries, the number of indices for the entire query set can be drastically reduced if index creation considers potential prefix intersections with other queries. Similarly, sort-ordering in indices can be used to sort query results via sort intersection in case of matching sort patterns. For example, an index of the shape $[(f_1, ASC), (f_2, DESC)]$ can be used to sort ascending/descending and descending/ascending (i.e. inverted) sort patterns, but not ascending/ascending or descending/descending. Based on these indexing rules, we define the following state, action, and reward model.

States. Identifying the correct index for a query requires knowledge of the query shape, e.g. its operators and requested attributes. To leverage intersection, the state must also contain information on existing indices which could be used to evaluate a query. We parse queries via a tree-walk, strip concrete values from each sub-expression, and only retain a sequence of operators and attributes. If an index already exists on an attribute, we insert an additional token after the respective attribute to enable the agent to learn about index intersection and avoid adding unnecessary indices. For example, consider the simple following query counting entries with name "Jane":

```
collection.find({'$eq: {name: "Jane"}}).count()
```

Assuming an ascending index on the `name` field already exists, the tokenized query looks as follows (with EOS representing the end-of-sentence):

```
[$eq name IDX_ASC count EOS]
```

These tokens are then converted to integers using a word embedding as commonly used in natural language processing applications to map a discrete set of words to a continuous vector space [45]. In practice, a maximum fixed input length is assumed and shorter inputs are padded with zeros.

Actions. For every query we seek to output an index (or none) spanning at most k attributes where k is a small number as indices covering more than 2-4 attributes are rare in practice. This is also because compound indices containing arrays, which require multi-key indices (each array element indexed separately), scale poorly and can slow down queries. Additionally, as discussed above, index intersection makes indices order- and sort-sensitive, thus requiring to also output a sort order per attribute in a multi-key index.

The action scheme should scale independently of the number of attributes in the document schema. Consider a combinatorial action model where the agent is modelled with one explicit action per attribute, and a separate action output per possible index-key. A 3-key index task on 10 attributes would already result in thousands of action options per step ($10^3 * 3 = 3000$) when including an extra action for the three possible sort patterns (both ascending/descending, descending-ascending, ascending-descending). This approach would not generalize to changing schemas or data sets. We propose a positional action model wherein the number of actions is linear in k . When receiving a query, we extract all query attributes and interpret an integer action as creating an index on the i th input attribute, thus allowing the agent to learn the importance of key-order for prefix intersection. To distinguish sort patterns, we create an extra action per key (one ascending, one descending with ascending default). This results in $1 + 2k$ actions for a k -key index with one output for no-op.

Figure 3 illustrates state and action parsing for $k = 2$ and a simple query on `name` and `age` attributes. In the example, the `name` field is already indexed so when the query is tokenized, a special index token (`IDX_ASC`) is inserted to indicate the existing index. The tokenized sequence is mapped to integers via the embedding layer and passed through the network, which outputs k integer actions. In the example, the agent decides to implement one additional single-key index by outputting 3 and 0, where 3 implies an ascending index on the second input attribute, and 0 is used for no-op if fewer than k keys are required in the index.

Rewards. The optimal indexing strategy is the minimal set of indices I meeting performance level objectives such as mean latency or 90th and 99th latency percentiles for a set of queries Q . Let $t(q)$ be the time to execute a query $q \in Q$ under an index set I and let $m(I)$ be the memory usage of the current index set. We set the reward $r(q)$ as the negative

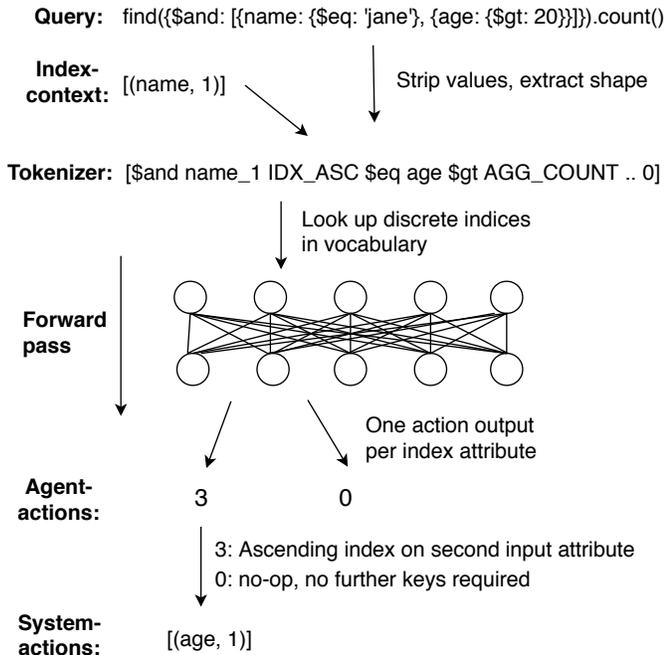


Figure 3: State and action parsing scheme for the indexing case study.

weighted combination of these to allow expressing trade-offs on memory usage against runtime requirements:

$$r(q) = -\omega_1 m(I) - \omega_2 t(q).$$

5.3 Demonstrations and Pretraining

We now describe the ideas behind learning from demonstrations as used in LIFT. Our approach is motivated by the observation that a human systems developer encountering a tuning problem can frequently use their expertise to come up with an initial heuristic. For example, in the indexing problem, a database expert can typically determine an effective configuration for a given application within a reasonable time frame (e.g. a few hours) with access to profiling tools. Distilling this intuitive expertise into a fully automated approach is difficult, and simple heuristics may perform well in small scenarios but fail at scale. Moreover, as discussed in §3, training a RL model from scratch is expensive and difficult, while refining a model pretrained from not necessarily fully correct demonstrations may be more effective. We hence argue for an approach that leverages pre-existing domain knowledge by initializing training from demonstrations.

Demonstration data. In the indexing task, demonstrations may exist in the form of:

- (1) Query logs from applications configured by a database administrator where indices are assumed to be correct, where correctness implies fully meeting service level objectives (not necessarily being optimal).

- (2) Query logs from applications where indices were created using any heuristic understood to be sub-optimal and not necessarily meeting service objectives.
- (3) Queries and index pairs for which no runtime data is available, e.g. procedurally generated examples with either manually or heuristically chosen index recommendations (both correct and imperfect).

The key difference between (1) and (2) is that when encountering a query for which an imperfect demonstration was available during pre-training, we do not mind testing other choices while this is unnecessary if a demonstration was optimal for the query given. This confidence must be reflected in the pretraining procedure. Further, the difference between (1), (2) and (3) is that in the latter, no reward is available without creating indices and measuring queries. Note that the key difference between demonstrations and simulation in our applications is the absence of information on system dynamics (i.e. state transitions).

A simulator for query indexing would provide insights into how addition and removal of an index affects performance. In contrast, a demonstration extracted from the slow query log of a database indicates how fast a query performed using the index chosen by the query planner, but not how much faster the index was versus not using an index, or a different index. We make use of all demonstration types but focus on (2) and (3), as we could not obtain existing traces from expert-configured systems and thus had to manually tune configurations.

Algorithm. Hester et al. have described an algorithm to perform Deep Q-learning from such expert demonstrations (DQfD) using the example of Atari games [27]. In their work, an agent is trained until sufficient performance, and then games played by that agent are given as demonstrations to a new agent. DQfD works by assigning an 'expert margin' to demonstration actions by extending double Q-learning [74], a Q-learning variant which corrects biased Q- estimates in the original DQN by decoupling action selection and action evaluation. Specifically, the double DQN loss

$$J_{DQ}(Q) = (R(s, a) + \gamma Q(s_{t+1}, a_{t+1}^{max}; \theta') - Q(s, a; \theta))^2 \quad (3)$$

where

$$a_{t+1}^{max} = \operatorname{argmax}_a Q(s_{t+1}, a; \theta) \quad (4)$$

uses the target network (as explained in §2, parametrized by θ') to evaluate the action selected using the training network (with parameters θ). This is combined with another expert loss function J_E :

$$J_E(Q) = \max_{a \in A} [Q(s, a) + l(s, a_E, a)] - Q(s, a_E) \quad (5)$$

Here, $l(s, a_E, a)$ is a function which outputs 0 for the expert action, and a margin value > 0 otherwise. We convey the

intuition of this loss function by recalling the action selection mechanism in Q-learning.

Recall that $Q(s, a, \theta) = \mathbb{E}[R_t | s_t = s, a]$, i.e. the expected returns from taking a decision a in state s . At each step, the neural network (parameterized by θ) used to approximate Q outputs Q-values for all available actions and selects the action with the highest Q-value. By adding the expert margin to the loss of Q-values of *incorrect actions*, the agent is biased towards the expert actions as a difference between expert actions and other actions of at least the margin is enforced [56]. The DQFD-agent keeps a separate memory of these expert demonstrations which are first used to pretrain the agent, then combined with new online experiences at runtime so that the agent keeps being 'reminded' of demonstrations.

What does the choice of $l(s, a_E, a)$ imply for imperfect or noisy demonstrations? A large margin makes it difficult to learn about any better actions in a given state because even if, via exploration, a different action is selected and yields a higher return, an update may not change Q-values of better action beyond the margin. Second, the DQFD loss only enforces a difference in Q-values between demonstrated action and all other actions; no assumptions are made about the relationship between non-expert actions (e.g. second highest, third highest Q-value). This behavior is desirable in the indexing example because even semantically similar indices (e.g. different order, partially covering same fields) can result in much worse performance than the demonstrated index, so we initially do not want to express any preference on non-demonstrated indices. Consequently, we choose a very small margin ≤ 0.1 which in practice results in a pre-generated model which initially only slightly favors the demonstrated action.

5.4 Putting it all together.

Algorithm 1 shows pseudo-code for the online training procedure. Following pre-training on the demonstration data set, we start LIFT in online mode, initialize an agent with the demo model, and load the demo data. We then begin the episodic training procedure on a new set of queries Q_{test} we want to index. In each training episode, all indices are first removed from the database. Then, each query q (sorted by length to improve intersection) is tokenized and the suggested index created. Recall that the tokenization includes the current index set \mathcal{I} for the agent to learn the impact of existing indices. The size of the index set $m(\mathcal{I})$ and the runtime of the query $t(q)$ are used to inform the reward of the agent. For direct search tasks like indexing, we keep the list of index tuples associated with the highest reward during training. In the final evaluation, we recreate these indices and then run all queries 5 times on the full index set. For

Algorithm 1 Online training procedure.

```

Initialize agent with demo-model and demo-data  $D$ 
Initialize LIFT system_model, model_converter
Load application queries  $Q_{test}$ 
// Fixed time budget or until objectives met
for  $i = 1, N$  do
     $\mathcal{I}_{test} \leftarrow \emptyset$ , clear index set in DB
    for  $q$  in  $Q_{test}$  do
        // Tokenize, include existing indices
         $s(q) \leftarrow model\_converter.to\_agent\_state(q, \mathcal{I}_{test})$ 
         $index \leftarrow agent.act(s(q))$ 
        // Create index, execute query
         $m(\mathcal{I}_{test}) \leftarrow system\_model.act(index)$ 
         $t(q) \leftarrow system\_model.execute(q)$ 
        // Compute reward from runtime and size
         $r_q \leftarrow -\omega_1 m(\mathcal{I}_{test}) - \omega_2 t(q)$ 
         $agent.observe(r_q)$ 
        Add  $index$  to  $\mathcal{I}_{test}$ 
    end for
end for
// Final evaluation, create best  $\mathcal{I}_{test}$ :
// Measure final size  $m(\mathcal{I}_{test})$ , run  $Q_{test}$ 

```

dynamic tasks where the agent is invoked repeatedly at runtime, we simply export the trained model which can then be used to control a system.

6 EVALUATION

6.1 Aims

We evaluate our LIFT prototype through two case studies: 1) the indexing case study in which we minimize latency and memory usage by learning compound index combinations, and 2) the stream processing resource management case study in which we tune latency by setting parallelism levels under a varying workload. In both case studies, we used LIFT to implement a controller, manage demonstration data, and interact with the system. The difference is that the indexing task is an offline optimization (index set is determined once, then deployed), while in the stream processing task we use a controller at runtime to react to varying workloads. The evaluation focuses on evaluating the utility of LIFT and TensorFlow to solve data management tasks, and on understanding the impact of learning from demonstrations to overcome long training times.

6.2 Compound indexing

Setup. We evaluate the indexing task both on a real-world dataset (IMDB [30]) and using synthetic queries and data. The synthetic query client is based on the YCSB benchmark [13]. YCSB generates keys and synthetic values to evaluate

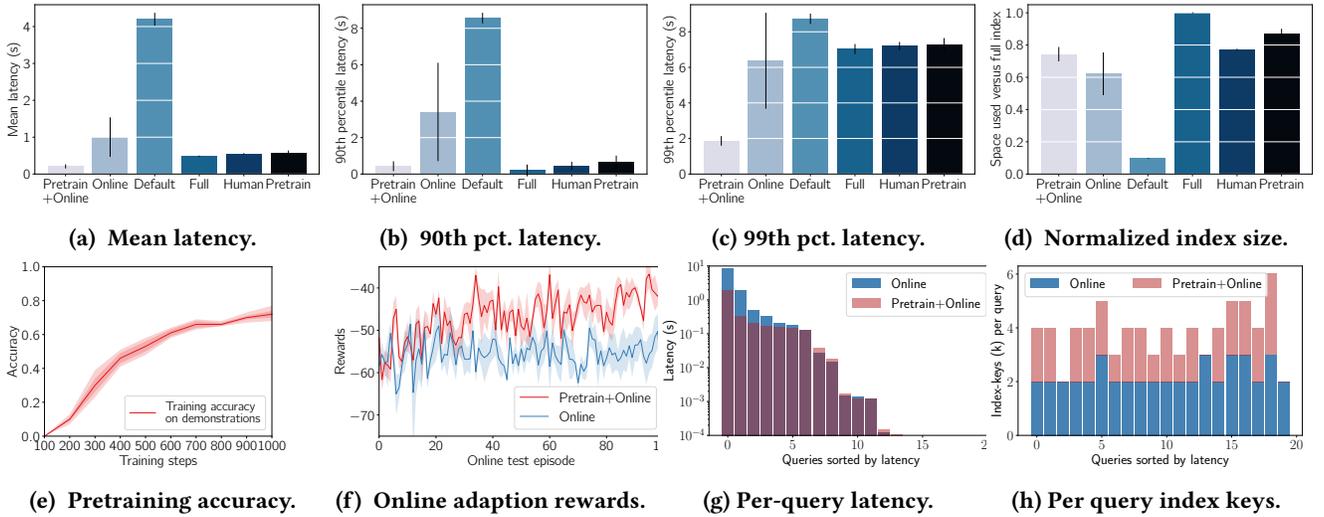


Figure 4: Performance evaluation on the IMDB data set.

cloud database performance via a set of common workload mixtures but has no provisions for complex queries. We implemented a YCSB-style client and workload generator targeting secondary indexing. The client is configured with a schema containing attribute names and types. The workload generator receives a query configuration containing valid query operators, maximum allowed operator degrees, query height, and distribution of aggregation operators. It can then generate a specified number of queries index suggestions based on provided rules. All experiments were run on a variety of commodity server class machines (e.g. 24 cores (Xeon E5-2600) and 198 GB RAM) and using MongoDB v3.6.4.

Queries and demonstrations are imported into LIFT’s pretrain-controller which instantiates a DQfD agent and parses queries and demonstrations to states and actions as described before. We then run a small number of pretraining steps until the agent has approximately adopted the rule. We use batch sizes of 32 queries on a neural network with 1 embedding layer and 1 dense layer with 128 neurons each. Learning is executed using an adaptive moment optimizer (Adam [32]) with a learning rate of 0.0005, and an expert margin of 0.1. To refine the pretrained model, we restart LIFT in online mode and train as described in Algorithm 1. The max number of attributes per index was $k = 3$.

Indexing baselines. We consider human and rule-based baselines due to a lack of standard tools for automatic indexing in document stores. The first rule-based strategy we use to generate demonstrations is full indexing (*Full* in the following) wherein we simply create a compound index covering all fields in a query (respecting its sort order), thus ensuring an index exists for every query. In the synthetic regime,

where query shapes are re-sampled every experiment to evaluate generalization to different query sets, human baselines were uneconomical, and we experimented with other rule-based baselines. Partial indexing (*Partial* hereafter) attempts to avoid unnecessary indices by considering existing indices on any attribute in a query, and only covering unindexed fields. Note that we do not claim these to be the most effective heuristics but merely initial guidance for a model. We also experimented with a rule based on operator and schema hints but this frequently did not perform well due to unforeseen edge cases. We refer to the following modes in the evaluation: no indexing (*Default*), online learning from scratch without pretraining (*Online*), pretraining without online refinement (*Pretrain*, online learning following pretraining (*Pretrain+Online*), human expert (*Human*) and the two baselines described above.

Basic behaviour. We first show results on the publicly available internet movie database (IMDB) datasets [30]. We imported datasets for titles and ratings (*title.akas*, *title.basics*, *title.ratings*) comprising ≈ 10 million documents. We manually defined a representative set of 20 queries such as ‘How many comedies with length of 90 minutes or less were made before 2000?’. For this fixed set, we compared our method to human expert intuition. Using human baselines (which are common in deep learning tasks) in data management is difficult due to inherent bias and prior knowledge on experiment design. Generally, a human expert can identify effective indices for a small query set given unlimited trials to refine guesses. For a more interesting comparison, we hence devised a single-pass experiment where the expert was allowed to observe runtimes on the full indexing baseline and subsequently tried to estimate an index per query. Figures 4a, 4b

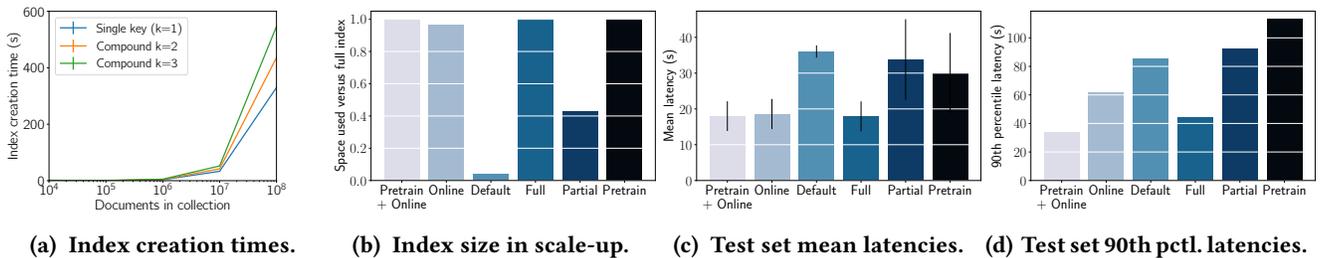


Figure 5: Scalability generalization analysis using the synthetic query client. Learning was performed on 10 million documents, a new set of test queries was evaluated on 100 million documents.

and 4c give mean, 90th and 99th latencies respectively on the final evaluation in which each query is executed 5 times (final results averaged over five trainings with different random seeds). The combined *Pretrain+Online* strategy outperforms other methods significantly, in particular improving mean latency by 57% and 62% against *Full* and *Human* respectively and by 74% on 99th percentile latency against both. Differences in 90th percentile latencies were within standard error. In the human experiment, the expert attempted to reduce indices by leveraging intersection, creating only 14 indices versus 15 on average for *Pretrain+Online*. The size of the created indices was however (marginally) bigger compared to *Pretrain+Online* (as shown in Figure 4d) which achieved on average 25% index size improvement against the *Full* strategy. Note that MongoDB always creates a default index on the *_id* attribute so the default size is not zero. We normalize sizes against the size of the full index to evaluate improvement. The expert’s attempt to exploit intersection also underestimated the necessity of compound indices for some queries. The outcome illustrates the difficulty of solving the task without iterative manual analysis.

Pretrain and *Online* can perform similar to full indexing. The performance of *Pretrain* (in its degree of similarity to *Full*) depends on whether pretraining is continued until the rule is fully adopted. We found early stopping at 70 – 80% accuracy to be effective when using our imperfect rules to avoid overfitting (Figure 4e). *Online* can sometimes find good configurations but tends to perform significantly worse than *Pretrain+Online* in mean reward due to random initialization, as seen in Figure 4f which shows reward curves (i.e. combined size and latency) and 1σ confidence intervals over 5 runs. We breakdown individual queries and indices for *Pretrain+Online* and *Online*, i.e. standard RL. In Figure 4g, we sort queries of one experiment by latency and show runtime differences (n.b. log scale), and in figure 4h the number of keys in the index decision (0-3) for the query (stacked on top of each other). Performance differences are concentrated in the five slowest queries with the rest being effectively indexed by both strategies. Comparing keys used per query also

Workload means	Total time	Pct.
Waiting on system	64869 s (\pm 4403 s)	97.8 %
Agent interaction/evaluation	1446 s (\pm 218 s)	2.2 %
Mean episode duration	663 s (\pm 42 s)	n/a
Min episode duration	419 s (\pm 88 s)	n/a
Max episode duration	880 s (\pm 62 s)	n/a
Pretrain+Online time to max	43044 s (\pm 16106 s)	n/a
Online time to max	19088 s (\pm 15011 s)	n/a

Table 2: Wall clock times on the IMDB data set. One episode refers to creating the entire application index set. On average, *Pretrain+Online* reaches its max performance much later in the experiment as it keeps improving while *Online* stops improving early.

shows that *Pretrain+Online* created indices systematically spanning fewer keys than *Online*. This does not necessarily imply smaller total index size depending on the attributes indexed but indicates more effective intersection.

Timing. Next, we analyze time spent in different training phases. Due to small neural network size, pretraining could be comfortably performed within few minutes on a CPU. This includes intermediate evaluations to test accuracy of the model on the set of rule-based demonstrations, and identifying conflicting rules. In table 2, we break down time spent interacting with TensorFlow for requesting actions/performing updates, and time spent waiting on the environment and evaluating indexing decisions by running queries. 97.9% of time was spent waiting on the database to finish creating and removing indices, and only 2.1% was spent on fetching and evaluating actions/queries, and updating the RL model. Pre-training is negligible compared to online evaluation times, so pretraining is desirable if data is available. If LIFT is used for online training, employing pretraining only requires few extra converter methods.

Scalability. The indexing problem is complicated by step durations growing with problem scale. Figure 5a shows index creation times for increasing collection sizes. At 100 million documents generated from our synthetic schema, creating

an index set for a set of queries can take hours, resulting in weeks of online training. As RL algorithms struggle with data efficiency, we believe these scalability problems will continue to present obstacles for problems such as cluster scheduling. We explore an approach where training is performed on a small data set of 10 million documents. Newly sampled test queries are evaluated on the 100 million document collection without further refinement. Figures 5b, 5c, and 5d show index size and latencies. All learned strategies created one index per query with query runtimes increasing corresponding to document count, and *Pretrain+Online* performing best. Latency metrics were dominated by a single long-running query with two expensive *\$gt* expressions which could not be meaningfully accelerated. While scalability transfer results show some promise, we plan to investigate an approach where a model of the query planner is learned to be able to evaluate indices without needing to run them at full scale.

Workload means :	Mean	90th	99th	Norm. Size (GB)
<i>Pretrain+Online</i>	0.5 s	1.7 s	3.5 s	0.43
<i>Online</i>	0.55 s	2.1 s	3.5 s	0.53
<i>Default</i>	0.94 s	2.7 s	3.6 s	0.03
<i>Full</i>	0.51 s	1.5 s	3.9 s	1.0
<i>Partial</i>	0.96 s	3.4 s	4.4 s	0.32
<i>Pretrain</i>	0.59 s	2.2 s	4.1 s	1.0

Table 3: Performance variation when sampling different query sets per run. *Min*, *90th*, *99th* are referring to average latencies across different query sets.

Generalization. Traditional database benchmarks such as TPC-H use fixed sets of query templates sampling different attribute values at runtime. This is problematic from a deep learning perspective as the number of distinct query shapes (i.e. operator structure) is too small to evaluate generalization to unseen queries. Our synthetic benchmark client does not only sample attribute values on fixed shapes, but also query shapes. We investigate query generalization via our synthetic client by sampling 5 different query sets, and reporting on variation in learning performance. We insert 5 million documents with 15 attributes with varying data types (schema details provided in appendix). Next, 10,000 queries and rule-based demonstrations are generated using *Full* indexing as the demonstration rule. We did not see improvement when generating more examples, indicating these were sufficient to cover rule behaviour on the synthetic schema. We pretrain on these queries as before, then sample 20 new queries as the test set and perform online training. Table 3 gives an overview on performance variation across query sets. *Pretrain+Online* saves more than 50% space while performing better or comparably across latency metrics. *Partial* saves even more space but fails on improving latency. Values

are averaged across different tasks, thus means per task are expected to be different. Importantly, performance of our approach is not an artefact on a specific query set designed for this task but generalizes.

6.3 Stream task parallelism

Problem setup. Distributed stream processing systems (DSPS) such as Storm [70], Heron [34] or Flink [5] are widely used in large scale real time processing. To this end, DSPS have to meet strict service level objectives on message latency and throughput. Achieving these objectives requires careful tuning of various scheduling parameters, as processing instances may fail and workloads may vary with sudden spikes. Floratou et al. suggested the notion of self-regulating stream processing with Dhalion [17], a rule-based engine on top of Heron which collect performance metrics, identifies symptoms of performance problems (e.g. instance failure), generates diagnoses and tries to resolve issues by making adjustments (e.g. changing packing plan). We use LIFT to learn to tune task parallelism in Heron using RL. Task parallelism corresponds to the number of physical cores allocated to a specific task in the processing topology. We use the same 3 stage word-count topology as described in Dhalion on a small cluster using 5 machines (1 master, 4 slaves).

Model. We again use LIFT to implement state and action models, and to interface Heron’s metric collection. For the state, we use a matrix containing CPU and memory usage, and time spent in back-pressure (a special message used by Heron to indicate lack of resources on a task) for all task instances. As actions, the agent outputs (integer) task parallelism values for each component in the topology. The reward is a linear combination of normalized message latencies square roots (to smooth outliers), throughput, and the fraction of available instances used.

Results. Collecting data for the stream processing task is difficult as each step requires multiple minutes of collecting metrics so performance can stabilize after changes, and updating the topology by creating a new packing plan and deploying it. Due to an outstanding issue in the scheduler, we did not manage to run Dhalion itself. We could also not easily port its parallelism rule to LIFT because not all used metrics were exposed via the Heron tracker API. For the purpose of this experiment, we hence collected demonstration data from a simple threshold rule. The aim of this case study is hence not to prove superiority over Dhalion, but evaluate if rule-based demonstrations can help RL in dynamic workload environments. We train and evaluate dynamic behavior by randomly sampling different workload changes such as load moving up and down periodically, or changing from low to high/high to low. Figure 6 shows results by comparing average reward over the duration of the evaluation which

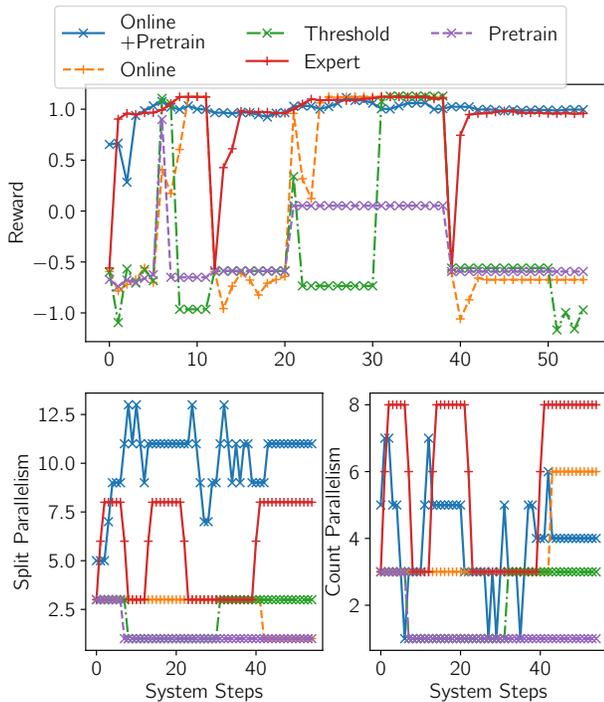


Figure 6: Top: Rewards through varying workload. Bottom: Task parallelisms for splitter and count bolts.

presented the controller with all possible workloads in deterministic order. Each step corresponds to about 2-4 minutes real time to receive metrics and implement changes.

We defined an *Expert* configuration which had predetermined good configurations for each workload change. The bottom row shows how parallelism settings for both bolts are adjusted by the different strategies over time. The *Expert* systematically alternates between two configurations for each component, incurring temporary low rewards upon changes. The *Pretrain+Online* agent managed to avoid temporary reward loss by anticipating workload changes, and by always keeping split parallelism high as to have enough capacity for changes, thus outperforming the pre-tuned *Expert* configurations slightly (3%. This 'anticipation effect' is a consequence of the agent observing regularities in how workloads change. *Online* failed to adopt an effective strategy within the same training time (1.5 days). Other methods performed worse although the threshold rule-based model could have been improved by manually fitting thresholds to workload changes (thus being closer to *Expert*).

We provide further analysis by comparing training rewards with and without pretraining in Figure 7. *Online* without pretraining could on average not recover good configurations, thus most of the time being at a low reward, and only occasionally seeing high rewards when workloads matched its configuration. In contrast, *Pretrain+Online* achieved much

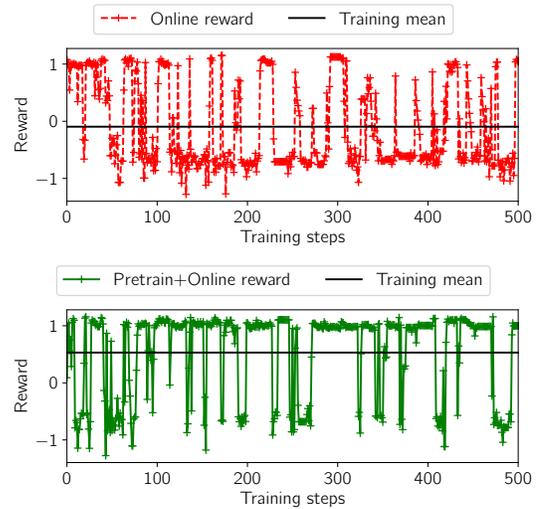


Figure 7: Heron training rewards.

higher mean rewards as after around 100 episodes of training, it began to quickly recover from workload changes to reach (normalized) high reward regions again. Our experiments show the combination of pretraining and online refinement can produce effective results on dynamic workloads. A key question is if workloads in practice exhibit irregularities which are difficult to address through manual tuning. We suspect the advantage of RL will increase for larger topologies with many different bolt types on heterogeneous resources.

6.4 Discussion

Limitations. Our results indicate the potential of imperfect demonstrations when applying RL to data management tasks, improving latency metrics by up to 70% in the IMDB case study against several baselines, and outperforming the expert configuration in Heron. Our experiments did not include some subtasks which prolong training times. For example, in the indexing task, we omitted considerations for indexing shards and replica sets. As RL applications in data management move from simulation to real world, they will incrementally cover additional subtasks. We also showed the difficulty of tackling tasks where step times increase with scale. Here, mechanisms such as pretraining and training on partial tasks provide a promising direction to eventually apply RL at data center scale.

Learning. The algorithmic limitations of current RL algorithms continue to present significant limitations in real world applications. Learning from scratch can take infeasibly long and may also be unreliable due to the stochastic nature of training. Further, learning a task once and deploying the resulting model in different contexts is unlikely to succeed due to the sensitivity of RL algorithms [31]. We rely on online refinement following the pretraining procedure

which incurs only little overhead after initial implementation. The aim of our experiments was not to demonstrate the best way to e.g. perform workload management in stream processing. Recent work has illustrated how neural networks struggle with forecasting spiky workloads [40]. We focused on evaluating if pretraining can help the notoriously difficult application of RL to data management tasks. In summary, DRL remains a promising direction, as even results in this exploratory phase show the ability to learn complex behaviour which normally requires manual solution design.

7 RELATED WORK

RL in data management. Early work in exploring RL in computer systems can be found in routing (Q-routing) and protocol optimization [8, 35, 36]. Subsequent research has covered a diverse range of domains such as cloud workload allocation, cluster scheduling, networking, or bitrate selection [15, 42, 43, 68, 72]. Many of these works have outperformed existing approaches in simulation, but did not translate into real world deployments due to the difficulties discussed elsewhere in this paper. Notably, the idea of using neural networks in combination with RL in systems can be found as early as 2006 in Tesauro et al.’s work on server resource allocation [69]. The authors incorporated pre-existing knowledge by initially bootstrapping control from a rule, whereas we use offline demonstrations to reduce training times. RL for combinatorial selection has also found application in tuning compression of neural networks for mobile devices [39]. Mirhoseini et al. demonstrated how to use attention-based and we p hierarchical methods known from neural machine translation to effectively perform TensorFlow device placements [46, 47]. The difference to our work is that each graph step only takes few seconds so online training can be performed more effectively. Sharma et al. used RL to learn single-key indices in relational databases, and simplified the problem by manually constructing features such as selectivity [64].

Adaptive indexing. A large body of work exists on indexing strategies which are widely used in practice. Offline indexing is performed using the design tuning tools provided by commercial database products which require database administrators to manually interact with the tool, and make ultimate design decisions [3, 11, 14]. Online indexing addresses this limitation by making decisions based on continuous online monitoring [60]. Adaptive (or holistic [55]) indexing (e.g. in columnar databases) enable even faster reaction to workload changes by building and refining indices via lightweight incremental modifications [21, 24, 29]. A similar depth of work in indexing is not available for document databases, although many techniques are likely transferable [57]. Commercial

MongoDB services sometimes offer index recommendations based on longer term workload patterns ([51]).

ML in databases. Recently, machine learning approaches have been explored in data management. Pavlo et al. proposed the idea of a self-driving database with initial focus on employing ML techniques for workload forecasting [53]. In subsequent work, these forecasts were evaluated on their ability to help create SQL indices [40]. Their work in particular found that neural networks were not as effective in capturing spiky loads as traditional time series techniques. OtterTune [73] automatically determines relevant database parameters to create end-to-end tuning pipelines [19]. BO is not easily applicable in problems like index selection or generally combinatorial problems as it requires a similarity function (Kernel) to interpolate a smooth objective function between data points. Defining a custom Kernel between databases is difficult because semantically similar indices can perform vastly different on a workload. Kraska et al. explored representing the index data structure itself as a neural network with the aim to learn to match data distributions and access patterns [33]. Bailis et al. subsequently argued that well tuned cuckoo hashing could still outperform these learned indices [54]. We similarly argue that deep RL techniques are in an exploratory phase and cannot yet replace well established tuning methods.

Learning from demonstrations. Learning from expert demonstration is a well studied notion in RL. DAGGER (for Dataset Aggregation) is a popular approach for imitation learning which requires an expert to continuously provide new input [59]. While this is not directly compatible with learning from traces, we are considering future additions to LIFT where a human may interactively provide demonstrations between trials to further accelerate training. Other familiar approaches include behavioural cloning, recently also in the context of generative adversarial models [28, 75]. Snorkel is a system to help generate weakly supervised data via labeling functions which is conceptually similar to our rule-based demonstrations [58]. In this work, we relied on DQfD as a conceptually simple extension to Deep Q-learning [27]. Its main advantage is that the large margin function gives a simple way of assigning low or high confidence to demonstrations via single tunable parameter, thus in practice also allowing the use of imperfect demonstrations. Gao et al. recently suggested employing a unified objective which incorporates imperfect demonstrations naturally by accounting for uncertainty using an entropy approach [18].

8 CONCLUSION

In this paper, we discuss long evaluation times, algorithmic instability, and lack of widely available software as key obstacles to the applicability of RL in data management tasks. To

help address these issues, we introduce LIFT, the first end-to-end software stack for applying RL to data management. As part of LIFT, we also introduce TensorForce, a practical deep reinforcement learning library providing a declarative API to common RL algorithms. The key idea of LIFT is to help developers leveraging existing knowledge from trace data, rules or any other form of demonstrations to guide model creation. We demonstrate the practical potential of LIFT in two proof-of-concept case studies. If online-only training takes impractically long, our results show that pretraining can significantly improve final results.

REFERENCES

- [1] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467* (2015).
- [2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, Vol. 16. 265–283.
- [3] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*. Very Large Data Bases Endowment Inc. <https://www.microsoft.com/en-us/research/publication/database-tuning-advisor-for-microsoft-sql-server-2005/>
- [4] Amazon Inc. 2018. Amazon DynamoDB. website. (2018). <https://aws.amazon.com/dynamodb/>
- [5] Apache Foundation. 2018. Apache Flink. website. (2018). <https://flink.apache.org>
- [6] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. A Brief Survey of Deep Reinforcement Learning. *arXiv preprint arXiv:1708.05866* (2017).
- [7] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. 2013. The Arcade Learning Environment: An evaluation platform for general agents. *J. Artif. Intell. Res. (JAIR)* 47 (2013), 253–279.
- [8] Justin A Boyan and Michael L Littman. 1994. Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in neural information processing systems* (1994), 671–671.
- [9] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI gym. *arXiv preprint arXiv:1606.01540* (2016).
- [10] Itai Caspi, Gal Leibovich, and Gal Novik. 2017. Reinforcement Learning Coach. (Dec. 2017). <https://doi.org/10.5281/zenodo.1134899>
- [11] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin “What-if” Index Analysis Utility. *SIGMOD Rec.* 27, 2 (June 1998), 367–378. <https://doi.org/10.1145/276305.276337>
- [12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).
- [13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC ’10)*. ACM, New York, NY, USA, 143–154. <https://doi.org/10.1145/1807128.1807152>
- [14] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. 2004. Automatic SQL Tuning in Oracle 10G. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30 (VLDB ’04)*. VLDB Endowment, 1098–1109. <http://dl.acm.org/citation.cfm?id=1316689.1316784>
- [15] Xavier Dutreilh, Sergey Kirgizov, Olga Melekhova, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. 2011. Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*. 67–74.
- [16] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. [n. d.]. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. ([n. d.]). [arXiv:cs.LG/1802.01561v2](https://arxiv.org/abs/1802.01561v2)
- [17] Avriilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1825–1836.
- [18] Yang Gao, Huazhe, Xu, Ji Lin, Fisher Yu, Sergey Levine, and Trevor Darrell. [n. d.]. Reinforcement Learning from Imperfect Demonstrations. ([n. d.]). [arXiv:cs.AI/1802.05313v1](https://arxiv.org/abs/1802.05313v1)
- [19] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1487–1495.
- [20] Google Inc. 2018. Google Cloud Datastore. website. (2018). <https://cloud.google.com/datastore/>
- [21] Goetz Graefe and Harumi Kuno. 2010. Self-selecting, Self-tuning, Incrementally Optimized Indexes. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT ’10)*. ACM, New York, NY, USA, 371–381. <https://doi.org/10.1145/1739041.1739087>
- [22] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. 2016. Continuous Deep Q-Learning with Model-based Acceleration. (March 2016). [arXiv:1603.00748](https://arxiv.org/abs/1603.00748)
- [23] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. 2017. Reinforcement learning with deep energy-based policies. *arXiv preprint arXiv:1702.08165* (2017).
- [24] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. 2012. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-memory Column-stores. *Proc. VLDB Endow.* 5, 6 (Feb. 2012), 502–513. <https://doi.org/10.14778/2168651.2168652>
- [25] Daniel Hein, Stefan Depeweg, Michel Tokic, Steffen Udluft, Alexander Hentschel, Thomas A Runkler, and Volkmar Sterzing. 2017. A Benchmark Environment Motivated by Industrial Control Problems. *arXiv preprint arXiv:1709.09480* (2017).
- [26] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2017. Deep reinforcement learning that matters. *arXiv preprint arXiv:1709.06560* (2017).
- [27] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. 2017. Learning from Demonstrations for Real World Reinforcement Learning. *CoRR abs/1704.03732* (2017). [arXiv:1704.03732](https://arxiv.org/abs/1704.03732) <http://arxiv.org/abs/1704.03732>
- [28] Jonathan Ho and Stefano Ermon. [n. d.]. Generative Adversarial Imitation Learning. ([n. d.]). [arXiv:cs.LG/1606.03476v1](https://arxiv.org/abs/1606.03476v1)
- [29] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. 2011. Merging What’s Cracked, Cracking What’s Merged: Adaptive Indexing in Main-memory Column-stores. *Proc. VLDB Endow.* 4, 9 (June 2011), 586–597. <https://doi.org/10.14778/2002938.2002944>
- [30] imdb.com. 2018. IMDb Datasets. website. (2018). <https://www.imdb.com/interfaces/>
- [31] Ken Kansky, Tom Silver, David A. Mély, Mohamed Eldawy, Miguel Lázaro-Gredilla, Xinghua Lou, Nimrod Dorfman, Szymon Sidor, Scott

- Phoenix, and Dileep George. [n. d.]. Schema Networks: Zero-shot Transfer with a Generative Causal Model of Intuitive Physics. ([n. d.]). arXiv:cs.AI/1706.04317v2
- [32] Diederik Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. (Dec. 2014). arXiv:1412.6980
- [33] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. [n. d.]. The Case for Learned Index Structures. ([n. d.]). arXiv:cs.DB/1712.01208v2
- [34] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 239–250.
- [35] Shailesh Kumar and Risto Miikkulainen. 1997. Dual reinforcement Q-routing: An on-line adaptive routing algorithm. In *Artificial neural networks in engineering*.
- [36] Shailesh Kumar and Risto Miikkulainen. 1999. Confidence based dual reinforcement q-routing: An adaptive online network routing algorithm. In *IJCAI*, Vol. 99. Citeseer, 758–763.
- [37] Yuxi Li. 2017. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274* (2017).
- [38] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. 2017. Ray RLLib: A Composable and Scalable Reinforcement Learning Library. *arXiv preprint arXiv:1712.09381* (2017).
- [39] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. 2018. On-Demand Deep Model Compression for Mobile Devices: A Usage-Driven Model Selection Framework. (2018).
- [40] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 631–645. <https://doi.org/10.1145/3183713.3196908>
- [41] Horia Mania, Aurelia Guy, and Benjamin Recht. [n. d.]. Simple random search provides a competitive approach to reinforcement learning. ([n. d.]). arXiv:cs.LG/1803.07055v1
- [42] 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*. ACM, 50–56.
- [43] 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 (SIGCOMM '17)*. ACM, New York, NY, USA, 197–210. <https://doi.org/10.1145/3098822.3098843>
- [44] Microsoft. 2018. CosmosDB - A globally distributed database for low latency and massively scalable applications, with native support for NoSQL. website. (2018). <https://azure.microsoft.com/en-gb/services/cosmos-db/>
- [45] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. [n. d.]. Efficient Estimation of Word Representations in Vector Space. ([n. d.]). arXiv:cs.CL/1301.3781v3
- [46] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. 2018. Hierarchical Planning for Device Placement. <https://openreview.net/pdf?id=Hkc-TeZ0W>
- [47] 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. *arXiv preprint arXiv:1706.04972* (2017).
- [48] Volodymyr Mnih, Adria Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. (Feb. 2016). arXiv:1602.01783
- [49] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602* (2013).
- [50] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fiedjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.
- [51] ObjectLabs Corporation. 2018. mLab cloud service. website. (2018). <https://mlab.com>
- [52] OpenAI. 2018. OpenAI Five DOTA. website. (June 2018). <https://blog.openai.com/openai-five/>
- [53] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*.
- [54] Peter Bailis et al. 2018. Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes. website. (Jan. 2018). <https://dawn.cs.stanford.edu/2018/01/11/index-baselines/>
- [55] Eleni Petraki, Stratos Idreos, and Stefan Manegold. 2015. Holistic Indexing in Main-memory Column-stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1153–1166. <https://doi.org/10.1145/2723372.2723719>
- [56] Bilal Piot, Matthieu Geist, and Olivier Pietquin. 2014. Boosted Bellman residual minimization handling expert demonstrations. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 549–564.
- [57] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. 2018. A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 551–566. <https://doi.org/10.1145/3183713.3196900>
- [58] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid training data creation with weak supervision. *Proceedings of the VLDB Endowment* 11, 3 (2017), 269–282.
- [59] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. [n. d.]. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. ([n. d.]). arXiv:cs.LG/1011.0686v3
- [60] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. 2006. COLT: Continuous On-line Tuning. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. ACM, New York, NY, USA, 793–795. <https://doi.org/10.1145/1142473.1142592>
- [61] 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 (ICML-15)*. 1889–1897.
- [62] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347* (2017).
- [63] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2135–2135.
- [64] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. [n. d.]. The Case for Automatic Database Administration using Deep Reinforcement Learning. ([n. d.]).

arXiv:cs.DB/http://arxiv.org/abs/1801.05643v1

- [65] Szymon Sidor and John Schulman. 2017. OpenAI Baselines. website. (2017). <https://blog.openai.com/openai-baselines-dqn/>
- [66] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.
- [67] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.
- [68] Gerald Tesauro, Rajarshi Das, Hoi Chan, Jeffrey Kephart, David Levine, Freeman Rawson, and Charles Lefurgy. 2007. Managing power consumption and performance of computing systems using reinforcement learning. In *Advances in Neural Information Processing Systems*. 1497–1504.
- [69] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. 2006. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing (ICAC '06)*. IEEE Computer Society, Washington, DC, USA, 65–73. <https://doi.org/10.1109/ICAC.2006.1662383>
- [70] The Apache Software Foundation. 2018. Apache Storm. <https://storm.apache.org/>. (2018).
- [71] Joshua Tobin, Wojciech Zaremba, and Pieter Abbeel. 2017. Domain Randomization and Generative Models for Robotic Grasping. *arXiv preprint arXiv:1710.06425* (2017).
- [72] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, and Aviv Tamar. 2017. A Machine Learning Approach to Routing. *arXiv preprint arXiv:1708.03074* (2017).
- [73] 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*. ACM, 1009–1024.
- [74] H. van Hasselt, A. Guez, and D. Silver. 2015. Deep Reinforcement Learning with Double Q-learning. *ArXiv e-prints* (Sept. 2015). arXiv:cs.LG/1509.06461
- [75] Ziyu Wang, Josh Merel, Scott Reed, Greg Wayne, Nando de Freitas, and Nicolas Heess. [n. d.]. Robust Imitation of Diverse Behaviors. ([n. d.]). arXiv:cs.LG/1707.02747v2
- [76] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8, 3-4 (1992), 229–256.

A SYNTHETIC CLIENT

We proceed to describe the synthetic data layout and query generation procedure used in the scalability and generalization experiments. Synthetic documents each contained 15 attributes with 6 strings, 6 integers of different ranges, 2 date fields, and 1 string array for full text.

Synthetic were generated by sampling between 1 and 3 attributes. For each attribute, a sub-expression was generated by sampling both a comparison operator and a value for the attribute, e.g.:

```
sub_expr := {$gt: {"attribute": "value"}}
```

Sub-expressions were then concatenated by uniformly sampling a logical operator, e.g.:

```
expr := {$or: [sub_expr, sub_expr, sub_expr]}
```

Finally, we sampled an aggregation operator from a discrete distribution where a limit without sort or count had 0.1 probability and sort/count 0.45 each, as sort and counts caused more difficult indexing decisions. If a sort aggregation was sampled, sorting was requested for each attribute in the query with 0.5 probability. Below is an example of a resulting query:

```
Q := find({'$or': [{'f2': {'$eq': 'centimeter'}}, {'f10': {'$gte': {'$date': 1394135731965}}]}))  
.limit(10)
```

In summary, our synthetic query client enables users to generate query shapes of varying difficulty and size to investigate indexing behavior.